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.

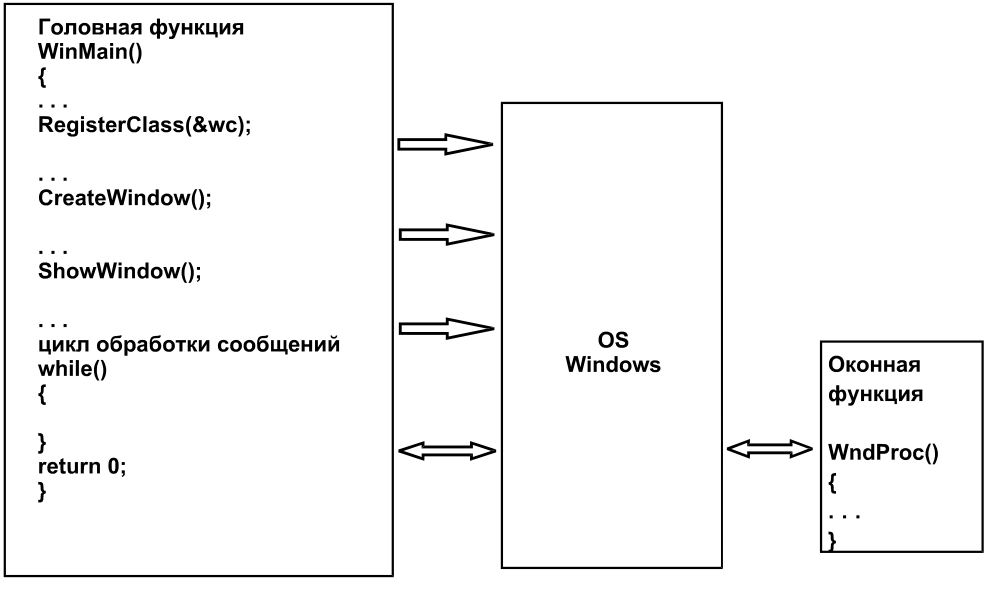


Рис. 1.1. Структура Windows-приложения

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.

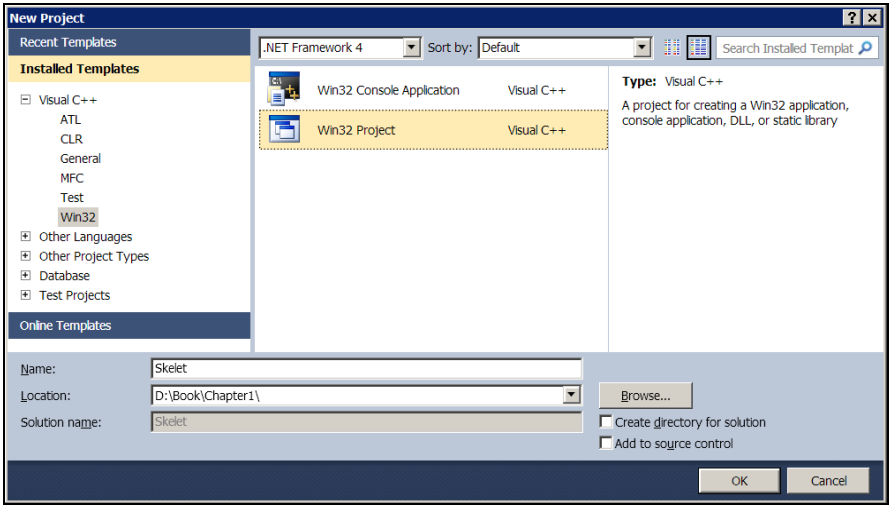


Рис. 1.2. Выбор типа проекта

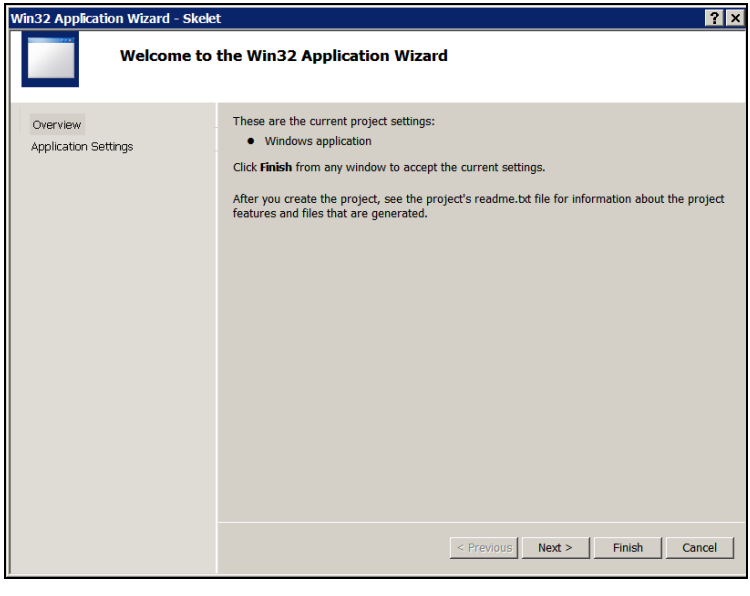


Рис. 1.3. Стартовое окно построителя приложения

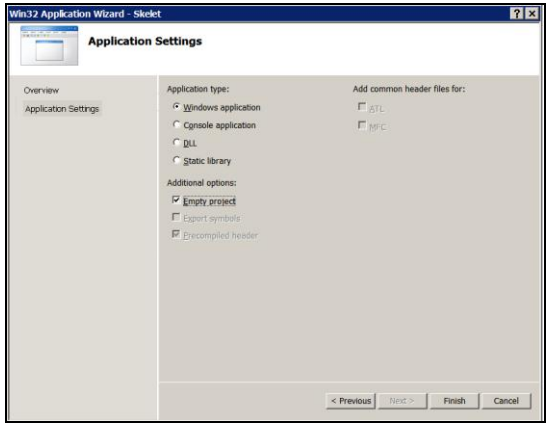


Рис. 1.4. Окно опций проекта

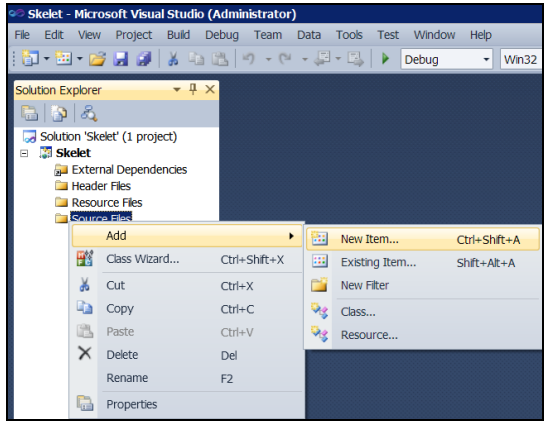


Рис. 1.5. Добавление к проекту нового объекта с помощью контекстного меню

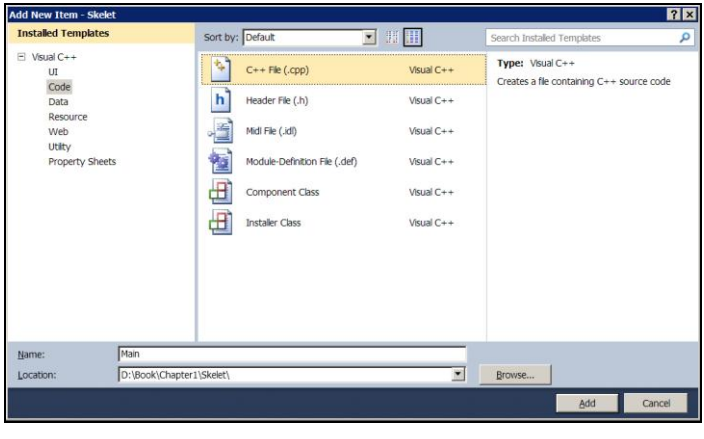


Рис. 1.6. Выбор шаблона объекта

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;
}
```

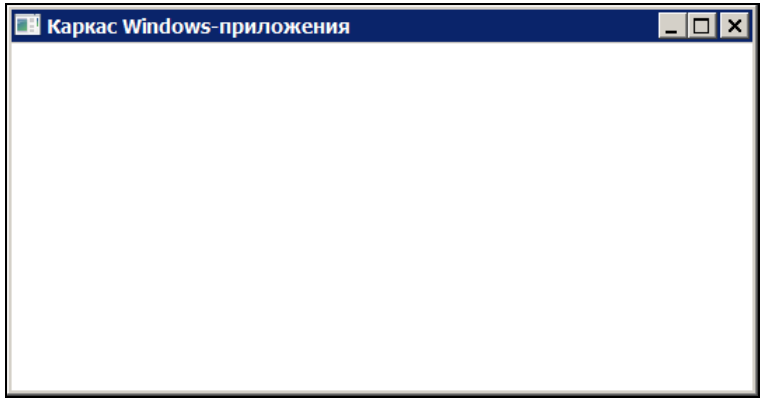


Рис. 1.7. Окно первой Windows-программы

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**.

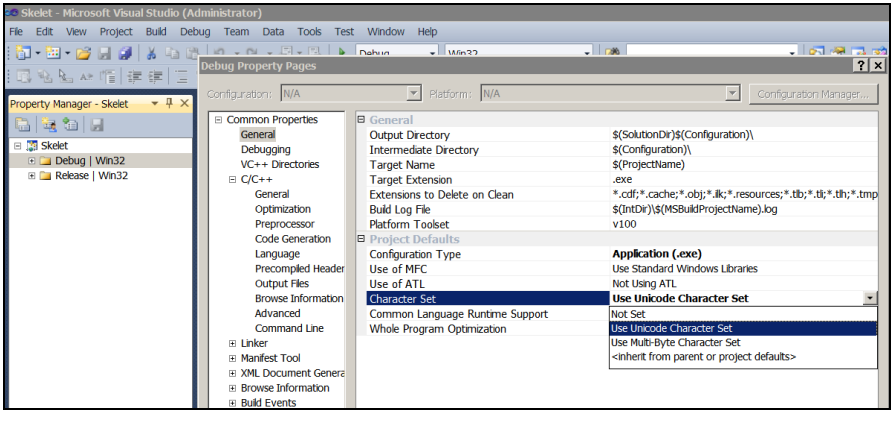


Рис. 1.8. Страница общих свойств проекта

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.

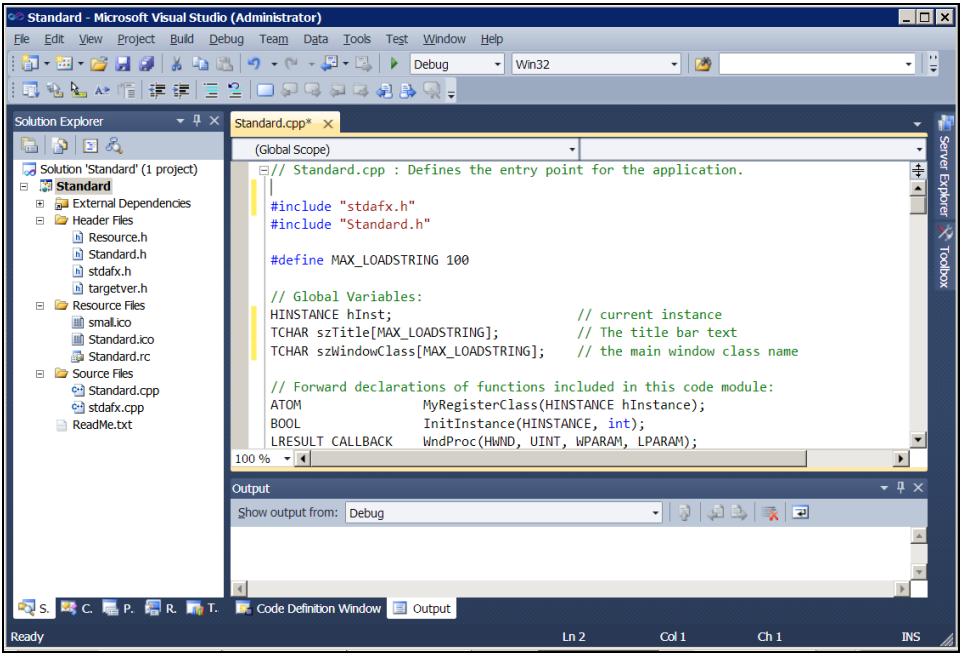


Рис. 1.9. Состав стандартной заготовки Win32-приложения

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.

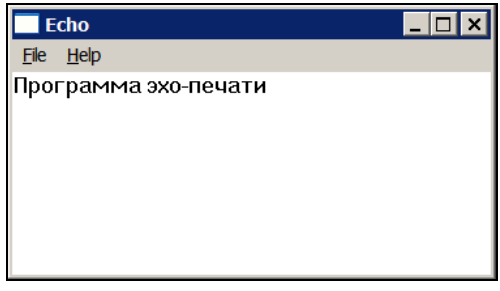


Рис. 1.10. Вид окна программы эхо-печати

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.

Еще одно обстоятельство нужно учитывать при выводе в окно — только пара функций `BeginPaint()` и `EndPaint()` удаляют из очереди сообщение `WM_PAINT` после его обработки. Если же сообщение не удалить, система будет "до бесконечности" перерисовывать окно.

Сообщение `WM_PAINT` является асинхронным и имеет достаточно низкий приоритет. Это приводит к тому, что может сложиться ситуация, когда окно получает сообщение `WM_PAINT`, а предыдущее сообщение еще не обработано. В этом случае операционная система "складывает" недействительные прямоугольники и объединяет перерисовку окна в одном сообщении.

Сообщение мыши

Нажатие на кнопки мыши приводит к ряду сообщений, вот некоторые из них:

- `WM_LBUTTONDOWN` — нажатие на левую кнопку мыши;
- `WM_LBUTTONUP` — отпускание левой кнопки мыши;
- `WM_RBUTTONDOWN` — нажатие на правую кнопку мыши;
- `WM_RBUTTONUP` — отпускание правой кнопки мыши;
- `WM_MOUSEMOVE` — перемещение мыши.

Рассмотрим в листинге 1.4 тестовую программу для обработки этих сообщений. Причем покажем, как можно осуществить вывод в окно непосредственно в обработчике сообщения о нажатии кнопки мыши. Для подобной задачи это вполне допустимо.

Листинг 1.4. Обработка сообщений нажатия на кнопку мыши

```
TCHAR *r_str = _T("Нажата правая кнопка мыши");
TCHAR *l_str = _T("Нажата левая кнопка мыши");
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        HDC hdc;
        int x, y;
        switch (message)
        {
        case WM_RBUTTONDOWN:
                x = LOWORD(lParam);
                y = HIWORD(lParam);
                hdc = GetDC(hWnd);
                TextOut(hdc, x, y, r_str, _tcsclen(r_str));
                ReleaseDC(hWnd, hdc);
                break;
        case WM_LBUTTONDOWN:
```

```
            x = LOWORD(lParam);
            y = HIWORD(lParam);
            hdc = GetDC(hWnd);
            TextOut(hdc, x, y, l_str, _tcsclen(l_str));
            ReleaseDC(hWnd, hdc);
            break;
        case WM_RBUTTONUP:
        case WM_LBUTTONUP:
            InvalidateRect(hWnd, NULL, TRUE);
            break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

Текстовые строки для вывода в окне мы описали на глобальном уровне, что в данном случае не обязательно. При обработке сообщения о нажатии кнопки мыши можно извлечь текущие координаты курсора из младшего и старшего слова `lParam`:

```
x = LOWORD(lParam);
y = HIWORD(lParam);
```

Контекст устройства получаем из функции `GetDC()`, передавая ей параметром дескриптор окна `hWnd`. Выводим текст по координатам курсора. Единственная проблема здесь — это вычисление длины строки. Если бы это была C-строка, мы обратились бы к функции `strlen()`, если это строка Unicode — `wcslen()`. А в общем случае используем макрос `_tcsclen`, который обеспечит правильный выбор. Соответствующий макрос размещен в файле `tchar.h`.

Освобождаем контекст устройства функцией `ReleaseDC()`.

При отпускании кнопки мыши хотелось бы очистить окно — это можно сделать, объявив его недействительным функцией `InvalidateRect()`. Причем мы вообще убрали обработчик сообщения `WM_PAINT`, нам достаточно того, что сделает функция обработки сообщения по умолчанию `DefWindowProc()`.

Иногда бывает полезно отследить состояние регистровых клавиш <Ctrl> и <Shift> при нажатии на кнопку мыши, коды этих клавиш передаются в переменную `wParam` и могут быть извлечены при обработке сообщения мыши, например:

```
case WM_LBUTTONDOWN:
        if (MK_SHIFT & wParam)
            if (MK_CONTROL & wParam)
      {
       //Нажаты клавиши Shift и Ctrl
      }
```

```
                else
        {
                //Нажата клавиша Shift
       }
        else
                if (MK_CONTROL & wParam)
       {
        //Нажата клавиша Ctrl
       }
                else
        {
                //Регистровые клавиши не нажаты
        }
break;
```

Можно упростить условный оператор, используя возможность логического сложения условий. Так, для проверки одновременного нажатия клавиш <Ctrl> и <Shift> при щелчке левой кнопки мыши запишем выражение:

```
if (MK_SHIFT|MK_CONTROL|MK_LBUTTON == wParam). . .
```

Создание окна

При создании окна генерируется сообщение `WM_CREATE` еще до его отображения, что позволяет производить некоторые начальные установки. Мы продемонстрируем использование этого сообщения в следующем примере.

Таймер

В программе можно установить один или несколько *таймеров*, которые позволяют производить отсчет времени. Таймер создается функцией `SetTimer()`:

```
UINT_PTR WINAPI SetTimer(HWND hWnd, UINT_PTR nIDEvent, UINT uElapse,
TIMERPROC lpTimerFunc);
```

Параметры функции:

- `hWnd` — дескриптор окна;
- `nIDEvent` — номер таймера, задается целым числом;
- `uElapse` — временной интервал таймера в миллисекундах;
- `lpTimerFunc` — указатель на функцию, вызываемую при обработке таймера.

Функция таймера определена следующим образом:

```
VOID CALLBACK TimerProc(
HWND hWnd,     // Дескриптор окна
UINT uMsg,     // WM_TIMER сообщение
UINT_PTR idEvent,    // Номер таймера
```

```
DWORD dwTime  // Текущее системное время
);
```

если `lpTimerFunc = NULL`, для обработки сообщения таймера вызывается функция окна-владельца. Обычно так и поступают.

Рассмотрим в листинге 1.5 задачу отсчета времени с момента начала работы приложения. Создадим таймер при создании окна в обработчике сообщения `WM_CREATE`:

```
SetTimer(hWnd, 1, 1000, NULL);
```

Таймеру присвоим номер 1, а интервал отсчета зададим в 1000 миллисекунд (1 секунда), в качестве же обработчика таймера используем оконную функцию.

В обработчике сообщения от таймера `WM_TIMER` увеличим переменную `t` на 1 секунду и объявляем окно недействительным:

```
t++;
InvalidateRect(hWnd, NULL, TRUE);
```

Строку текста для вывода в окне сформируем в обработчике сообщения `WM_PAINT`:

```
_tcscat(str, _itot(t, s, 10));
TextOut(hdc, 0, 0, str, _tcsclen(str));
```

К строке `str` добавим значение переменной `t`, для чего преобразуем ее в символьный вид. Здесь мы опять воспользовались макросами из `tchar.h`:

- C-строка — `_tcscat` преобразуется в `strcat`, `_itot` в `_itoa`;
- Unicode — `_tcscat` преобразуется в `wcscat`, `_itot` в `_itow`.

В функциях `_itoa()`, `_itow()` для преобразования целого числа в строку, третий параметр — основание системы счисления 10, возвращаемое значение — указатель на строку результата `s`. Функцией `strcat()` или `wcscat()` "приклеим" к исходной строке с текстом `"Секунды: "` и выведем полученную строку функцией `TextOut()`.

Перед закрытием окна уничтожим таймер функцией `KillTimer()`:

```
BOOL WINAPI KillTimer(HWND hWnd, UINT_PTR uIDEvent);
```

Второй параметр функции `uIDEvent` — номер таймера — имеет тип `UINT_PTR`, эквивалентный `unsigned int` для 32-разрядных приложений и введен для совместимости с 64-разрядными приложениями, где он уже ассоциируется с `unsigned __int64`.

Листинг 1.5. Оконная функция для таймера

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
        PAINTSTRUCT ps;
        HDC hdc;
        static int t;
        TCHAR s[10], str[20] = _T("Секунды: ");
        switch (message)
```

```
        {
        case WM_CREATE :
                SetTimer(hWnd, 1, 1000, NULL);
                break;
        case WM_TIMER :
                t++;
                InvalidateRect(hWnd, NULL, TRUE);
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                _tcscat(str+9, _itot(t, s, 10));
                TextOut(hdc, 0, 0, str, _tcsclen(str));
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY:
                KillTimer(hWnd, 1);
                PostQuitMessage(0);
                break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

Рисование в окне

При рисовании в окне можно использовать достаточно большой набор GDI-функций. Действуют следующие соглашения: все линии рисуются текущим пером, а области заполняются текущей кистью. Как устанавливать перо и кисть мы обсудим позднее, а пока будем рисовать пером и кистью по умолчанию. Все эти функции возвращают ненулевое значение в случае успешного выполнения, и 0 — в случае ошибки. Графические функции работают с логическими координатами, одна логическая единица по умолчанию равна одному пикселу.

> ***Примечание***
>
> *Пиксел* — точка на экране монитора. Размер зависит от выбранного разрешения.

По умолчанию устанавливается графический режим с началом координат в левом верхнем углу клиентской области окна, ось *x* направлена вправо, ось *y* — вниз.

Рисование линии

Для того чтобы нарисовать линию, используется функция `LineTo()`:

```
BOOL WINAPI LineTo(HDC hdc, int x, int y);
```

где `hdc` — дескриптор контекста устройства, `x, y` — координаты конца линии. Начало линии определяется положением текущей позиции рисования.

При создании окна текущая позиция определяется в начале координат (0,0). Если же необходимо нарисовать линию из другой точки, положение текущей позиции рисования может быть изменено.

Установка текущей позиции

Функция MoveToEx():

```
BOOL WINAPI MoveToEx(HDC hdc, int x, int y, LPPOINT oldPoint);
```

устанавливает текущую позицию в точку с координатами (x,y) и передает в структуру POINT координаты предыдущей позиции.

```
typedef struct tagPOINT
{   LONG  x;
    LONG  y;
} POINT;
```

Если последний параметр NULL, то предыдущие координаты не сохраняются.

Обычно эти функции используются в паре, обе они возвращают ненулевое значение, в случае успешного завершения, и 0 — в случае ошибки.

Например, если нам нужно нарисовать линию между двумя точками, можно сделать это так:

```
MoveToEx(hdc, 50, 100, NULL);
LineTo(hdc, 350, 100);
```

Определение размера клиентской области

При выводе данных в окно необходимо определить фактический размер этого окна, точнее, его клиентской области. Для решения данной задачи существует функция GetClientRect():

```
BOOL WINAPI GetClientRect(HWND hWnd, LPRECT lpRect);
```

Размеры клиентской области окна возвращаются в структуре RECT:

```
typedef struct tagRECT
{   LONG    left;     //x - координата левого верхнего угла
    LONG    top;      //y - координата левого верхнего угла
    LONG    right;    //x - координата правого нижнего угла
    LONG    bottom;   //y - координата правого нижнего угла
} RECT;
```

> ***Примечание***
>
> В данном контексте применения функции GetClientRect() поля left и top структуры RECT будут равны 0, а поля right и bottom будут равны ширине и высоте клиентской области окна.

Имеется, однако, более простое решение для определения клиентской области окна. Воспользуемся сообщением WM_SIZE, которое генерируется системой при создании окна после сообщения WM_CREATE и при каждом изменении его размеров:

```
case WM_SIZE:
       sx = LOWORD(lParam);  //ширина окна
```

```
            sy = HIWORD(lParam);  //высота окна
            break;
```

`lParam` возвращает в младшем слове ширину, а в старшем слове — высоту клиентской области окна. Если мы опишем две статические переменные целого типа `sx` и `sy`, то всегда будем иметь текущую ширину и высоту окна.

Теперь у нас достаточно информации, чтобы рассмотреть простой пример, приведенный на рис. 1.11, где мы построили сетку горизонтальными и вертикальными линиями с шагом в 1/10 размера окна, см. листинг 1.6. Причем сетка должна перестраиваться при изменении размеров окна.

Листинг 1.6. Рисование сетки

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        PAINTSTRUCT ps;
        HDC hdc;
        int x, y;
        static int sx, sy;
        switch (message)
        {
        case WM_SIZE:
                sx = LOWORD(lParam);
                sy = HIWORD(lParam);
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                for (x = 0; x < sx; x += sx/10)
                {
                        MoveToEx(hdc, x, 0, NULL);
                        LineTo(hdc, x, sy);
                }
                for (y = 0; y < sy; y += sy/10)
                {
                        MoveToEx(hdc, 0, y, NULL);
                        LineTo(hdc, sx, y);
                }
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

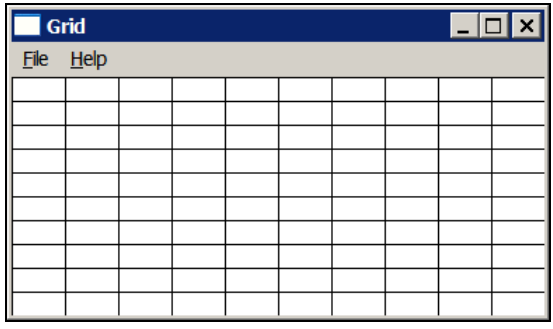


Рис. 1.11. Построение сетки

Рисование прямоугольника

Нарисовать прямоугольник можно при помощи функции:

```
BOOL WINAPI Rectangle(HDC hdc, int x1, int y1, int x2, int y2);
```

где `(x1,y1)` — координаты левого верхнего угла прямоугольника, а `(x2,y2)` — координаты правого нижнего угла. Прямоугольник заполняется текущей кистью, поэтому, если в предыдущем примере (см. листинг 1.6) добавить в обработчик сообщения `WM_PAINT` следующую строку:

```
Rectangle(hdc, sx/4, sy/4, sx*3/4, sy*3/4);
```

мы получим прямоугольник в половину окна, расположенный по центру.

Рисование эллипса

Функция для отображения эллипса имеет те же параметры, поскольку эллипс определяется ограничивающим его прямоугольником:

```
BOOL WINAPI Ellipse(HDC hdc, int x1, int y1, int x2, int y2);
```

внутренняя область также заполняется текущей кистью. Таким образом, если мы добавим строку:

```
Ellipse(hdc, sx/3, sy/4, sx*2/3, sy*3/4);
```

то получим внутри прямоугольника эллипс (рис. 1.12).

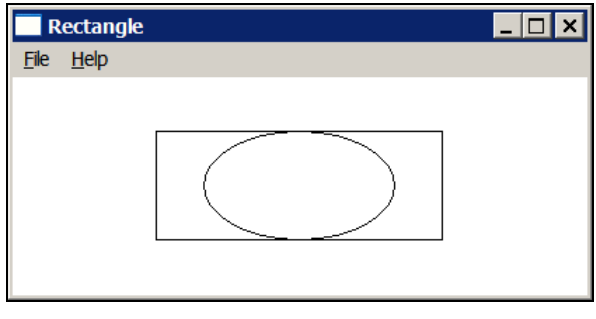


Рис. 1.12. Прямоугольник и эллипс в ¼ окна

Рисование точки

Функция `SetPixel()` позволяет вывести в окно одну точку (пиксел):

```
COLORREF WINAPI SetPixel(HDC hdc, int x, int y, COLORREF color);
```

где `hdc` — контекст устройства, `(x,y)` — координаты точки, `color` — цвет точки. Функция возвращает прежний цвет точки в случае успешного завершения, и `-1` при ошибке.

Тип `COLORREF` представляет длинное целое число, где три младших байта кодируют соответственно *красный*, *синий*, *зеленый* цвета. Старший байт всегда 0. Числовое значение байта определяет интенсивность каждого цвета, а цвет точки определяется смешиванием трех базовых цветов. Для получения нужного цвета проще всего воспользоваться предопределенным в файле `wingdi.h` макросом `RGB()`, который описан как:

```
#define RGB(r,g,b)
((COLORREF)(((BYTE)(r)|((WORD)((BYTE)(g))<<8))|(((DWORD)(BYTE)(b))<<16)))
```

Например, если нам нужен чисто зеленый цвет, мы могли бы определить переменную `color` так:

```
COLORREF color = RGB(0,255,0);
```

Эту функцию редко используют для рисования, поскольку точка в 1 пиксел выглядит достаточно мелко.

С помощью листинга 1.7, используя рассмотренные выше графические примитивы, построим кардиоиду, уравнение которой в полярных координатах:

$$\rho = 1 - \cos(\varphi).$$

Листинг 1.7. Построение кардиоиды

```
#define _USE_MATH_DEFINES
#include <cmath>
const int R = 100;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        static int sx, sy;
        int a, b, x, y;              //Экранные координаты
        double angle;         //Физические координаты
        switch (message)
        {
        case WM_SIZE:
               sx = LOWORD(lParam);  //Ширина окна
               sy = HIWORD(lParam);  //Высота окна
               break;
        case WM_PAINT:
```

```
            hdc = BeginPaint(hWnd, &ps);
            a = sx/2;
            b = sy/2;
            MoveToEx(hdc, 0, b, NULL);
            LineTo(hdc, sx, b);
            MoveToEx(hdc, a, 0, NULL);
            LineTo(hdc, a, sy);
            MoveToEx(hdc, a, b, NULL);
            for (angle = 0.0; angle < 2*M_PI; angle += 0.1)
            {
                  x = a + R*(1 - cos(angle))*cos(angle);
                  y = b - R*(1 - cos(angle))*sin(angle);
                  LineTo(hdc, x, y);
            }
            EndPaint(hWnd, &ps);
            break;
      case WM_DESTROY: PostQuitMessage(0); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

Нам необходим файл включений `cmath`, где определены основные математические функции. Для доступа к константам, заданным в этом файле, необходимо разрешить их использование, определив символическую константу:

```
#define _USE_MATH_DEFINES
```

Теперь описываем экранные координаты типа `int` и физическую координату типа `double`. Размеры окна определим, как обычно, в сообщении `WM_SIZE`.

В сообщении `WM_PAINT` строим координатные оси, устанавливаем текущую координату в центре окна и в цикле строим фигуру ломаной линией (рис. 1.13). В качестве масштабного множителя используем константу `R`, равную 100.

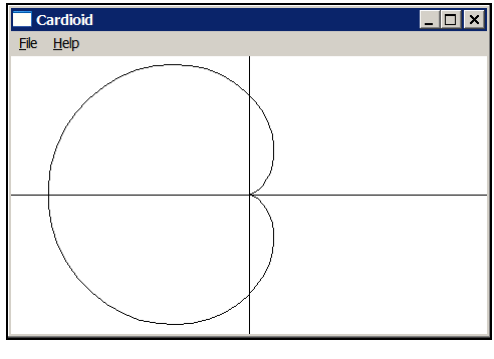


Рис. 1.13. Кардиоида

Обратите внимание, что координаты `x` и `y` мы отсчитываем от центра окна, причем для координаты `y` выражение записано со знаком минус, поскольку ось *y* в окне направлена вниз.

Здесь мы воспользовались побочным эффектом функции `LineTo()`, которая устанавливает текущей координатой конечную точку линии.

Создание пера

Рассмотрев предыдущие примеры, отметим их общий недостаток: все построения производились черной линией толщиной в 1 пиксел — это *перо* по умолчанию, которым и строились все линии.

Можно создать *пользовательское перо* при помощи функции `CreatePen()`:

```
HPEN WINAPI CreatePen(int style, int width, COLORREF color);
```

Параметр `style` определяется макросом и задает тип линии, список его возможных значений приведен в табл. 1.2.

Таблица 1.2. *Макросы, определяющие тип линии*

Макрос	Тип линии
`PS_DASH`	Штриховая линия (`---`)
`PS_DASHDOT`	Штрихпунктирная линия (`-.-`)
`PS_DASHDOTDOT`	Штрихпунктирная линия (`-..-`)
`PS_DOT`	Точечная линия (`...`)
`PS_INSIDEFRAME`	Сплошная линия внутри области
`PS_NULL`	Прозрачное перо
`PS_SOLID`	Сплошная линия

Параметр `width` задает ширину пера в логических единицах. Если ширина пера установлена в 0, то линии рисуются в 1 пиксел, независимо от режима рисования.

Примечание

1. По умолчанию одна логическая единица равна одному пикселу.
2. Пунктирные линии рисуются только при ширине пера в 1 лог. единицу.

Параметр `color` задает цвет пера. Как задать цвет макросом `RGB()` мы уже рассматривали, но далее, в табл. 1.3, приведем список "чистых цветов".

Примечание

Это дань прошлому, когда перу можно было присвоить только чистый цвет, однако рекомендуется придерживаться этих соглашений.

Таблица 1.3. *Значения параметров макроса RGB для генерации чистых цветов*

Цвет	**Red, Green, Blue**
Темно-красный	128, 0, 0
Светло-красный	255, 0, 0
Темно-зеленый	0, 128, 0
Светло-зеленый	0, 255, 0
Темно-синий	0, 0, 128
Светло-синий	0, 0, 255
Темно-желтый	128, 128, 0
Светло-желтый	255, 255, 0
Темно-бирюзовый	0, 128, 128
Светло-бирюзовый	0, 255, 255
Темно-сиреневый	128, 0, 128
Светло-сиреневый	255, 0, 255
Черный	0, 0, 0
Темно-серый	128, 128, 128
Светло-серый	192, 192, 192
Белый	255, 255, 255

После создания *пера* выбирается как *текущее* с помощью функции `SelectObject()`:

```
HGDIOBJ WINAPI SelectObject(HDC hdc, HGDIOBJ handle);
```

где указываются контекст устройства вывода и дескриптор созданного пера. Создадим синее перо толщиной 2 пиксела для построения координатных осей:

```
HPEN hpen1 = CreatePen(PS_SOLID, 2, RGB(0, 0, 255));
SelectObject(hdc, hpen1);
```

После этого нетрудно убедиться, что линии действительно стали синими и в два раза толще.

> ***Примечание***
>
> *Активным* может быть только одно *перо*, выбранное функцией `SelectObject()` в качестве текущего, или черное перо по умолчанию, если такой выбор не сделан.

Можно воспользоваться готовыми перьями Windows. В этом случае дескриптор пера получим при помощи функции `GetStockObject()`:

```
HGDIOBJ WINAPI GetStockObject(int);
```

указывая в качестве параметра соответствующую константу:

- `BLACK_PEN` — черное перо;
- `WHITE_PEN` — белое перо;
- `NULL_PEN` — прозрачное перо.

Примечание

Поскольку функция `GetStockObject()` может возвращать различные объекты, то для возвращаемого значения необходимо явно указывать преобразование типа, например: `HPEN hpen = (HPEN)GetStockObject(BLACK_PEN);`.

Перед выходом из программы все созданные перья необходимо *удалить* (за исключением системных), поскольку каждое вновь созданное перо занимает определенные ресурсы, которые автоматически не освобождаются при переходе на другое перо. Сделать это нужно функцией `DeleteObject()`:

```
BOOL WINAPI DeleteObject(HGDIOBJ handle);
```

Напишем программу вывода графика синусоиды с учетом возможности создавать и менять перья в процессе вывода. Выведем координатные оси синей линией толщиной 2 пиксела, а кривую — сплошной красной линией той же толщины: в этом случае мы можем выбрать шаг цикла по оси *x* в 3 и более пиксела. Воспользуемся тем, что функция `LineTo()` перемещает текущую позицию в конец линии. Создание перьев вынесем отдельно в сообщение `WM_CREATE`, а их удаление — в `WM_DESTROY`, см. листинг 1.8. Результат работы программы представлен на рис. 1.14.

Листинг 1.8. Синусоида

```
#define _USE_MATH_DEFINES
#include <cmath>
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        static int sx, sy;
        static HPEN hpen1, hpen2;
        int a, b, x_scr, y_scr;      // Экранные координаты
        double x, h;          // Физические координаты
        switch (message)
        {
        case WM_CREATE :
                hpen1 = CreatePen(PS_SOLID, 2, RGB(0, 0, 255));
                hpen2 = CreatePen(PS_SOLID, 2, RGB(255, 0, 0));
                break;
        case WM_SIZE:
                sx = LOWORD(lParam);  //Ширина окна
                sy = HIWORD(lParam);  //Высота окна
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                a = sx/2;                    //Координаты
                b = sy/2;                    //центра окна
```

```
            SelectObject(hdc,hpen1);     //Синее перо
            MoveToEx(hdc, 0, b, NULL);
            LineTo(hdc, sx, b);
            MoveToEx(hdc, a, 0, NULL);
            LineTo(hdc, a, sy);
            MoveToEx(hdc, 0, b, NULL);
            SelectObject(hdc,hpen2);     //Красное перо
            h = 3*M_PI/a;                //Шаг по оси x
            for (x = -M_PI, x_scr = 0; x < M_PI; x += h)
            {
                  x_scr = (x + M_PI)*a/M_PI;
                  y_scr = b - b*sin(x);
                  LineTo(hdc, x_scr, y_scr);
            }
            EndPaint(hWnd, &ps);
            break;
      case WM_DESTROY:
            DeleteObject(hpen1);
            DeleteObject(hpen2);
            PostQuitMessage(0);
            break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

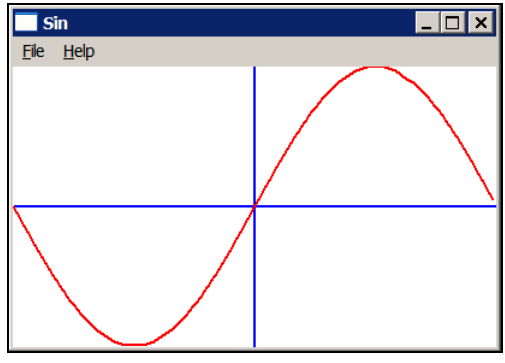


Рис. 1.14. Вывод графика функции сплошной линией

Настройка графического режима

Как показал предыдущий пример, строить график функций в такой системе координат довольно неудобно. Действительно, нам всегда нужно помнить, что система координат имеет начало в левом верхнем углу окна, а ось *y* направлена вниз. Такая

система координат хороша для вывода текста, однако графики было бы удобнее выполнять в декартовой системе координат с началом в центре окна. Система программирования предоставляет набор функций, которые позволят установить такую систему координат. Можно задать как логические размеры области вывода, так и ее физические размеры, а также новое начало координат и направление осей. Но вначале рассмотрим возможные режимы отображения графики.

Режимы отображения

По умолчанию устанавливается графический режим `MM_TEXT`, при котором одна логическая единица равна 1 экранному пикселу. Этот режим обычно используется для вывода текста.

> ***Примечание***
>
> Установка режима отображения не меняет ни размера окна, ни разрешающей способности, а лишь изменяет способ преобразования логических координат в экранные пикселы.

Для установки текущего режима отображения используется функция `SetMapMode()`:

```
int WINAPI SetMapMode(HDC hdc, int mode);
```

где `mode` определяет графический режим и принимает одно из заданных значений:

- `MM_TEXT` — одна логическая единица равна 1 пикселу;
- `MM_LOMETRIC` — одна логическая единица равна 0,1 миллиметра;
- `MM_HIMETRIC` — одна логическая единица равна 0,01 миллиметра;
- `MM_LOENGLISH` — одна логическая единица равна 0,01 дюйма;
- `MM_HIENGLISH` — одна логическая единица равна 0,001 дюйма;
- `MM_TWIPS` — одна логическая единица равна 1/12 точки принтера, 1/1440 дюйма;
- `MM_ISOTROPIC` — режим отображения логических единиц с одинаковым масштабированием по осям координат;
- `MM_ANISOTROPIC` — режим отображения логических единиц с различным масштабированием по осям координат.

Определение логических размеров окна

При использовании режимов отображения `MM_ISOTROPIC` и `MM_ANISOTROPIC` необходимо задать размеры окна в логических координатах. Именно с этими логическими координатами мы будем обращаться к графическим функциям.

> ***Примечание***
>
> При задании логических размеров окна лучше задавать их как можно ближе к соотношению реальных размеров окна, иначе толщина вертикальных и горизонтальных линий может существенно отличаться.

Логические размеры окна определяются функцией `SetWindowExtEx()`:

```
BOOL WINAPI SetWindowExtEx(HDC hdc, int x, int y, LPSIZE size);
```

Функция принимает контекст устройства `hdc` и логические размеры `x` и `y`; возвращает ненулевое значение при успешном завершении. Если последним параметром передан адрес структуры `SIZE`, там будет сохранена информация о предыдущем размере окна, если эта информация не нужна, указывается `NULL`.

Определение области вывода

Для режимов отображения `MM_ISOTROPIC` и `MM_ANISOTROPIC` необходимо указать также физический размер области экрана в пикселах. Этот размер может быть как меньше, так и больше фактического размера окна и определяется вызовом функции `SetViewportExtEx()`:

```
BOOL WINAPI SetViewportExtEx(HDC hdc, int x, int y, LPSIZE size);
```

Эта функция имеет такой же набор параметров, что и предыдущая, но `x` и `y` в данном случае определяют физические размеры области вывода в пикселах.

> ***Примечание***
>
> Задание размера окна отрицательным числом приводит к изменению направления оси координат.

Задание начала системы координат

Осталось определить новое начало системы координат. По умолчанию начало координат, т. е. точка с координатами (0; 0), находится в левом верхнем углу клиентской области окна. Мы можем установить начало координат в любой точке области вывода при помощи функции `SetViewportOrgEx()`:

```
BOOL WINAPI SetViewportOrgEx(HDC hdc, int x, int y, LPPOINT point);
```

Функция принимает контекст устройства `hdc`, координаты нового начала координат `(x,y)` в экранных координатах и указатель на структуру `POINT` для хранения "старого" начала координат, или `NULL`, тогда предыдущее начало координат не сохраняется.

Для демонстрации последних рассмотренных функций выведем в одном окне два графика: синусоиду и восьмилепестковую розу (рис. 1.15), см. листинг 1.9.

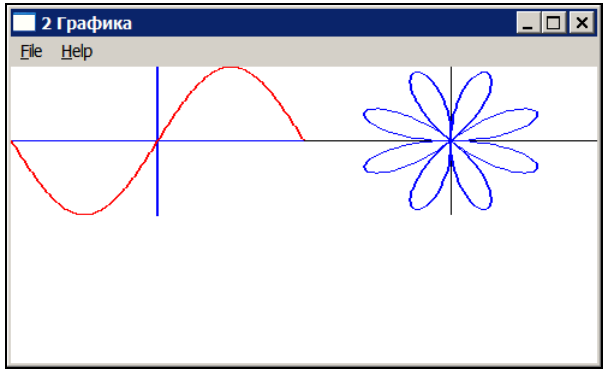


Рис. 1.15. Два графика в одном окне

У рассматриваемой функции `sin(x)` аргумент `x` меняется в диапазоне [–π; π], а сама функция принимает значения в диапазоне [–1; 1]. Если бы мы задали логический размер окна [6.28; 2], то вряд ли мы получили бы в окне вывода что-либо разумное. Поэтому введем масштабный множитель и зададим размер так: [628; 400], т. е. умножим *x*-координату на 100, а *y*-координату на 200.

Вторая фигура — восьмилепестковая роза, в полярных координатах:

$$\rho = a\sin(4\varphi).$$

Построим ее, сдвинув начало системы координат по оси *x*, и используем масштабный множитель 200.

Если сравнить графические построения, приведенные в листинге 1.9, с предыдущей задачей, можно убедиться, что вычисления упростились и стали более наглядными.

Листинг 1.9. Вывод графиков в локальной системе координат

```
#define _USE_MATH_DEFINES
#include <cmath>
const int WIDTH = 314;
const int HEIGHT = 200;
const double K = 4.0;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        static int sx, sy;
        static HPEN hpen1, hpen2;
        int x_scr, y_scr;     //Экранные координаты
        double x;             //Физические координаты
        switch (message)
        {
        case WM_CREATE:       //Создаем перья
               hpen1 = CreatePen(PS_SOLID,4,RGB(0, 0, 255));
               hpen2 = CreatePen(PS_SOLID,4,RGB(255, 0, 0));
               break;
        case WM_SIZE: //Определяем физические размеры окна
               sx = LOWORD(lParam);
               sy = HIWORD(lParam);
               break;
        case WM_PAINT:
               hdc = BeginPaint(hWnd, &ps);
               //Установка режима
               SetMapMode(hdc, MM_ANISOTROPIC);
               //Установка логических размеров вывода, ось y направлена вверх
```

```
            SetWindowExtEx(hdc, 2*WIDTH, -2*HEIGHT, NULL);
            //Установка физических размеров на четверть окна
            SetViewportExtEx(hdc, sx/2, sy/2, NULL);
            // Установка начала координат
            SetViewportOrgEx(hdc, sx/4, sy/4, NULL);
            SelectObject(hdc, hpen1);           //Синее перо
            MoveToEx(hdc, -WIDTH, 0, NULL);     //Рисуем координатные оси
            LineTo(hdc, WIDTH, 0);
            MoveToEx(hdc, 0, HEIGHT, NULL);
            LineTo(hdc, 0, -HEIGHT);
            SelectObject(hdc, hpen2);           //Красное перо
            MoveToEx(hdc, -WIDTH, 0, NULL);
            for (x = -M_PI, x_scr = -WIDTH; x < M_PI; x += 0.03, x_scr += 3)
            {
                  y_scr = HEIGHT*sin(x);
                  LineTo(hdc, x_scr, y_scr);
            }
            // Перенос начала координат
            SetViewportOrgEx(hdc, sx*3/4, sy/4, NULL);
            SelectObject(hdc, GetStockObject(BLACK_PEN)); //Черное перо
            MoveToEx(hdc, -WIDTH, 0, NULL);     //Рисуем координатные оси
            LineTo(hdc, WIDTH, 0);
            MoveToEx(hdc, 0, HEIGHT, NULL);
            LineTo(hdc, 0, -HEIGHT);
            SelectObject(hdc, hpen1);    //Синее перо
            MoveToEx(hdc, 0, 0, NULL);   //Текущая точка в начале координат
            for (double angle = 0.0; angle < 2*M_PI; angle += 0.02)
            {
                  x_scr = HEIGHT*sin(K*angle)*cos(angle);
                  y_scr = HEIGHT*sin(K*angle)*sin(angle);
                  LineTo(hdc, x_scr, y_scr);
            }
            EndPaint(hWnd, &ps);
            break;
      case WM_DESTROY:
            DeleteObject(hpen1);
            DeleteObject(hpen2);
            PostQuitMessage(0);
            break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

Контекст устройства вывода хранит информацию для преобразования логических координат в координаты устройства (физические координаты):

$$X_{\text{физ}} = X_{\text{лог}} \cdot K_x + X_0, \qquad K_x = \frac{\text{физический размер области вывода по оси } x}{\text{логический размер области вывода по оси } x},$$

$$Y_{\text{физ}} = Y_{\text{лог}} \cdot K_y + Y_0, \qquad K_y = \frac{\text{физический размер области вывода по оси } y}{\text{логический размер области вывода по оси } y}.$$

Рассмотренные выше функции изменяют значение масштабных коэффициентов K_x, K_y и начала координат (X_0,Y_0). Для этих целей хватило бы и одной функции, но так уж сложилось.

Из этих формул должно быть ясно, чем отличается режим `MM_ANISOTROPIC` от `MM_ISOTROPIC`. В режиме `MM_ISOTROPIC` масштабный коэффициент одинаков для обеих осей.

Создание кисти

Кисть используется для заполнения фона окна или замкнутой области внутри окна. В каждый момент времени активной может быть лишь одна кисть либо созданная пользователем, либо кисть по умолчанию, зарегистрированная в классе окна. Кисть может быть сплошной или штриховой.

Сплошная кисть создается при помощи функции `CreateSolidBrush()`:

```
HBRUSH CreateSolidBrush(COLORREF color);
```

и устанавливается функцией `SelectObject()`, например:

```
HBRUSH hbrush = CreateSolidBrush(RGB(255, 255, 0));
SelectObject(hdc, hbrush);
```

Штриховая кисть создается функцией `CreateHatchBrush()`:

```
HBRUSH CreateHatchBrush(int nIndex, COLORREF color);
```

где первый параметр `nIndex` определяет тип штриховки и может принимать следующие значения:

- `HS_BDIAGONAL` — слева направо и снизу вверх;
- `HS_CROSS` — горизонтальная и вертикальная штриховка;
- `HS_DIAGCROSS` — под углом в 45 градусов;
- `HS_FDIAGONAL` — слева направо и сверху вниз;
- `HS_HORIZONTAL` — горизонтальная штриховка;
- `HS_VERTICAL` — вертикальная штриховка.

Второй параметр — цвет линии штриховки. Вот так можно создать штриховую кисть темно-зеленого цвета:

```
HBRUSH hbrush = CreateHatchBrush(HS_CROSS, RGB(0, 128, 0));
SelectObject(hdc,hbrush);
```

Кисть может быть удалена функцией `DeleteObject()`.

Примечание

Нельзя удалять текущую кисть, поэтому перед удалением ее нужно освободить заданием новой текущей кисти, в качестве которой можно использовать одну из системных кистей.

Имеется набор системных кистей, которые могут быть получены функцией `GetStockObject()`, их идентификаторы:

- `BLACK_BRUSH` — черная кисть;
- `DKGRAY_BRUSH` — темно-серая кисть;
- `GRAY_BRUSH` — серая кисть;
- `LTGRAY_BRUSH` — светло-серая кисть;
- `NULL_BRUSH` — нулевая кисть;
- `WHITE_BRUSH` — белая кисть.

Примечание

Системные кисти не нужно удалять при завершении работы.

После всего вышеизложенного представим простую тестовую программу (листинг 1.10), которая показывает заполнение графических объектов сплошной и штриховыми кистями.

Листинг 1.10. Тест для демонстрации кистей

```
HBRUSH hbrush, h_brush[6];
TCHAR *str = _T("сплошное заполнение");
TCHAR *hstr[] = {_T("HS_BDIAGONAL – слева направо и снизу вверх"),
                 _T("HS_CROSS - горизонтальная и вертикальная штриховка"),
        _T("HS_DIAGCROSS - под углом в 45 градусов"),
        _T("HS_FDIAGONAL - слева направо и сверху вниз"),
        _T("HS_HORIZONTAL - горизонтальная штриховка"),
        _T("HS_VERTICAL - вертикальная штриховка")};

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        int i;
        int nIndex[] = {HS_BDIAGONAL, HS_CROSS, HS_DIAGCROSS, HS_FDIAGONAL,
                HS_HORIZONTAL, HS_VERTICAL};
        switch (message)
        {
        case WM_CREATE :
```

```
            hbrush = CreateSolidBrush(RGB(255, 255, 0));
            for (i = 0; i < 6; i++)
            h_brush[i] = CreateHatchBrush(nIndex[i], RGB(0, 128, 0));
            break;
      case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            SelectObject(hdc, hbrush);
            Ellipse(hdc, 1, 1, 40, 40);
            TextOut(hdc, 50, 11, str, _tcsclen(str));
            for (i = 0; i < 6; i++)
            {
                  SelectObject(hdc, h_brush[i]);
                  Rectangle(hdc, 1, 41+i*40, 40, 80+i*40);
                  TextOut(hdc, 50, 51+i*40, hstr[i], _tcsclen(hstr[i]));
            }
            EndPaint(hWnd, &ps);
            break;
      case WM_DESTROY :
            DeleteObject(hbrush);
            for (i = 0; i < 6; i++) DeleteObject(h_brush[i]);
            PostQuitMessage(0);
            break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

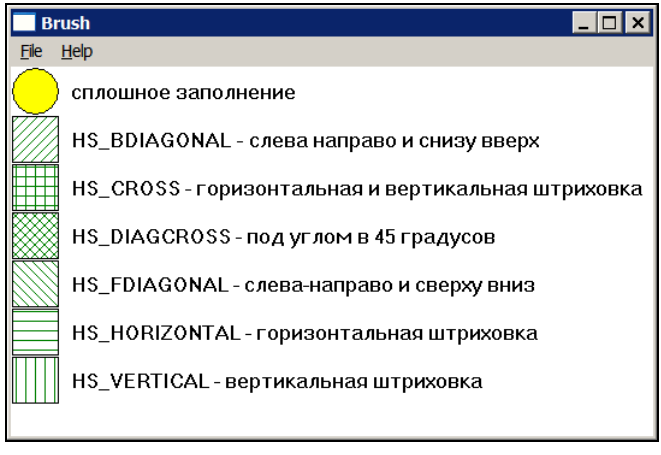


Рис. 1.16. Заполнение различными кистями

Здесь мы определили дескрипторы кистей и текстовые строки на глобальном уровне, причем определили их в виде массивов, что позволило организовать цикл для перебора всех 6 штриховых кистей.

В результате работы программы получим окно, изображенное на рис. 1.16.

Прямоугольники, регионы и пути

Графическая библиотека интерфейса GDI (Graphic Device Interface) имеет большой набор функций, работающих с *регионами* (regions) и *путями* (paths). Эти функции были введены в Windows NT и существенно улучшают возможности графических построений. Также были добавлены функции для работы с прямоугольниками.

Прямоугольники

Прямоугольники (rectangles) широко используются в графике, поэтому представляет интерес набор функций для манипуляций с ними:

- `BOOL WINAPI CopyRect(LPRECT lprcDst, CONST RECT *lprcSrc);`

  Функция копирует один прямоугольник в другой.

- `BOOL EqualRect(CONST RECT *lprc1,CONST RECT *lprc2);`

  Функция определяет, равны ли два прямоугольника, сравнивая их координаты.

- `BOOL InflateRect(LPRECT lprc, int dx, int dy);`

  Функция увеличивает или уменьшает ширину и высоту указанного прямоугольника.

- `int WINAPI FillRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr);`

  Функция заполняет прямоугольник кистью `hbr`.

- `int WINAPI FrameRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr);`

  Функция рисует контур прямоугольника кистью `hbr`.

- `BOOL WINAPI IntersectRect(LPRECT lprcDst, CONST RECT *lprcSrc1, CONST RECT *lprcSrc2);`

  Функция выполняет пересечение двух прямоугольников.

- `BOOL WINAPI InvertRect(HDC hDC, CONST RECT *lprc);`

  Выполняется побитовая инверсия цвета прямоугольника.

- `BOOL WINAPI IsRectEmpty(CONST RECT *lprc);`

  Осуществляется определение, является ли прямоугольник пустым?

- `BOOL WINAPI OffsetRect(LPRECT lprc, int dx, int dy);`

  Выполняется перемещение координат прямоугольника на `dx и dy`.

- `BOOL WINAPI PtInRect(CONST RECT *lprc, POINT pt);`

  Определение, содержится ли точка `pt` внутри прямоугольника `*lprc`.

- `BOOL WINAPI SetRect(LPRECT lprc, int xLeft, int yTop, int xRight,int yBottom);`

  Задание полей структуры прямоугольника.

- `BOOL WINAPI SetRectEmpty(LPRECT lprc);`

  Установка полей структуры прямоугольника в ноль.

- `BOOL SubtractRect(LPRECT lprcDst, CONST RECT *lprcSrc1, CONST RECT *lprcSrc2);`

  Функция определяет координаты прямоугольника, вычитая один прямоугольник из другого.

- `BOOL WINAPI UnionRect(LPRECT lprcDst, CONST RECT *lprcSrc1, CONST RECT *lprcSrc2);`

  Осуществление объединения двух прямоугольников.

Прямоугольники удобны тем, что они рисуются кистью, которую нет необходимости устанавливать в качестве текущей.

Регионы

Регион — это область экрана, представляющая собой комбинацию прямоугольников, полигонов и эллипсов. Регионы можно использовать для заливки сложных фигур, а также для установки области отсечения, т. е. области вывода.

Простейшие регионы — прямоугольный и эллиптический — создаются функциями:

```
HRGN WINAPI CreateRectRgn(int x1, int y1, int x2, int y2);
HRGN WINAPI CreateRectRgnIndirect(CONST RECT *lprect);
HRGN WINAPI CreateEllipticRgn(int x1, int y1, int x2, int y2);
HRGN WINAPI CreateEllipticRgnIndirect(CONST RECT *lprect);
```

Функции возвращают дескриптор созданного региона.

По завершении работы регион должен быть удален функцией `DeleteObject()`.

Из множества функций, работающих с регионами, рассмотрим лишь несколько:

- `int WINAPI CombineRgn(HRGN hrgnDst,HRGN hrgnSrc1,HRGN hrgnSrc2, int iMode);`

  Функция объединяет два региона `hrgnSrc1` и `hrgnSrc2`, результат помещает в `hrgnDst`. Все три региона должны быть действительны.

  Параметр `iMode` определяет, как объединяются 2 региона:

iMode	**Новый регион**
RGN_AND	Область пересечения двух исходных регионов
RGN_OR	Объединение двух исходных регионов
RGN_XOR	Объединение двух исходных регионов за исключением области пересечения
RGN_DIFF	Часть региона `hrgnSrc1`, не входящая в регион `hrgnSrc2`
RGN_COPY	Копия региона `hrgnSrc1`

- `BOOL WINAPI FillRgn(HDC hdc, HRGN hrgn, HBRUSH hbr);`

  Функция закрашивает область `hrgn` кистью `hbr`.

- `BOOL WINAPI PaintRgn(HDC hdc, HRGN hrgn);`

  Функция закрашивает область `hrgn` текущей кистью.

- `int  WINAPI OffsetRgn(HRGN hrgn, int dx, int dy);`

  Функция смещает регион `hrgn` на `dx` и `dy`.

- `BOOL WINAPI PtInRegion(HRGN hrgn, int x, int y);`

  Функция определяет, входит ли точка (`x, y`) в регион `hrgn`?

- `BOOL RectInRegion(HRGN hrgn, CONST RECT *lprc);`

  Функция определяет, лежит ли какая-либо часть указанного прямоугольника в пределах границ региона.

- `int WINAPI SetPolyFillMode(HDC hdc, int mode);`

  Функция устанавливает режим закрашивания перекрывающихся областей. При значении параметра `mode` — `ALTERNATE` перекрывающиеся области не закрашиваются, если же значение параметра `WINDING` — будет закрашена вся фигура.

Рассмотрим пример графического построения с использованием регионов (листинг 1.11). Фигуру, изображенную на рис. 1.17, без использования регионов построить было бы сложно.

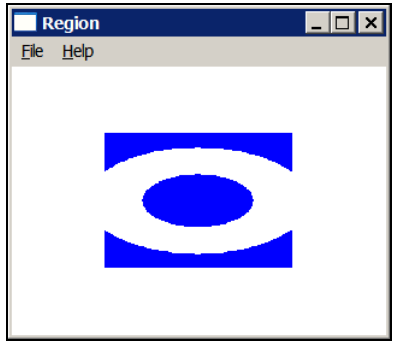


Рис. 1.17. Простая графика с использованием регионов

Листинг 1.11. Использование регионов для графических построений

```
RECT pRect = {-100, -100, 100, 100};
RECT pEllips = {-120, -80, 120, 80};
RECT pSm = {-60, -40, 60, 40};
const int WIDTH = 400;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        PAINTSTRUCT ps;
```

```
        HDC hdc;
        static int sx, sy;
        HRGN hRgnEllipse;
        HRGN hRgn;
        static HBRUSH hBrush;
        switch (message)
        {
        case WM_SIZE:
               sx = LOWORD(lParam);
               sy = HIWORD(lParam);
               break;
        case WM_CREATE:
               hBrush = CreateSolidBrush(RGB(0, 0, 255));
               break;
        case WM_PAINT:
               hdc = BeginPaint(hWnd, &ps);
               SetMapMode(hdc, MM_ANISOTROPIC);
               SetWindowExtEx(hdc, WIDTH, - WIDTH, NULL);
               SetViewportExtEx(hdc, sx, sy, NULL);
               SetViewportOrgEx(hdc, sx/2, sy/2, NULL);
               hRgn = CreateRectRgnIndirect(&pRect);
               hRgnEllipse = CreateEllipticRgnIndirect(&pEllips);
               CombineRgn(hRgn, hRgn, hRgnEllipse, RGN_DIFF);
               DeleteObject(hRgnEllipse);
               hRgnEllipse = CreateEllipticRgnIndirect(&pSm);
               CombineRgn(hRgn, hRgn, hRgnEllipse, RGN_OR);
               DeleteObject(hRgnEllipse);
               FillRgn(hdc, hRgn, hBrush);
               DeleteObject(hRgn);
               EndPaint(hWnd, &ps);
               break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

Опишем в программе переменные типа `RECT` для задания координат прямоугольника и двух эллипсов. В сообщении `WM_PAINT` установим логическую систему координат размером 400×400 с началом в центре окна. Строим прямоугольный регион `hRgn` и эллиптический `hRgnEllipse`, после чего объединяем их:

```
CombineRgn(hRgn, hRgn, hRgnEllipse, RGN_DIFF);
```

Режим `RGN_DIFF` обеспечит "вычитание" области эллиптического региона из прямоугольного. Теперь строим еще один эллиптический регион и "складываем" его с предыдущим регионом.

Отображаем полученную фигуру, закрашивая ее синей кистью:

```
FillRgn(hdc, hRgn, hBrush);
```

Не забываем, для экономии памяти, удалить регионы, как только надобность в них отпадает.

Пути

Путь — это набор прямых линий и кривых. Его можно применять для графических построений, а также конвертировать в регион и использовать для отсечения.

Рассмотрим пример использования путей для графики (листинг 1.12), вот некоторые из необходимых функций:

- `HDC WINAPI BeginPath(HDC hdc);`

  Открывает путь, теперь графические функции в окно ничего не выводят, а строят путь.

- `HDC WINAPI CloseFigure(HDC hdc);`

  Закрывает открытую фигуру в пути. Замыкает первую и последнюю точки.

- `HDC WINAPI EndPath(HDC hdc);`

  Закрывает путь и помещает его в контекст устройства.

- `HDC WINAPI FillPath(HDC hdc);`

  Закрашивает текущей кистью область, ограниченную путем.

- `HRGN WINAPI PathToRegion(HDC hdc);`

  Преобразует путь в область, возвращает ее дескриптор.

- `HDC WINAPI StrokePath(HDC hdc);`

  Обводит путь текущим пером.

Построим закрашенную красным цветом звезду (рис. 1.18), обводя ее контуры по 5 вершинам.

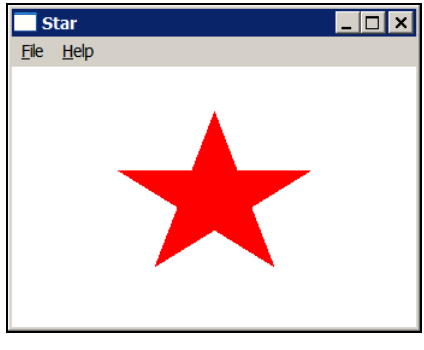


Рис. 1.18. Звезда

Массив вершин опишем на глобальном уровне, указывая координаты через одну, чтобы соблюдать порядок ее построения. Кисть красного цвета создадим при открытии окна. Все построения осуществим в сообщении `WM_PAINT`, где для простоты перейдем в локальную систему координат с началом в центре окна, размеры окна зададим 400×300.

Откроем путь функцией `BeginPath()`. После этого графические функции не выводят в окно, а строят путь. В нашем случае обведем контур функцией `Polyline()`. Функция `CloseFigure()` замыкает фигуру, `EndPath()` — закрывает путь.

Выбираем красную кисть в качестве текущей и закрашиваем область, ограниченную путем, функцией `FillPath()`. Однако если так сделать, центральная часть фигуры закрашена не будет, поскольку по-умолчанию устанавливается `ALTERNATE` режим закраски. Можно изменить режим, вызвав предварительно функцию:

```
SetPolyFillMode(hdc, WINDING);
```

Теперь фигура построится правильно, см. листинг 1.12 и рис. 1.18.

Листинг 1.12. Графические построения с использованием путей

```
POINT pt[5] = { {0,100}, {-59,-81}, {95,31}, {-95,31}, {59,-81} };
const int WIDTH = 400;
const int HEIGHT  = 300;
 LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        static int sx, sy;
        static HBRUSH hBrush;
        HRGN hRgn;
        switch (message)
        {
        case WM_CREATE:
                hBrush = CreateSolidBrush(RGB(255, 0, 0));
                break;
        case WM_SIZE:
                sx = LOWORD(lParam);
                sy = HIWORD(lParam);
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                SetMapMode(hdc, MM_ANISOTROPIC);
                SetWindowExtEx(hdc, WIDTH, -HEIGHT , NULL);
```

```
                SetViewportExtEx(hdc, sx, sy, NULL);
                SetViewportOrgEx(hdc, sx/2, sy/2, NULL);
                BeginPath(hdc);
                Polyline(hdc, pt, 5);
                CloseFigure(hdc);
                EndPath(hdc);
                SelectObject(hdc, hBrush);
                SetPolyFillMode(hdc, WINDING);
                FillPath(hdc);
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

Области отсечения

Мы уже имели дело с "недействительным прямоугольником", который представляет собой простейший случай "региона отсечения", поскольку обновляется только область, ограниченная указанным прямоугольником, весь остальной вывод игнорируется. Для управления недействительным прямоугольником имеется 3 функции:

```
BOOL WINAPI InvalidateRect(HWND hWnd, CONST RECT *lpRect, BOOL bErase);
BOOL WINAPI ValidateRect(HWND hWnd, CONST RECT *lpRect);
BOOL WINAPI GetUpdateRect(HWND hWnd, LPRECT lpRect, BOOL bErase);
```

Первая из них объявляет "недействительный прямоугольник", вторая позволит объявить прямоугольник "действительным", т. е. исключить его из региона отсечения. Третья функция позволит получить координаты "недействительного прямоугольника".

Если же область вывода имеет более сложный вид, можно создать "недействительный регион", для чего имеется набор функций:

```
BOOL WINAPI InvalidateRgn(HWND hWnd, HRGN hRgn, BOOL bErase);
BOOL WINAPI ValidateRgn(HWND hWnd, HRGN hRgn);
```

Аналогично случаю прямоугольной области, первая функция объявляет "недействительный регион", а вторая, наоборот, объявляет регион "действительным". Для использования региона отсечения его нужно выбрать для активного контекста функцией `SelectObject()` или `SelectClipRgn()`:

```
int  WINAPI SelectClipRgn(HDC hdc, HRGN hrgn);
```

Рассмотрим только две из имеющегося набора функций для манипуляций регионом отсечения:

- `BOOL WINAPI SelectClipPath(HDC hdc, int mode);`

  Выбирает текущий путь как область отсечения, объединяя его с текущей областью отсечения в зависимости от значения параметра:

mode	Новая область отсечения
RGN_AND	Пересечение текущей области и текущего пути
RGN_OR	Объединение текущей области и текущего пути
RGN_XOR	Объединение текущей области и текущего пути за исключением области пересечения
RGN_DIFF	Текущая область за исключением текущего пути
RGN_COPY	Копия текущего пути

- `int WINAPI SelectClipRgn(HDC hdc, HRGN hrgn);`

  Выбирает область отсечения для данного контекста.

Вывод текста

До сих пор мы выводили текст, используя стандартный шрифт по умолчанию. Сейчас посмотрим, как можно установить другой шрифт для вывода текста, а также использовать различные возможности его оформления.

Цвет текста и фона

При выводе текста функцией `TextOut()` имеется возможность установить цвет фона и текста при помощи пары функций: `SetBkColor()` и `SetTextColor()`:

```
COLORREF WINAPI SetBkColor(HDC hdc, COLORREF color);
COLORREF WINAPI SetTextColor(HDC hdc, COLORREF color);
```

Обе функции принимают дескриптор контекста устройства `hdc` и цвет `color`, и возвращают предыдущий цвет или `CLR_INVALID` в случае ошибки.

> ***Примечание***
>
> Функция `SetBkColor()` определяет также фон между линиями для штриховой кисти.

Имеется еще одна функция `SetBkMode()`, которая управляет режимом отображения фона при выводе текста:

```
int WINAPI SetBkMode(HDC hdc, int mode);
```

Эта функция также получает дескриптор контекста устройства `hdc` и `mode` — режим отображения, принимающий одно из двух значений:

- `OPAQUE` — цвет фона при выводе текста будет определяться цветом, заданным функцией `SetBkColor()`, установлено по умолчанию;
- `TRANSPARENT` — цвет фона при выводе текста не изменится.

Например, так можно установить вывод синего текста на желтом фоне:

```
SetBkColor(hdc,RGB(255,255,0));
SetTextColor(hdc,RGB(0,0,128));
```

Получение метрики текста

При помощи функции `GetTextMetrics()` можно получить информацию о текущем шрифте данного контекста. Функция заполняет структуру `TEXTMETRIC` информацией о текущем шрифте.

```
BOOL GetTextMetricsA(HDC hdc, TEXTMETRIC* tm);
```

Структура имеет множество полей, содержащих характеристики шрифта, но в настоящее время нас интересуют лишь те, которые определяют размер шрифта (рис. 1.19).

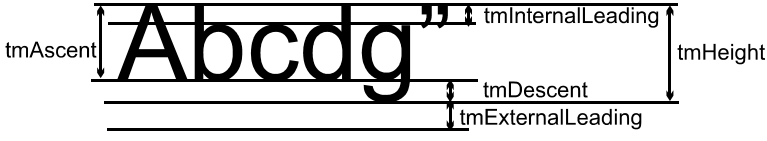


Рис. 1.19. Основные характеристики шрифта

Построим тестовую программу для демонстрации текстового вывода шрифтом по умолчанию (листинг 1.13). Покажем различное фоновое заполнение, установку цвета текста и основные характеристики шрифта.

Листинг 1.13. Вывод числовых характеристик шрифта, заданного по умолчанию

```
TCHAR *text = _T("Текст для вывода в окне");
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        PAINTSTRUCT ps;
        HDC hdc;
        TEXTMETRIC tm;
        TCHAR str[256];
        RECT rt;
        SIZE size;
        switch (message)
        {
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                SetBkColor(hdc, RGB(255, 255, 0));  // Желтый фон
                SetTextColor(hdc, RGB(0, 0, 128));  // Синий шрифт
                TextOut(hdc, 0, 0, text, _tcsclen(text));
                SetBkMode(hdc, TRANSPARENT);        // Прозрачный фон
                SelectObject(hdc, GetStockObject(ANSI_VAR_FONT));
```

```
                GetTextMetrics(hdc, &tm);
                _stprintf(str, _T("tmHeight = %d\ntmInternalLeading =\
                        %d\ntmExternalLeading = %d\ntmAscent = %d\ntmDescent =
%d\n"),
                        tm.tmHeight, tm.tmInternalLeading, tm.tmExternalLeading,
                        tm.tmAscent, tm.tmDescent);
                SetRect(&rt, 0, 20, 150, 100);
                DrawText(hdc, str, _tcslen(str), &rt, DT_LEFT);
                GetTextExtentPoint32(hdc, text, _tcsclen(text), &size);
                _stprintf(str, _T("Ширина строки = %d\nВысота строки = %d"),
                        size.cx, size.cy);
                SetRect(&rt, 0, 100, 150, 150);
                DrawText(hdc, str, _tcslen(str), &rt, DT_LEFT);
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

Для работы со строками проще всего использовать функцию форматного вывода в строку `_stprintf()`. Получив параметры шрифта функцией `GetTextMetrics()`, сформируем строку для вывода, "склеивая" комментарий с числовым значением. Вывод осуществим при помощи функции `DrawText()`:

```
int WINAPI DrawTextW(HDC hdc, LPCWSTR lpchText, int cchText,
LPRECT lprc, UINT format);
```

которая обеспечит многострочный вывод строки `str` в заданный прямоугольник `rt` в соответствии с заданным форматом `DT_LEFT` и правильно обработает символы перевода строки `'\n'`.

Обратите внимание, что первая строка отображается с желтым фоном, поскольку по умолчанию работает параметр `OPAQUE` функции `SetBkMode()`. Однако после установки `SetBkMode(hdc, TRANSPARENT)` остальной вывод будет на белом фоне.

Отметим, в заключение, что высота шрифта определяется суммой высоты букв и межстрочным расстоянием:

```
Height = tmHeight + tmExternalLeading;
```

Определение длины строки

Иногда при выводе текста необходимо определить длину строки в логических единицах. Это бывает нужно или для определения позиции последующего вывода, или для организации переноса, когда строка текста не входит в одну экранную строку. Задача не столь проста, как может показаться на первый взгляд, поскольку, если шрифт не является моноширинным, то символы имеют различную ширину, что

имеет место для шрифтов TrueType. К счастью, имеется специальная функция, GetTextExtentPoint32(), которая выполнит эту работу:

```
BOOL APIENTRY GetTextExtentPoint32W(HDC hdc, LPCWSTR lpString, int c, LPSIZE
psizl);
```

hdc — дескриптор контекста устройства, lpString — выводимая строка, c — длина этой строки и psizl — указатель на структуру SIZE:

```
struct SIZE
{
   LONG cx;    //Ширина
   LONG cy;    //Высота
};
```

Если мы допишем в предыдущую программу пару строк:

```
SIZE size;
GetTextExtentPoint32(hdc, text, _tcsclen(text), &size);
```

то из переменных size.cx и size.cy можно извлечь и вывести в окно значения ширины и высоты строки. Проделайте это самостоятельно.

Системные шрифты

Операционная система имеет набор встроенных шрифтов, которые может использовать прикладная программа, выбрав их при помощи функции GetStockObject(). В табл. 1.4 приведены символические имена, определяющие эти шрифты.

Примечание

Шрифт должен быть выбран в качестве текущего для данного контекста.

Таблица 1.4. *Идентификаторы стандартных шрифтов*

Идентификатор	Шрифт
ANSI_FIXED_FONT	Шрифт с фиксированным размером символов
ANSI_VAR_FONT	Шрифт с переменной шириной символов
DEVICE_DEFAULT_FONT	Шрифт по умолчанию
DEFAULT_GUI_FONT	Шрифт графического интерфейса по умолчанию
OEM_FIXED_FONT	OEM-шрифт (кириллицы нет)
SYSTEM_FONT	Системный шрифт
SYSTEM_FIXED_FONT	Системный шрифт (устаревший)

Для тестирования стандартных шрифтов добавим, например, такую строку в сообщение WM_PAINT:

```
SelectObject(hdc, GetStockObject(ANSI_VAR_FONT));
```

Вывод текста теперь будет осуществляться другим шрифтом.

Освобождать ресурсы, занятые встроенными шрифтами, не нужно.

Определение произвольных шрифтов

Кроме встроенных шрифтов, в программе может использоваться любой шрифт, зарегистрированный в системе. Для того чтобы программа могла осуществлять вывод выбранным шрифтом, его необходимо создать функцией `CreateFont()` или `CreateFontIndirect()`:

```
HFONT WINAPI CreateFontW(int Height, int Width, int Escapement,
             int Orientation, int Weight, DWORD Ital,
             DWORD Underline, DWORD StrikeThru, DWORD Charset,
             DWORD Precision, DWORD ClipPrecision,
             DWORD Quality, DWORD Pitch, LPCWSTR FontName);
HFONT WINAPI CreateFontIndirectW(CONST LOGFONTW *lplf);
```

Структура `LOGFONTW` имеет поля, соответствующие параметрам функции `CreateFontW()`:

- `Height` — высота шрифта;
- `Width` — ширина шрифта;
- `Escapement` — угол наклона строки текста относительно горизонтальной оси в десятых долях градуса;
- `Orientation` — угол наклона каждого символа относительно горизонтальной оси в десятых долях градуса (только для расширенного графического режима);
- `Weight` — насыщенность (жирность) текста, лежит в диапазоне [0; 1000]. (400 — нормальный текст, 700 — жирный.) `FW_NORMAL` или 0 задает нормальную жирность;
- `Ital` — ненулевое значение, создает наклонный шрифт;
- `Underline` — ненулевое значение, создает подчеркнутый шрифт;
- `StrikeThru` — ненулевое значение, создает перечеркнутый шрифт;
- `Charset` — определяет множество символов шрифта, обычно задается макросом, например, `DEFAULT_CHARSET`;
- `Precision` — точность отображения шрифта, например, `OUT_DEFAULT_PRECIS`;
- `ClipPrecision` — определяет, как будут отсекаться символы, не попадающие в видимую область вывода, например, `CLIP_DEFAULT_PRECIS`;
- `Quality` — качество шрифта: `DEFAULT_QUALITY`, `DRAFT_QUALITY`, `PROOF_QUALITY`;
- `Pitch` — тип и семейство шрифтов. Значение формируется операцией логического сложения для выбранного типа и семейства.

  Тип шрифта: `DEFAULT_PITCH, FIXED_PITCH, VARIABLE_PITCH.`

  Семейство шрифта: `FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS.`

  Тип и семейство используются системой для подбора наиболее подходящего шрифта, если не будет найден шрифт с указанным именем;
- `FontName` — указатель на строку, содержащую имя шрифта (до 32 символов).

Примечание

Функция `CreateFont()` не создает шрифт, а лишь настраивает зарегистрированный в системе шрифт в соответствии с заданными параметрами.

Приведем небольшую программу (листинг 1.14) для демонстрации использования произвольных шрифтов (рис. 1.20). Используем возможность изменения ориентации выводимой строки — параметр `Orientation`. Изменим также насыщенность шрифта `Weight` и наклон `Ital`.

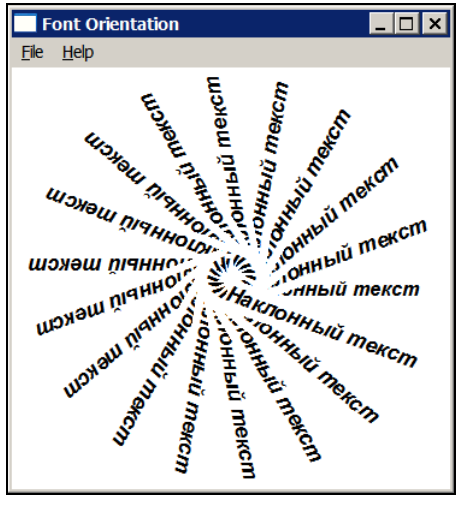


Рис. 1.20. Вывод текста по наклонным линиям из центра окна

Листинг 1.14. Вывод текста по радиальным линиям

```
TCHAR str[] = _T("Наклонный текст");
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        HFONT oldFont, newFont;
        int Escapement;
        static int sx, sy;
        switch (message)
        {
        case WM_SIZE:
                sx = LOWORD(lParam)/2;
                sy = HIWORD(lParam)/2;
                break;
```

```
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                for (Escapement = 0; Escapement < 3600; Escapement += 200)
                        {
                newFont = CreateFont(20, 0, Escapement, 0, 700, 1, 0, 0,
                        DEFAULT_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY,      DEFAULT_PITCH | FF_DONTCARE,_T("Arial"));
                                oldFont = (HFONT)SelectObject(hdc, newFont);
                                TextOut(hdc, sx, sy, str,_tcsclen(str));
                                SelectObject(hdc, oldFont);
                                DeleteObject(newFont);
                        }
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default:        return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

В обработчике сообщения `WM_PAINT` будем создавать шрифт в цикле, меняя лишь угол наклона через 20°. Устанавливаем созданный шрифт текущим и выводим строку текста. Теперь необходимо удалить шрифт, чтобы освободить память, однако текущий шрифт удалять нельзя, поэтому мы делаем текущим предыдущий шрифт:

```
SelectObject(hdc, oldFont);
```

именно для этого мы и сохранили его дескриптор.

Затем шрифт удаляем:

```
DeleteObject(newFont);
```

> ***Примечание***
>
> Если не удалять шрифты в подобных ситуациях, можно сильно "замусорить" память, поскольку автоматически шрифты будут удалены лишь при завершении приложения.

Часто поступают иначе и создают шрифты при создании окна (сообщение `WM_CREATE`), а уничтожают при его закрытии (сообщение `WM_DESTROY`).

Диалог с пользователем

Вся идеология построения Windows-приложения ориентирована на взаимодействие с пользователем. Это может быть выбор пункта меню, специальные окна для ввода или выбора данных и т. п. Наиболее простым элементом интерфейса является *окно сообщений*.

Окно сообщений

Часто возникает ситуация, когда программа должна получить разрешение на выполнение действий. Для этих целей используется функция `MessageBox()`, которая отображает сообщение и ждет реакции пользователя. Никакого дополнительного кода писать не нужно, функция сама создаст и отобразит окно, а после ответа пользователя — уничтожит.

```
int WINAPI MessageBoxW(HWND hWnd, LPCWSTR lpText, LPCWSTR lpCaption,
UINT uType);
```

где параметры:

- `hWnd` — дескриптор родительского окна;
- `lpText` — указатель строки сообщения;
- `lpCaption` — указатель строки заголовка окна сообщения;
- `uType` — определяет свойства окна сообщения и строится как логическая сумма этих свойств. Вот некоторые типичные его значения:
  - `MB_OK` — кнопка `OK`
  - `MB_OKCANCEL` — кнопки `OK` и `CANCEL`
  - `MB_ABORTRETRYIGNORE` — кнопки `ABORT`, `RETRY` и `IGNORE`
  - `MB_YESNOCANCEL` — кнопки `YES`, `NO` и `CANCEL`
  - `MB_YESNO` — кнопки `YES` и `NO`
  - `MB_RETRYCANCEL` — кнопки `RETRY` и `CANCEL`
  - `MB_ICONHAND` — иконка ⊗
  - `MB_ICONQUESTION` — иконка ?
  - `MB_ICONEXCLAMATION` — иконка !
  - `MB_ICONASTERISK` — иконка *

Так, если в предыдущем примере мы хотели бы перед выводом текста спросить пользователя, хочет ли он выводить текст курсивом, то могли бы это реализовать при открытии окна следующим образом:

```
case WM_CREATE:
       i = MessageBox(hWnd, _T("Будем выводить текст курсивом"),
       _T("Оформление текста"), MB_YESNO | MB_ICONQUESTION);
       k = (i == IDYES)? 1 : 0;
       break;
```

а шестым параметром `Ital` функции `CreateFont()` вместо `1` укажем переменную `k`.

Переменные опишем так:

```
static int k;
int i;
```

В результате мы увидим диалоговое окно, изображенное на рис. 1.21.

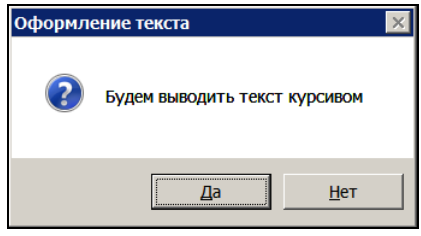


Рис. 1.21. Диалоговое окно `MessageBox`

При нажатии кнопки **Да** функция возвратит значение `IDYES`, и в этом случае мы присваиваем переменной `k` значение 1, иначе — 0. Таким образом, в зависимости от ответа пользователя текст будет выводиться либо курсивом, либо "нормальным" шрифтом.

Меню

Ни одна Windows-программа не обходится без *меню*. При выполнении команд меню происходит открытие и закрытие файлов, выбор и настройка устройства печати, настройки параметров программы и т. п. Во всех Windows-программах при создании меню необходимо придерживаться некоторых устоявшихся традиций. Так, меню верхнего уровня всегда отображается под заголовком окна, подменю отображаются как выпадающие меню.

Меню является одним из ресурсов приложения, для редактирования которого используется редактор ресурсов. Все ресурсы хранятся в текстовом файле определенной структуры с расширением `rc`. Файл ресурсов компилируется специальным компилятором ресурсов.

Файл ресурсов можно редактировать и обычным текстовым редактором типа Блокнота, однако делать это не рекомендуется, поскольку компилятор ресурсов не имеет встроенных средств отладки и при появлении малейшей ошибки найти ее будет довольно сложно.

При создании стандартной заготовки Win32-приложения автоматически создается меню с минимальным набором команд. Если в окне **Solution Explorer** перейти на подложку **Resource View**, мы можем открыть меню приложения (рис. 1.22).

> ***Примечание***
>
> Если подложка **Resource View** отсутствует в **Solution Explorer**, достаточно щелкнуть мышкой по имени файла ресурса (расширение имени `rc`) в папке **Resource Files**.

В окне **Properties** устанавливаются свойства пункта меню. Задается имя пункта меню и его идентификатор, используемый в оконной функции при обработке пункта меню. Новый пункт меню вводится в поле **Type Here**, если же нужно изменить порядок следования пунктов меню, используется "буксировка". Амперсанд "&" в имени пункта меню приводит к подчеркиванию следующего символа — так обычно выделяют "горячие клавиши", работающие с регистровой клавишей <Alt>.

Можно определить произвольный набор "горячих клавиш", для этого откроем таблицу акселераторов и введем "горячую клавишу" <Alt>+<X> для пункта меню **Exit** (рис. 1.23).

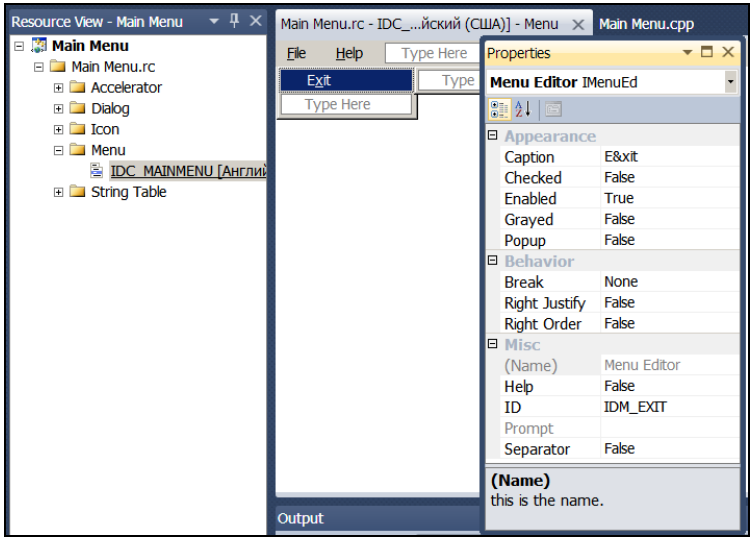


Рис. 1.22. Стандартное меню Win32-приложения

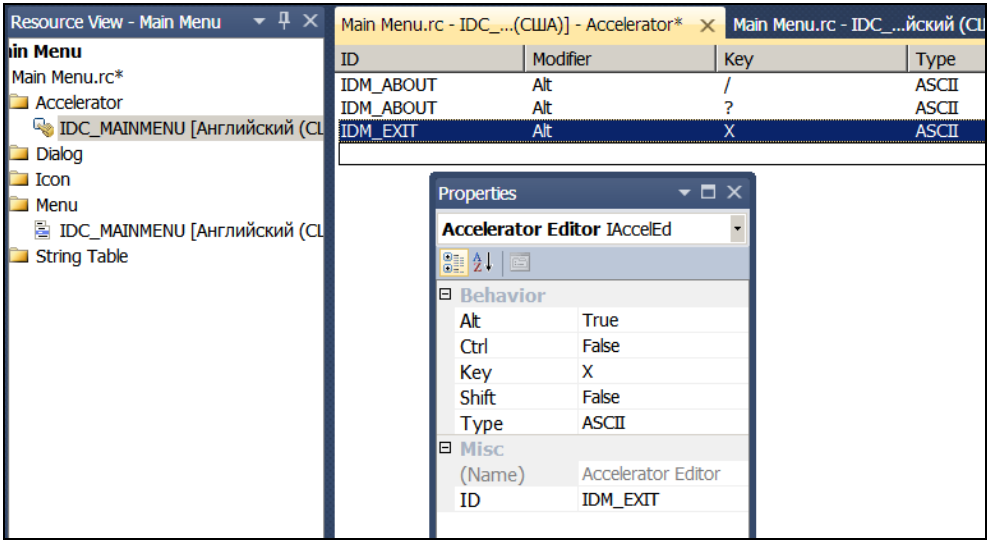


Рис. 1.23. Таблица акселераторов

Заполнение таблицы интуитивно понятно: в первой колонке **ID** выбираем идентификатор пункта меню, выбираем клавишу-модификатор, символ **X** и тип **ASCII**, что ограничивает выбор алфавитно-цифровой клавиатурой (для использования

функциональных клавиш необходимо выбрать тип **VIRTKEY**). Можно выбрать "горячую клавишу" и в окне свойств.

Выбор пункта меню, как и клавиша-акселератор, приводит к генерации сообщения `WM_COMMAND`, при этом в младшем слове `wParam` возвращается идентификатор выбранного пункта меню. Обычно обработчик сообщения `WM_COMMAND` строят на переключателе `switch()`, где и проводят обработку команд выбранного пункта меню (см. листинг 1.2).

Пример интерактивной графики

Подводя итоги рассмотрения графических примитивов, приведем более сложный пример — построение кривой Безье с возможностью ее модификации. Кривые Безье — это сплайн-функции, широко используемые в графических построениях, в частности, шрифты TrueType построены кривыми Безье.

Поставим задачу — создать приложение, которое читает массив точек из текстового файла и строит кривую Безье по этим точкам (рис. 1.24). Подготовим в текущей папке файл `dat.txt` с координатами 10 точек. "Захватив" точку нажатием левой кнопкой мыши, ее можно переместить, кривая при этом должна перестраиваться. При закрытии приложения "новый" набор точек сохраняется в том же файле.

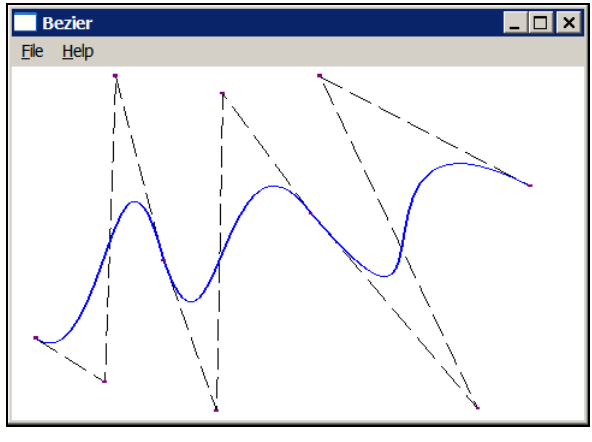


Рис. 1.24. Кривая Безье

В листинге 1.15 приведен текст оконной функции, где введены вспомогательные функции `DcInLp()` и `transform()`.

Листинг 1.15. Построение кривой Безье

```
#include <fstream>
static int sx, sy;
const int SCALE = 1000;
```

```
const int MARK = 4;
void DcInLp(POINT &point)
{
	point.x = point.x* SCALE/sx;
	point.y = SCALE - point.y* SCALE/sy;
}
void transform(HDC& hdc)
{
	SetMapMode(hdc, MM_ANISOTROPIC);
	SetWindowExtEx(hdc, SCALE, -SCALE, NULL);
	SetViewportExtEx(hdc, sx, sy, NULL);
	SetViewportOrgEx(hdc, 0, sy, NULL);
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
	PAINTSTRUCT ps;
	HDC hdc;
	static HPEN hDash, hBezier;
	static HBRUSH hRect, hSel;
	static POINT pt[20];
	static POINT point;
	RECT rt;
	static int count, index;
	static bool capture;
	int i;
	std::ifstream in;
	std::ofstream out;
	switch (message)
	{
	case WM_CREATE:
		in.open("dat.txt");
		if (in.fail())
		{
			MessageBox(hWnd,_T("Файл dat.txt не найден"),
				_T("Открытие файла"), MB_OK | MB_ICONEXCLAMATION);
			PostQuitMessage(0);
			return 1;
		}
		for (count = 0; in >> pt[count].x; count++) in >> pt[count].y;
		in.close();   //В переменной count сохранится размер массива
точек
		hDash = CreatePen(PS_DASH, 1, 0);
```

```
            hBezier = CreatePen(PS_SOLID, 4, RGB(0, 0, 255));
            hRect = CreateSolidBrush(RGB(128, 0, 128));
            hSel = CreateSolidBrush(RGB(255, 0, 0));
            break;
      case WM_SIZE:
            sx = LOWORD(lParam);
            sy = HIWORD(lParam);
            break;
      case WM_LBUTTONDOWN:
            point.x = LOWORD(lParam);
            point.y = HIWORD(lParam);
            //Преобразование экранных координат мыши в логические
            DcInLp(point);
            for (i = 0; i < count; i++)
            {
                  SetRect(&rt,pt[i].x-MARK,pt[i].y-
MARK,pt[i].x+MARK,pt[i].y+MARK);
                  if (PtInRect(&rt, point))
                  {     //Курсор мыши попал в точку
                        index = i;
                        capture = true;
                        hdc = GetDC(hWnd);
                        transform(hdc);       //Переход в логические
                                                координаты
                        FillRect(hdc, &rt, hSel);//Отметим прямоугольник
                                                   цветом
                        ReleaseDC(hWnd, hdc);
                        SetCapture(hWnd);            //Захват мыши
                        return 0;
                  }
            }
            break;
      case WM_LBUTTONUP:
            if (capture)
            {
                  ReleaseCapture();           //Освобождение мыши
                  capture = false;
            }
            break;
      case WM_MOUSEMOVE:
            if (capture)
            {     //Мышь захвачена
                  point.x = LOWORD(lParam);
                  point.y = HIWORD(lParam);
```

```
                DcInLp(point);//Преобразование экранных координат мыши
                pt[index] = point;    //в логические координаты
                InvalidateRect(hWnd, NULL, TRUE);
            }
            break;
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            transform(hdc);                    //Переход в логические
                                                 координаты
            SelectObject(hdc, hDash);
            Polyline(hdc, pt, count);    //Строим ломаную линию
            SelectObject(hdc, hBezier);
            PolyBezier(hdc, pt, count);  //Строим кривую Безье
            for (i = 0; i < count; i++)
            {       //Закрашиваем точки графика прямоугольниками
                    SetRect(&rt,pt[i].x-MARK,pt[i].y-MARK,pt[i].
                    x+MARK,pt[i].y+MARK);
                FillRect(hdc, &rt, hRect);
            }
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            DeleteObject(hDash);
            DeleteObject(hBezier);
            DeleteObject(hRect);
            DeleteObject(hSel);
            out.open("dat.txt");
            for (i = 0;i<count;i++) out << pt[i].x << '\t' << pt[i].y << '\n';
            out.close();
            PostQuitMessage(0);
            break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

С файлом будем работать при помощи потоковых классов ввода-вывода, нам понадобится файл включений `fstream`. Поскольку для примера не нужно большого количества точек, опишем массив типа `POINT` фиксированного размера. При создании окна в сообщении `WM_CREATE` прочитаем этот массив методами потокового класса `ifstream`, проверяя, удалось ли открыть файл. Если попытка завершилась неудачно, выводим диалоговое окно `MessageBox` и завершаем работу.

При чтении файла, который содержит (*x*, *y*)-координаты точки из диапазона [0; 1000], подсчитываем количество точек `count`. Затем создаем два пера и две кисти.

В сообщении `WM_SIZE` определяем размеры клиентской области окна `sx`, `sy`.

Далее в сообщении `WM_PAINT` устанавливаем локальную систему координат размером 1000×1000 с центром в левом нижнем углу, ось *y* направлена вверх. Код выделен в отдельную функцию `transform()`.

Рисуем пером `hDash` многоугольник с помощью функции `Polyline()`:

```
BOOL WINAPI Polyline(HDC hdc, CONST POINT *apt, int cpt);
```

Затем пером `hBezier` строим кривую Безье по этим же точкам функцией `PolyBezier()`:

```
WINAPI PolyBezier(HDC hdc, CONST POINT * apt, DWORD cpt);
```

Размер массива для построения кривой Безье должен быть кратным 3 плюс 1 точка.

Затем вокруг каждой точки закрашиваем кистью `hRect` прямоугольник размером 8×8. Это мы сделаем в цикле, используя функции прямоугольника `SetRect()` и `FillRect()`.

Нажатием на левую кнопку мыши определяем координаты курсора мыши из младшего и старшего слова `lParam` и помещаем в переменную типа `POINT`. Чтобы иметь возможность сравнивать координаты, необходимо преобразовать аппаратные координаты мыши в логические, поскольку в сообщениях от мыши координаты всегда возвращаются в исходную систему координат с началом в левом верхнем углу, ось *x* направлена вправо, ось *y* — вниз. Для этих целей мы подготовили функцию `DcInLp()`. Теперь в цикле проверяем для каждой точки, не указывает ли курсор мыши на точку графика? Критерий — попадание в прямоугольник вокруг точки размером 8×8. Мы реализуем такую проверку, обращаясь к функции `PtInRect()`, но предварительно создадим для наглядности вокруг каждой точки локальный прямоугольник `rt`. Если в точку попали, сохраняем ее индекс, устанавливаем в `true` логическую переменную и перекрашиваем прямоугольник кистью `hSel`. Разумеется, нам пришлось вновь перейти к логическим координатам с помощью локальной функции `transform()`. После чего "захватываем" мышь функцией `SetCapture()`:

```
HWND WINAPI SetCapture(HWND hWnd);
```

Этот прием требует некоторых пояснений. Дело в том, что сообщения мыши поступают в окно, пока ее курсор находится внутри окна. Как только курсор покидает пределы окна, сообщения мыши окну уже не поступают, поэтому, если при нажатой кнопке мыши курсор выйдет за пределы окна, то сообщение об отпускании кнопки в окно уже не поступит и будет утеряно.

Одним из способов решения данной проблемы является "захват" мыши: после чего все сообщения мыши будут поступать в окно независимо от того, где находится ее курсор. Нужно только соблюдать "технику безопасности" и вовремя освобождать мышь. Общепринятая практика — захватить мышь при нажатии кнопки и освободить при ее отпускании. Так мы и будем поступать.

Во время перемещения мыши в сообщении `WM_MOUSEMOVE` проверим значение логической переменной `capture` и, если она равна `true`, преобразуем аппаратные координаты мыши в логические локальной функцией `DcInLp()`, и возвратим их в массив точек. После чего объявляем окно недействительным для его перерисовки.

И, наконец, в сообщении об отпускании мыши `WM_LBUTTONUP` "освобождаем" мышь функцией `ReleaseCapture()`:

```
BOOL WINAPI ReleaseCapture(VOID);
```

и устанавливаем логическую переменную `capture` в `false`.

Нам осталось лишь освободить ресурсы (перья и кисти) перед закрытием окна в сообщении `WM_DESTROY` и сохранить измененный массив точек в тот же текстовый файл, используя класс потокового вывода `ofstream`.

Вопросы к главе

1. Типы данных в Windows-приложениях.
2. Что такое дескриптор?
3. Что такое "Функция обратного вызова"?
4. Какие действия выполняет головная функция `WinMain()`?
5. Назначение оконной функции.
6. Как происходит обработка сообщения о нажатии клавиши на клавиатуре?
7. В чем специфика сообщения `WM_PAINT`?
8. Как извлечь текущие координаты мыши?
9. Как заставить систему перерисовать окно?
10. Как создать и уничтожить таймер?
11. Функции простейших графических примитивов.
12. Два способа определения размеров клиентской области окна.
13. Как создать и установить перо и кисть?
14. Зачем нужно уничтожать созданные перья и кисти?
15. Что такое "Чистый цвет"?
16. Как устанавливаются логические координаты?
17. Как задать новое начало координат и сменить направление осей?
18. Установка фона и цвета при выводе текста.
19. Функции вывода текста.
20. Установка системного и произвольного шрифта.
21. Как получить метрику шрифта?
22. Как вывести окно сообщения?
23. Как добавить к приложению пункт меню?
24. Как создать клавишу-акселератор?
25. Как осуществляется "захват мыши"?

Задания для самостоятельной работы

1. Написать программу, строящую прямоугольник в центре окна. При нажатии на левую кнопку мыши его размеры уменьшаются, а при нажатии на правую кнопку — увеличиваются на 10 пикселов.
2. Решить предыдущую задачу, только размеры должны изменяться автоматически через 1 секунду. Нажатие на левую кнопку мыши меняет направление изменения размеров. Правая кнопка завершает работу.
3. Написать программу движения шарика в окне с отражением от стенок по законам геометрической оптики. Начало движения происходит из точки, в которой нажимается левая кнопка мыши. Скорость постоянна, начальный угол определяется случайным образом. Размер шарика и скорость движения выберите произвольно.
4. Написать программу, которая разрисует окно, как шахматную доску и при нажатии левой кнопки мыши выведет окно сообщений с именем клетки, где находится курсор в шахматной нотации.
5. Написать программу, которая с периодичностью в 0,1 секунды заполняет окно прямоугольниками случайного размера (не превосходящего ¼ окна) случайным цветом.
6. Построить стрелочный секундомер во все окно. Запускается секундомер левой кнопкой мыши, нажатие любой клавиши останавливает отсчет, правая кнопка мыши — сброс.
7. Создать тестовую программу вывода строки текста, меняя размер шрифта от минимально читаемого размера (определите опытным путем) до 1 дюйма.
8. Написать программу, которая выводит все характеристики шрифта, возвращаемые в структуре `TEXTMETRIC`. Проверить работу со шрифтом "Times New Roman".
9. Дополните задачу о построении кривой Безье (листинг 1.15) возможностью добавлять или удалять точки графика нажатием на правую кнопку мыши (если выделена существующая точка графика, она удаляется).
10. В той же задаче о построении кривой Безье предусмотрите возможность вывода координат точек графика по нажатию левой кнопки мыши вместе с клавишей <Ctrl>.

Глава 2

Работа с файлами

Для работы с файлами в API-библиотеке имеется собственный набор функций, однако будем пока работать с классами потокового ввода-вывода, которые хорошо зарекомендовали себя в консольных приложениях. Мы также имеем возможность использовать предоставляемые операционной системой дополнительные сервисы, например, для получения имени файла будем использовать стандартный диалог.

Диалог выбора файлов

Имеется два практически идентичных диалога выбора имени файла: открыть файл для чтения `GetOpenFileName()` и для записи `GetSaveFileName()`:

```
BOOL  APIENTRY GetOpenFileNameW(LPOPENFILENAMEW);
BOOL  APIENTRY GetSaveFileNameW(LPOPENFILENAMEW);
```

Аргумент функций — указатель на переменную типа `OPENFILENAMEW`. Поля этой структуры приведены в табл. 2.1.

Все функции, обеспечивающие диалог, описаны в файле включений `commdlg.h`.

Таблица 2.1. *Описание полей структуры* `OPENFILENAMEW`

Тип поля	Имя	Описание
`DWORD`	`lStructSize`	Размер структуры в байтах
`HWND`	`hwndOwner`	Дескриптор окна-владельца
`HINSTANCE`	`hInstance`	Дескриптор приложения
`LPCWSTR`	`lpstrFilter`	Фильтр из пар строк — в кодировке Unicode
`LPWSTR`	`lpstrCustomFilter`	Пользовательский фильтр — в кодировке Unicode
`DWORD`	`nMaxCustFilter`	Размер фильтра в символах
`DWORD`	`nFilterIndex`	Смещение в буфере фильтра
`LPWSTR`	`lpstrFile`	Указатель на строку с именем файла — в кодировке Unicode
`DWORD`	`nMaxFile`	Размер буфера для имени файла
`LPWSTR`	`lpstrFileTitle`	Указатель на строку с выбранным именем файла — в кодировке Unicode

***Таблица 2.1** (окончание)*

Тип поля	Имя	Описание
DWORD	nMaxFileTitle	Размер буфера для имени выбранного файла
LPCWSTR	lpstrInitialDir	Указатель на строку начального каталога — в кодировке Unicode
LPCWSTR	lpstrTitle	Указатель на строку с заголовком окна — в кодировке Unicode
DWORD	Flags	Флаг: OFN_ALLOWMULTISELECT — допускается выбор нескольких файлов; OFN_HIDEREADONLY — переключатель Read Only не выводится; OFN_READONLY — только для чтения; OFN_OVERWRITEPROMPT — генерируется сообщение при записи существующего файла
WORD	nFileOffset	Смещение имени файла в строке полного имени
WORD	nFileExtension	Смещение расширения имени файла
LPCWSTR	lpstrDefExt	Указатель на строку с расширением имени файла по умолчанию — в кодировке Unicode
LPARAM	lCustData	Данные для функции-перехватчика
LPOFNHOOKPROC	lpfnHook	Указатель на функцию-перехватчик
LPCWSTR	lpTemplateName	Шаблон диалогового окна — в кодировке Unicode

Таким образом, для организации диалога выбора файла необходимо заполнить поля структуры OPENFILENAME и вызвать одну из функций GetOpenFileName() или GetSaveFileName(), после их успешной работы в поле lpstrFile получим указатель на строку с полным именем выбранного файла.

Простой просмотрщик файлов

Рассмотрим простую задачу чтения текстового файла, где для выбора его имени используем стандартный Windows-диалог. В листинге 2.1 приведен текст оконной функции этого приложения.

Листинг 2.1. Просмотрщик файлов с диалогом выбора имени

```
#include <commdlg.h>
#include <fstream>
#include <vector>
#include <string>
const int LineHeight = 16;//Высота строки текста + межстрочное расстояние
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   int wmId;
   PAINTSTRUCT ps;
   HDC hdc;
   static TCHAR name[256] = _T("");;
   static OPENFILENAME file;
   std::ifstream in;
   std::ofstream out;
   static std::vector<std::string> v;
   std::vector<std::string>::iterator it;
   std::string st;
   int y;
   switch (message)
   {
   case WM_CREATE:
      file.lStructSize = sizeof(OPENFILENAME);
      file.hInstance = hInst;
      file.lpstrFilter = _T("Text\0*.txt");
      file.lpstrFile = name;
      file.nMaxFile = 256;
      file.lpstrInitialDir = _T(".\\");
      file.lpstrDefExt = _T("txt");
      break;
   case WM_COMMAND:
      wmId    = LOWORD(wParam);
      switch (wmId)
      {
         case ID_FILE_NEW :
            if (!v.empty()) std::vector<std::string>().swap(v);
            InvalidateRect(hWnd, NULL, TRUE);
            break;
         case ID_FILE_OPEN :
            file.lpstrTitle = _T("Открыть файл для чтения");
            file.Flags = OFN_HIDEREADONLY;
            if (!GetOpenFileName(&file)) return 1;
            in.open(name);
            while (getline(in,st)) v.push_back(st);
            in.close();
            InvalidateRect(hWnd,NULL,1);
            break;
         case ID_FILE_SAVE :
```

```
            file.lpstrTitle = _T("Открыть файл для записи");
            file.Flags = OFN_NOTESTFILECREATE;
            if (!GetSaveFileName(&file)) return 1;
            out.open(name);
            for (it = v.begin(); it != v.end(); ++it) out << *it << '\n';
            out.close();
            break;
         case IDM_EXIT: DestroyWindow(hWnd); break;
         default: return DefWindowProc(hWnd, message, wParam, lParam);
         }
         break;
      case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);
            for (y = 0, it = v.begin(); it < v.end(); ++it, y += LineHeight)
               TextOutA(hdc, 0, y, it->data(), it->length());
         EndPaint(hWnd, &ps);
         break;
      case WM_DESTROY: PostQuitMessage(0); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

Начнем с того, что добавим к существующему меню три пункта. Для этого перейдем на подложку **Resourse View** рабочей области и в выпадающем меню **File** введем три пункта **&New**, **&Open**, **&Save** (см. рис. 1.22). Если не вводить идентификаторы пунктов меню, то мастер присвоит им значения по умолчанию: `ID_FILE_NEW`, `ID_FILE_OPEN`, `ID_FILE_SAVE`. Затем создадим сепаратор (линию-разделитель): для этого достаточно ввести вместо имени дефис (-) или отметить флаг **Separator** в окне свойств. После чего "отбуксируем" пункт меню **Exit** в конец списка.

> ***Примечание***
>
> Чтобы вновь не открывать окно свойств для каждого пункта меню, достаточно прикрепить его к "холсту" верхней кнопкой с изображением канцелярской кнопки.

При выборе пункта меню будет сгенерировано сообщение `WM_COMMAND`, а в младшее слово `wParam` помещено значение соответствующего идентификатора. Наша задача заключается в добавлении к оконной функции обработчиков этих сообщений.

Однако предварительно необходимо решить, как мы будем хранить строки прочитанного файла.

Можно в стиле классического C читать из файла построчно и выделять память для хранения каждой строки, а указатели на строки хранить в одномерном массиве. Но поскольку мы не знаем заранее размер файла, то нам придется либо запасать большой массив указателей, на всякий случай (что, тем не менее, приведет к огра-

ничению размера файла), либо предварительно определить его размер, а затем выделить необходимую память. Первый вариант для нас совершенно неприемлем, поскольку он абсолютно непрофессионален, второй же вариант потребует читать файл 2 раза, что вряд ли улучшит эффективность программы.

Однако вспомним, что в составе стандартной библиотеки шаблонов *STL* (Standard Template Library) имеется набор контейнеров, которые достаточно просто разрешат наши проблемы. Выберем самый простой контейнер `vector`, где строки будем хранить в виде экземпляров стандартного класса `string`, что существенно упростит работу.

Текстовый файл будем читать, используя потоковые классы ввода/вывода.

Итак, добавляем в заголовок приложения операторы включения соответствующих библиотек `commdlg.h, vector, string, fstream`.

Здесь первая строчка необходима для подключения функций файлового диалога, а три следующих файла включений содержат определения классов: `vector`, `string`, `ifstream`, `ofstream`.

> ***Примечание***
>
> Все потоковые классы и библиотека STL определены в стандартной области имен `std`, однако подключение всего пространства имен директивой "`using namespace std`" не является хорошим решением. Для ускорения компиляции рекомендуется явно указывать область видимости для каждой переменной, например, `std::vector<std::string> v;`

Переменные, значение которых должно сохраниться после выхода из оконной функции, мы объявим как `static`:

- `name` — символьный массив для хранения имени открываемого файла;
- `file` — переменная типа структуры `OPENFILENAME`;
- `v` — вектор для данных типа `string`.

Остальные переменные могут быть автоматическими, потому что нет необходимости сохранять их значение между вызовами оконной функции.

Большую часть полей структуры `file` (см. табл. 2.1) разумно заполнить при обработке сообщения `WM_CREATE`, поскольку это сообщение обрабатывается только один раз при создании окна.

> ***Примечание***
>
> Поскольку переменная `file` описана как `static`, то нулевые значения можно не присваивать — эти поля и так будут инициированы нулями при создании переменной.

При выборе команды меню **New** мы должны очистить вектор и окно вывода.

Наиболее простой способ для этого — метод `clear()`, однако, очистив вектор, метод не освободит выделенной под него памяти, что не совсем хорошо. Даже если мы воспользуемся методом `v.resize(0)`, то и это не станет решением проблемы — он не сможет уменьшить размер выделенной памяти, поскольку предназначен лишь для увеличения размера контейнера.

Существует довольно элегантный прием для решения этой проблемы:

```
std::vector<std::string>().swap(v);
```

Здесь создается временный "пустой" вектор для переменных типа `string` и тут же обменивается с вектором `v`. Если вспомнить, что метод `swap()` обменивает не только содержимое контейнеров, но и их размеры, то становится ясно, что в результате мы получим пустой вектор `v` и временный вектор, содержащий предыдущий набор данных, который автоматически уничтожается деструктором.

Разумеется, если вектор и так "пустой", то очищать его нет смысла, поэтому мы используем условный оператор:

```
if (!v.empty()) . . .
```

Теперь, после того как вектор очищен, нужно перерисовать окно, как обычно обратившись к функции `InvalidateRect()`.

По команде **Open** мы заполним лишь два поля переменной `file`:

```
file.lpstrTitle = _T("Открыть файл для чтения");
file.Flags = OFN_HIDEREADONLY;
```

где первое поле обеспечит вывод заголовка диалогового окна, а второе поле можно было бы и не заполнять, поскольку `OFN_HIDEREADONLY` означает лишь то, что переключатель **Read Only** не выводится.

При обращении к функции `GetOpenFileName()` происходит вывод диалогового окна (рис. 2.1), где мы имеем возможность выбрать файл для чтения. Если работа функции завершилась успешно, то в символьном массиве `name` мы получим полное имя файла и сможем открыть его методом `open()` потокового класса.

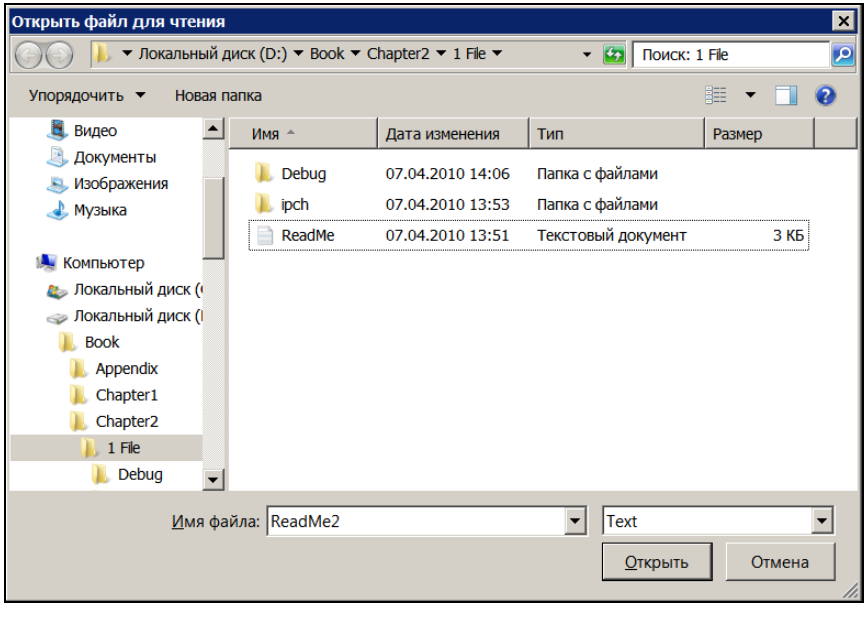


Рис. 2.1. Диалоговое окно выбора имени файла для чтения

Теперь файл построчно читаем в память. Лучше всего воспользоваться перегруженной для класса `string` функцией `getline()`, которая принимает в качестве параметров входной поток и переменную типа `string`. Функция читает из потока `in` строку и помещает ее в переменную `st`, причем конструктор класса `string` сам позаботится о выделении памяти. Далее, методом `v.push_back(st)` строка добавляется в конец контейнера. Возвращаемым значением функции `getline()` является входной поток `in`, поэтому цикл работает, пока весь файл не будет прочитан. При достижении конца файла функция `getline()` возвращает `NULL` и цикл прекращается.

Теперь входной поток нужно закрыть методом `close()` и перерисовать окно.

Обработка команды **Save** осуществляется аналогично. Мы также переопределяем значение двух полей `lpstrTitle` и `Flags`, изменяя заголовок диалогового окна и флаг открытия окна `OFN_NOTESTFILECREATE`, который позволяет выбрать имя существующего файла для замены.

При записи файла мы воспользовались диалогом `GetSaveFileName()`, который отличается от предыдущего только тем, что кнопка в диалоговом окне называется **Сохранить** вместо **Открыть** (рис. 2.2).

Для записи в файл содержимого контейнера организуем цикл от начала до конца контейнера по итератору `it`. Добавим в конце каждой строки символ перевода строки `'\n'`.

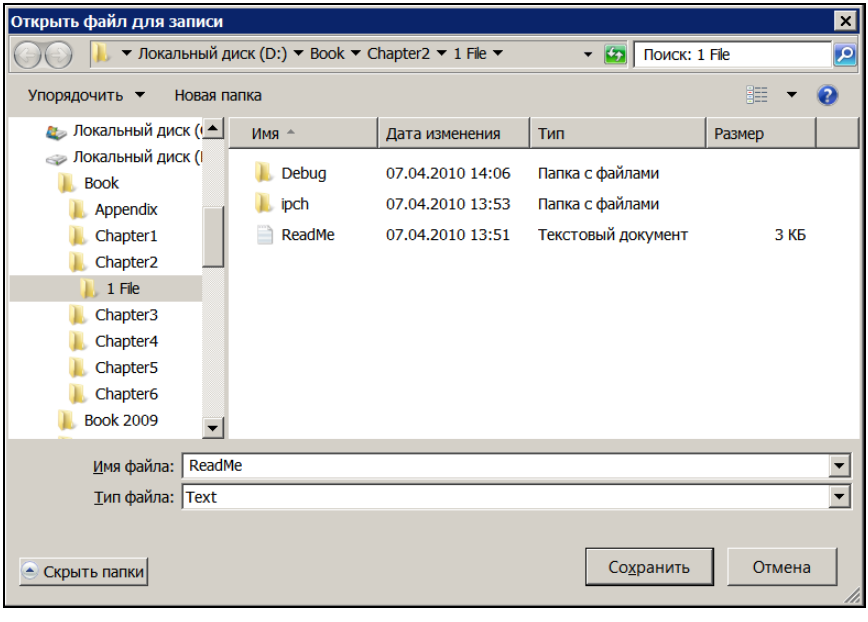


Рис. 2.2. Диалог выбора имени файла для записи

Теперь осталось организовать вывод содержимого контейнера в окно при обработке сообщения `WM_PAINT`. Мы можем выводить содержимое контейнера в окно по итератору `it`. Нужно только добавить переменную целого типа `y` для позициониро-

вания строк текста по вертикали. Будем увеличивать эту переменную на каждом шаге цикла на `LineHeight` — этого достаточно для шрифта по умолчанию.

Функция `TextOutA()` не требует задания С-строки, ей достаточно символьного массива типа `char*`, однако класс `string` имеет метод `data()`, возвращающий указатель на массив символов. Размер строки текста получим методом `length()`.

> ***Примечание***
>
> Поскольку приложение ориентировано на чтение файлов с однобайтной кодировкой, то мы должны явно указать суффикс "`A`" функции `TextOutA()`, что будет означать использование типа `char*` в качестве указателя строки.

Если все сделано правильно, то мы получим программу, читающую текстовый файл, выводящую его в окно и имеющую возможность сохранить его копию.

Приложение хотя и работает, но имеет ряд существенных недостатков:

1. Если весь текст не помещается в окне, часть его будет утеряна.
2. "Исчезают" символы, не имеющие графического образа, в частности символы табуляции.

Решение указанных проблем требует модернизации программы.

Для обработки символов табуляции проще всего заменить функцию вывода текста на `TabbedTextOutA()`:

```
TabbedTextOutA(hdc, 0, y, it->data(), it->length(), 0, NULL, 0);
```

Начальные параметры имеют тот же смысл, а три последних параметра определяют позиции табуляции. По умолчанию (нулевые значения параметров), символ табуляции разворачивается до средней ширины 8 символов.

Организация скроллинга

Скроллинг или прокрутку можно установить как свойство окна, добавив флаги горизонтального и вертикального скроллинга `WS_VSCROLL` и `WS_HSCROLL` в стиль создаваемого окна функции `CreateWindow()`.

```
hWnd = CreateWindow(szWindowClass, szTitle,
       WS_OVERLAPPEDWINDOW| WS_VSCROLL | WS_HSCROLL,
       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
```

Однако всю логику обслуживания полос прокрутки мы должны организовать самостоятельно (листинг 2.2). Используем в качестве основы предыдущую задачу, листинг 2.1, добавив сюда скроллинг.

Листинг 2.2. Оконная функция с полосами прокрутки

```
#include <commdlg.h>
#include <fstream>
#include <vector>
#include <string>
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static TCHAR name[256] = _T("");;
   static OPENFILENAME file;
   std::ifstream in;
   std::ofstream out;
   static std::vector<std::string> v;
   std::vector<std::string>::iterator it;
   std::string st;
   int y, k;
   static int n,length,sx,sy,cx,iVscrollPos,iHscrollPos,COUNT,MAX_WIDTH;
   static SIZE size = {8, 16}; //Ширина и высота символа
   switch (message)
   {
   case WM_CREATE:
      file.lStructSize = sizeof(OPENFILENAME);
      file.hInstance = hInst;
      file.lpstrFilter = _T("Text\0*.txt");
      file.lpstrFile = name;
      file.nMaxFile = 256;
      file.lpstrInitialDir = _T(".\\");
      file.lpstrDefExt = _T("txt");
      break;
   case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      k = n - sy/size.cy;
      if (k > 0) COUNT = k; else COUNT = iVscrollPos = 0;
      SetScrollRange(hWnd, SB_VERT, 0, COUNT, FALSE);
      SetScrollPos (hWnd, SB_VERT, iVscrollPos, TRUE);
      k = length - sx/size.cx;
      if (k > 0) MAX_WIDTH = k; else MAX_WIDTH = iHscrollPos = 0;
      SetScrollRange(hWnd, SB_HORZ, 0, MAX_WIDTH, FALSE);
      SetScrollPos(hWnd, SB_HORZ, iHscrollPos, TRUE);
      break;
   case WM_VSCROLL :
      switch(LOWORD(wParam))
      {
         case SB_LINEUP   : iVscrollPos--; break;
         case SB_LINEDOWN : iVscrollPos++; break;
         case SB_PAGEUP   : iVscrollPos -= sy / size.cy; break;
```

```
      case SB_PAGEDOWN : iVscrollPos += sy / size.cy; break;
      case SB_THUMBPOSITION : iVscrollPos = HIWORD(wParam); break;
    }
    iVscrollPos = max(0, min(iVscrollPos, COUNT));
    if (iVscrollPos != GetScrollPos(hWnd, SB_VERT))
    {
      SetScrollPos(hWnd, SB_VERT, iVscrollPos, TRUE);
      InvalidateRect(hWnd, NULL, TRUE);
    }
    break;
  case WM_HSCROLL :
    switch(LOWORD(wParam))
    {
      case SB_LINEUP   : iHscrollPos--; break;
      case SB_LINEDOWN : iHscrollPos++; break;
      case SB_PAGEUP   : iHscrollPos -= 8; break;
      case SB_PAGEDOWN : iHscrollPos += 8; break;
      case SB_THUMBPOSITION : iHscrollPos = HIWORD(wParam); break;
    }
    iHscrollPos = max(0, min(iHscrollPos, MAX_WIDTH));
    if (iHscrollPos != GetScrollPos(hWnd, SB_HORZ))
    {
      SetScrollPos(hWnd, SB_HORZ, iHscrollPos, TRUE);
      InvalidateRect(hWnd, NULL, TRUE);
    }
    break;
  case WM_COMMAND:
    switch (LOWORD(wParam))
    {
      case ID_FILE_NEW :
        if (!v.empty()) std::vector<std::string>().swap(v);
        n = length = 0;
        SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
        InvalidateRect(hWnd, NULL, TRUE);
        break;
      case ID_FILE_OPEN :
        file.lpstrTitle = _T("Открыть файл для чтения");
        file.Flags = OFN_HIDEREADONLY;
        if (!GetOpenFileName(&file)) return 1;
        in.open(name);
        while (getline(in,st))
        {
          if (length < st.length()) length = st.length();
          v.push_back(st);
```

```
            }
            in.close();
            n = v.size();
            SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
            InvalidateRect(hWnd, NULL, TRUE);
            break;
         case ID_FILE_SAVE :
            file.lpstrTitle = _T("Открыть файл для записи");
            file.Flags = OFN_NOTESTFILECREATE;
            if (!GetSaveFileName(&file)) return 1;
            out.open(name);
            for (it = v.begin(); it != v.end(); ++it) out << *it << '\n';
            out.close();
            break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      for(y=0,it=v.begin()+iVscrollPos;it!=v.end()&&y<sy;++it,y+=size.cy)
         if (iHscrollPos < it->length())
            TabbedTextOutA(hdc,0,y,it->data()+iHscrollPos,it->length()-
              iHscrollPos,0,NULL,0);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Первое, что необходимо сделать для создания полос прокрутки — задать диапазон и установить позицию движка. Две функции `SetScrollRange()` и `SetScrollPos()` позволят решить эту задачу:

```
BOOL WINAPI SetScrollRange(HWND hWnd, int nBar, int nMinPos, int nMaxPos, BOOL
bRedraw);
int WINAPI SetScrollPos(HWND hWnd, int nBar, int nPos, BOOL bRedraw);
```

- `hWnd` — дескриптор окна;
- `nBar` — идентификатор полосы прокрутки, здесь может принимать значение `SB_HORZ` или `SB_VERT`;

- nMinPos, nMaxPos — минимальное и максимальное значение позиции движка;
- nPos — текущая позиция движка;
- bRedraw — перерисовка полосы прокрутки, если значение поля TRUE, и нет перерисовки, если FALSE.

Удобнее всего установить параметры прокрутки в сообщении WM_SIZE, поскольку это сообщение обрабатывается при создании окна, и будет генерироваться системой при каждом изменении его размера.

Диапазон вертикальной прокрутки установим равным количеству прочитанных строк файла n за вычетом одной экранной страницы. Это желательно сделать, чтобы в конце текста не появлялось пустой страницы. Количество строк, помещающихся на одной странице, мы вычислим, поделив размер окна по вертикали на высоту строки sy/size.cy. Высоту окна получим здесь же, в сообщении WM_SIZE, извлекая старшее слово lParam, а геометрические параметры шрифта зададим при описании переменной:

```
static SIZE size = {8, 16};
```

Таким будет размер символа шрифта по умолчанию, как мы выяснили, исследуя системные шрифты в *главе 1* (см. листинг 1.13). Однако с такой арифметикой могут быть проблемы. Действительно, при создании окна переменная n равна 0, поскольку она объявлена как статическая и выражение:

```
k = n - sy/size.cy;
```

будет отрицательным. Такой же результат будет и при открытии файла, который целиком помещается в одном окне. Поэтому мы в следующей строке скорректируем значение переменной COUNT, которая и будет определять диапазон вертикального скроллинга.

```
if (k > 0) COUNT = k; else COUNT = iVscrollPos = 0;
```

Заодно присваиваем нулевое значение переменной iVscrollPos, являющейся индексом текущей позиции вертикального скроллинга.

Теперь установим диапазон и позицию скроллинга:

```
SetScrollRange(hWnd, SB_VERT, 0, COUNT, FALSE);
SetScrollPos (hWnd, SB_VERT, iVscrollPos, TRUE);
```

> ***Примечание***
>
> Если минимальное и максимальное значения позиции движка совпадают, то полоса прокрутки в окне не выводится.

Параметры горизонтального скроллинга устанавливаются аналогично.

Очевидно, что при открытии приложения, когда количество строк n и максимальная длина строки length равны нулю, полос прокрутки не будет, они должны появиться лишь при открытии файла достаточно большого размера.

Немного модифицируем обработчик сообщения WM_CREATE, добавив в поле фильтра еще один шаблон:

```
file.lpstrFilter = _T("Text\0 *.txt\0Все файлы\0 *.*");
```

Для корректной обработки команды **New** добавим две строки текста:

```
n = length = 0;
SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
```

Здесь мы обнулим переменные, описывающие количество строк файла и максимальную длину строки, после чего сгенерируем сообщение `WM_SIZE` при помощи функции `SendMessage()`, что вызовет переопределение параметров скроллинга:

```
LRESULT WINAPI SendMessageW(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lPa-
ram);
```

> ***Примечание***
>
> Этот прием генерации сообщения мы используем для экономии кода. Можно конечно оформить этот код отдельной функцией, но хотелось показать и эту возможность.

Функция `SendMessage()` обеспечивает передачу сообщения окну с дескриптором `hWnd`, дополнительная информация передается через `wParam` и `lParam`. Причем функция синхронная и возвращает управление только после обработки сообщения. Возвращаемое значение зависит от сообщения.

Поскольку в сообщении `WM_SIZE` мы получаем размеры окна в переменной `lParam`, то здесь мы должны самостоятельно сформировать его значение. В старшее слово `lParam` нужно поместить высоту окна — это мы сделаем операцией логического сдвига `sy<<16`, а ширину окна в младшее слово добавим операцией логического сложения. `wParam` здесь не используется, поэтому запишем `0`.

При обработке команды **Open** вычислим максимальное количество символов в строке, эта информация нам понадобится для определения горизонтального скроллинга. Добавим в цикл чтения файла строку:

```
if (length < st.length()) length = st.length();
```

По завершении чтения файла в переменной `length` будет храниться размер строки максимальной длины. Количество строк файла получим как размер контейнера:

```
n = v.size().
```

Перед тем как перерисовать окно, сгенерируем сообщение `WM_SIZE` для переустановки параметров скроллинга.

Теперь мы должны организовать логику работы с полосами прокрутки.

Любые действия с полосами прокрутки вызывают генерацию сообщения: `WM_VSCROLL` для вертикального и `WM_HSCROLL` для горизонтального скроллинга. В младшем слове `wParam` передается идентификатор команды. Мы будем обрабатывать лишь следующий набор команд:

- `SB_LINEUP` — щелчок мыши на стрелке вверх (влево);
- `SB_LINEDOWN` — щелчок мыши на стрелке вниз (вправо);
- `SB_PAGEUP` — щелчок мыши внутри полосы выше (левее) движка;
- `SB_PAGEDOWN` — щелчок мыши внутри полосы ниже (правее) движка;
- `SB_THUMBPOSITION` — отпускание движка после его буксировки.

Примечание

При рассмотрении горизонтального скроллинга логичнее было бы использовать символические константы `SB_LINELEFT`, `SB_LINERIGHT`, `SB_PAGELEFT`, `SB_PAGERIGHT`; они эквивалентны приведенным выше и представляют собой целые числа: `0`, `1`, `2`, `3`.

Для отслеживания позиции движков полос прокрутки используем две статические переменные целого типа `iVscrollPos` — для вертикального скроллинга и `iHscrollPos` — для горизонтального. По командам `SB_LINEUP` и `SB_LINEDOWN` уменьшим или увеличим значение переменной на `1`. По командам `SB_PAGEUP` и `SB_PAGEDOWN`, означающим перелистывание на страницу для горизонтального скроллинга, уменьшим или увеличим переменную `iVscrollPos` на количество помещающихся в окне строк, которое мы вычислим, поделив вертикальный размер окна на высоту строки: `sy/size.cy`. Для горизонтального скроллинга будем смещать переменную `iHscrollPos` на размер табуляции, равный 8 символам.

Последней обработаем команду `SB_THUMBPOSITION`, которая будет сгенерирована после отпускания движка по завершении его буксировки. Мы можем получить позицию движка из старшего слова `wParam`: `iVscrollPos = HIWORD(wParam)`.

Но если этим ограничиться, то будет возможно либо отрицательное значение, либо значение, превышающее максимальный размер скроллинга. Во избежание подобной ситуации воспользуемся следующей конструкцией:

```
iVscrollPos = max(0, min(iVscrollPos, COUNT));
```

`max()` и `min()` — макросы, определенные в файле включений `windows.h` для вычисления максимума и минимума двух арифметических выражений.

Теперь мы можем быть уверены, что переменная `iVscrollPos` не выйдет за пределы отрезка `[0; COUNT]`.

Затем проверяем, изменилась ли позиция движка?

```
if (iVscrollPos != GetScrollPos(hWnd, SB_VERT))
```

Сравним значение переменной `iVscrollPos` с возвращаемым значением функции `GetScrollPos()` и, если значения не совпадают, перерисуем полосу скроллинга с новым положением движка:

```
SetScrollPos(hWnd, SB_VERT, iVscrollPos, TRUE);
```

После чего перерисуем содержимое окна, вызывая функцию `InvalidateRect()`.

Для горизонтального скроллинга выполним аналогичные действия.

Нам осталось рассмотреть организацию вывода текста в сообщении `WM_PAINT`. Выводом будут управлять две переменные `iVscrollPos` и `iHscrollPos`, определяющие позиции скроллинга. Будем начинать вывод текста со строки с индексом `iVscrollPos`, а горизонтальное смещение определим индексом `iHscrollPos`.

В цикле `for(...)` блок начальной инициализации будет выглядеть так:

```
y = 0, it = v.begin() + iVscrollPos;
```

Переменная `y`, как и ранее, используется для вычисления *y*-координаты выводимой строки текста, а итератор `it`, которым мы выбираем строки текста из контейнера, сместим на `iVscrollPos` позиций. Для контейнера `vector` эта операция допустима и итератор сейчас показывает на строку текста с индексом `iVscrollPos`.

Рассмотрим, как записано условие:

```
it < v.end() && y < sy
```

Цикл завершится, если будет достигнут конец контейнера или *y*-координата строки выйдет за пределы окна.

Приращение координат:

```
++it, y += size.cy
```

Итератор `it` увеличивается на `1`, а *y*-координата на высоту строки.

Осталось вывести строку текста с учетом обработки табуляций:

```
TabbedTextOutA(hdc,0,y, it->data()+iHscrollPos, it->length()-iHscrollPos,
               0, NULL, 0);
```

Выводим строки текста со смещением `iHscrollPos`, размер строки уменьшается на эту же величину: `it->length()-iHscrollPos`.

> ***Примечание***
>
> Здесь мы поступаем не совсем корректно, поскольку выводим строки до конца, не заботясь о том, что часть строки может выйти за пределы окна. Это несколько ухудшает эффективность программы, но упрощает ее логику.

Панель инструментов

Чтобы получить максимально полное представление о возможностях Windows, можно создать для нашей задачи инструментальную панель, которая позволит выполнять команды меню нажатием на соответствующую кнопку. Для трех пунктов меню **New**, **Open** и **Save** создадим панель инструментов со стандартными для этих команд кнопками.

Традиционный путь создания панели инструментов заключается в вызове функции `CreateToolbarEx()`, прототип которой помещен в файл включений `commctrl.h`.

```
HWND WINAPI CreateToolbarEx(HWND hWnd, DWORD ws, UINT wID, int nBitmaps,
HINSTANCE hBMInst, UINT wBMID, LPCTBBUTTON lpButtons,int iNumButtons,
int dxButton, int dyButton, int dxBitmap, int dyBitmap,UINT uStructSize);
```

13 параметров этой функции имеют следующее значение:

- `HWnd` — описатель родительского окна;
- `Ws` — флаги стиля окна;
- `wID` — идентификатор дочернего окна панели инструментов;
- `nBitmaps` — число изображений в битовом образе;
- `hBMInst` — дескриптор ресурса битового образа для загрузки;
- `wBMID` — идентификатор ресурса битового образа;
- `lpButtons` — указатель на массив структур `TBBUTTON`, которые содержат информацию о кнопках;
- `iNumButtons` — число кнопок инструментальной панели;

- dxButton — ширина кнопок в пикселах;
- dyButton — высота кнопок в пикселах;
- dxBitmap — ширина, в пикселах, кнопки битового образа;
- dyBitmap — высота, в пикселах, кнопки битового образа;
- uStructSize — размер структуры TBBUTTON.

Информация о кнопках размещается в массиве структур TBBUTTON.

```
struct TBBUTTON
{
int iBitmap;          // индекс изображения кнопки
int idCommand;        // идентификатор команды, соответствующей кнопке
BYTE fsState;         // определяют начальное состояние и
BYTE fsStyle;         // стиль кнопки
BYTE bReserved[2];    // резерв
DWORD dwData;         // определенное приложением значение
int iString;          // индекс текстовой метки кнопки
};
```

Для описания включаемых в приложение кнопок в Windows имеется два набора готовых для использования битовых образов. Первый набор содержит изображения 15 кнопок, соответствующих командам меню **File** и **Edit**, откуда мы и позаимствуем необходимые изображения. Идентификаторы кнопок вполне отражают их назначение:

```
STD_CUT, STD_COPY, STD_PASTE, STD_UNDO, STD_REDOW, STD_DELETE,
STD_FILENEW, STD_FILEOPEN, STD_FILESAVE, TD_PRINTPRE, STD_PROPERTIES,
STD_HELP, STD_FIND, STD_REPLACE, STD_PRINT.
```

Внесем в программу (листинг 2.2) необходимые изменения.

1. Добавим файл включения директивой:

   ```
   #include <commctrl.h>
   ```

2. На глобальном уровне опишем 3 кнопки:

   ```
   TBBUTTON tbb[] ={
   {STD_FILENEW, ID_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON,0,0,0,0},
   {STD_FILEOPEN, ID_FILE_OPEN,TBSTATE_ENABLED, TBSTYLE_BUTTON,0,0,0,0},
   {STD_FILESAVE, ID_FILE_SAVE,TBSTATE_ENABLED, TBSTYLE_BUTTON,0,0,0,0}
   };
   ```

   Строка описания: имя кнопки, ее идентификатор, константа TBSTATE_ENABLED, которая означает, что кнопка активна; стиль кнопки стандартный — TBSTYLE_BUTTON, остальные 4 параметра здесь не используются, и мы заполним их нулями.

   Поскольку мы выбрали для идентификаторов кнопок те же значения, что и для пунктов меню, нажатие на кнопку панели инструментов генерирует те же сообщения, что и команды меню — WM_COMMAND со значением идентификатора соответствующего пункта меню в младшем слове wParam. Таким образом, нет необходимости писать код для обработки сообщения о нажатии на кнопку панели

инструментов, так как эти сообщения будут обработаны тем же кодом, что и команды меню.

3. Опишем дескриптор панели инструментов:

```
static HWND hwndToolBar;
```

4. Теперь вызовем функцию `CreateToolbarEx()` и передадим ей необходимые параметры. Это можно сделать в сообщении `WM_CREATE`.

```
hwndToolBar = CreateToolbarEx(hWnd, WS_CHILD | WS_VISIBLE | CCS_TOP,1,
0,HINST_COMMCTRL,IDB_STD_SMALL_COLOR,tbb,3,0,0,0,0,sizeof(TBBUTTON));
```

Стиль окна определяется логической суммой трех констант: `WS_CHILD` — окно является дочерним; `WS_VISIBLE` — окно отображается автоматически (если не включить этот стиль, то необходимо использовать функцию `ShowWindow()`); `CCS_TOP` — окно размещается сверху под строкой меню, причем размер по вертикали устанавливается исходя из размеров кнопок, а горизонтальный размер устанавливается по ширине родительского окна.

Следующий параметр — идентификатор окна панели инструментов, присваиваем ему номер `1`. Далее `0`, поскольку для стандартного ресурса `HINST_COMMCTRL` этот параметр не используется. Идентификатор ресурса выберем `IDB_STD_SMALL_COLOR` для "маленьких" (16×16) кнопок. Далее описывается `tbb` — указатель на массив структур `TBBUTTON` и количество элементов — `3`. Следующие 4 поля для стандартных элементов управления не используются и заполняются нулевым значением, а последним параметром стоит размер структуры `sizeof(TBBUTTON)`.

> ***Примечание***
>
> Следует иметь в виду, что для хранения изображения всех элементов панели инструментов используется один битовый образ, а извлечение необходимого образа происходит по индексу.

5. Необходимо предусмотреть изменение размеров панели инструментов при изменении размеров окна, иначе панель инструментов будет выглядеть не очень привлекательно. Лучше всего сделать это при обработке сообщения `WM_SIZE` посылкой сообщения `TB_AUTOSIZE`:

```
SendMessage(hwndToolBar, TB_AUTOSIZE, 0, 0);
```

что приведет к автоматической корректировке размера панели инструментов, и она гарантированно будет занимать всю верхнюю строку.

6. Теперь осталось решить последнюю проблему: дело в том, что панель инструментов, являясь дочерним окном, лежит внутри клиентской области окна и при выводе текста перекроет его верхнюю часть. Можно решить эту проблему, сместив начальную точку вывода текста. При обработке сообщения `WM_SIZE` вычислим высоту панели инструментов при помощи функции `GetWindowRect()`, которая возвращает координаты окна относительно рабочего стола:

```
GetWindowRect(hwndToolBar, &rt);
size_Toolbar = rt.bottom - rt.top;
```

переменные `rt` и `size_Toolbar` предварительно определим в оконной функции

```
static int size_Toolbar;
RECT rt;
```

В сообщении `WM_PAINT` при начальной инициализации цикла переменной `y` зададим значение:

```
for(y = size_Toolbar,...),
```

при этом первая строка текста будет смещена ниже панели инструментов. Однако в этом случае необходимо также скорректировать и размах скроллинга по вертикали, поскольку он зависит от размеров окна. В сообщении `WM_SIZE` отредактируем строку:

```
k = n - (sy - size_Toolbar)/size.cy;
```

7. Мы все сделали правильно, но при компиляции будет выдано сообщение об ошибке:

```
error LNK2001: unresolved external symbol __imp__CreateToolbarEx@52
```

Дело в том, что библиотека, в которой находится тело функции `CreateToolbarEx()`, по умолчанию не подключается к проекту. Необходимо сделать это *"вручную"*. Откроем окно свойств проекта в диалоговом окне **Solution Explorer**. На подложке **Linker | Input** в окне редактирования **Additional Dependencies** добавляем к списку библиотечных файлов `comctl32.lib`.

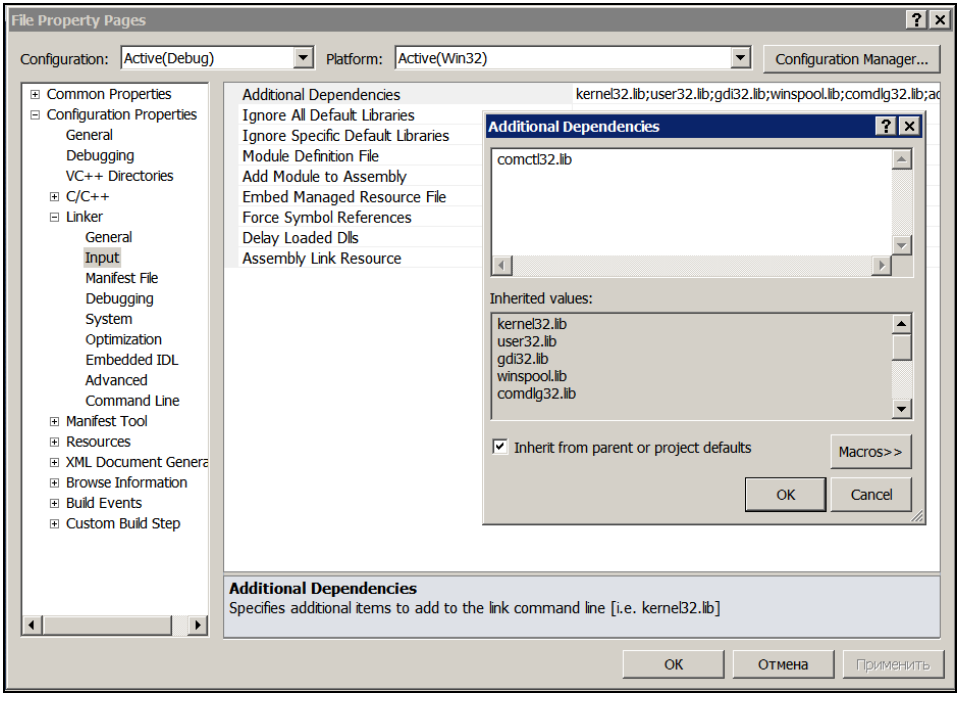


Рис. 2.3. Диалоговое окно настройки параметров проекта

Примечание

Имя файла `comctl32.lib` выглядит не очень красиво, но это наследие операционной системы MS-DOS, где имя файла не превышало 8 символов.

Теперь компиляция должна пройти успешно. На рис. 2.4 показан пример работы созданного приложения.

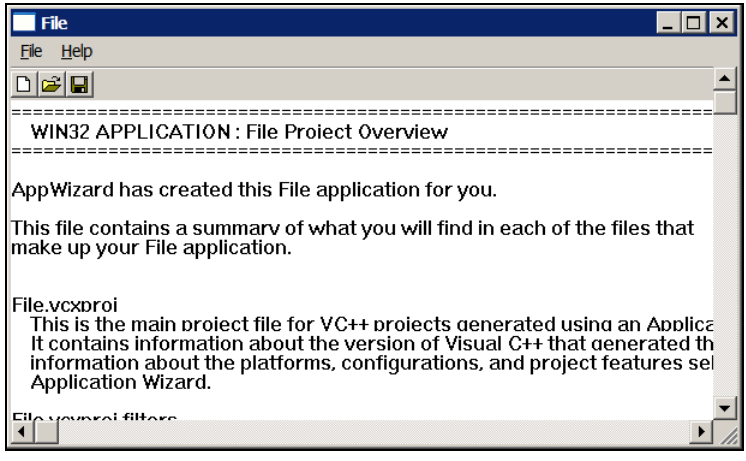


Рис. 2.4. Работа приложения с панелью инструментов

Выбор шрифтов

Можно и далее наращивать функциональные возможности программы просмотра файлов. Следующее расширение позволит программе произвольно выбирать шрифт для вывода текста. Воспользуемся имеющимся в системе стандартным диалогом выбора шрифта.

Для обращения к диалоговому окну выбора шрифта необходимо описать две переменные типа `CHOOSEFONT` и `LOGFONT`, где первая обеспечивает взаимодействие с функцией диалога, а вторая требуется для хранения информации о выбранном шрифте. Обе структуры имеют большое количество полей и определены в файлах включений `commdlg.h` и `wingdi.h`.

Финальный текст оконной функции приведен в листинге 2.3, здесь же обсудим лишь новые элементы. Начнем с описания переменных:

```
static LOGFONT lf;
static CHOOSEFONT cf;
static HFONT hfont;
```

Нам потребуется определить лишь 4 поля структуры `CHOOSEFONT`. Сделаем это в сообщении `WM_CREATE`:

```
cf.lStructSize = sizeof(CHOOSEFONT);
cf.Flags = CF_EFFECTS | CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS;
```

```
cf.hwndOwner = hWnd;
cf.lpLogFont = &lf;
```

Первое поле содержит размер структуры `CHOOSEFONT`. Следующие поля определяют флаги и дескриптор родительского окна, а последнее поле хранит указатель на переменную типа `LOGFONT`.

`Flags` задает режим работы диалогового окна:

- `CF_EFFECTS` — позволяет пользователю определять зачеркнутый, подчеркнутый шрифт, а также цвет текста;
- `CF_INITTOLOGFONTSTRUCT` — используется структура `LOGFONT`;
- `CF_SCREENFONTS` — выводится список только экранных шрифтов, поддерживаемых системой.

Теперь можно строить обращение к функции диалога `ChooseFont()`:

```
BOOL APIENTRY ChooseFont(LPCHOOSEFONT);
```

Функция принимает указатель на переменную типа `CHOOSEFONT` и возвращает `TRUE` при успешной работе диалога.

Добавим в выпадающее меню **Оформление** пункт **Шрифт** с идентификатором `ID_FONT` и построим его обработчик.

> ***Примечание***
>
> Перед открытием окна **Menu** нужно установить в поле свойств **Language** — Русский (Россия).

После выбора шрифта мы, чтобы "не замусорить" память, уничтожим предыдущий шрифт, если он существовал, при помощи конструкции `if (hfont) ...`

Затем создадим шрифт функцией `CreateFontIndirect()`.

В качестве параметра функция принимает указатель на структуру `LOGFONT` и заполняет ее поля в ходе диалога. Теперь необходимо определить метрику текста: нам нужна высота шрифта и средняя ширина символа. Определить эти параметры можно обращением к функции `GetTextMetrics()`, которую мы уже рассматривали в *главе 1*. Однако предварительно необходимо получить контекст устройства `hdc` и выбрать созданный шрифт текущим. Среднюю ширину символа мы получим из поля `tmAveCharWidth` структуры `TEXTMETRIC`, которую также нужно предварительно описать:

```
TEXTMETRIC tm;
```

Высоту строки дает выражение: `tm.tmHeight + tm.tmExternalLeading`.

Осталось переопределить параметры скроллинга, послав сообщение `WM_SIZE`:

```
SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
```

и перерисовать окно вызовом функции `InvalidateRect()`.

В сообщении `WM_PAINT` необходимо добавить код для переключения на выбранный шрифт и установить его цвет.

И последний штрих. Практически все современные манипуляторы типа "мышь" имеют среднее колесико, которое обычно используется для прокрутки текста. Было

бы уместно и нам воспользоваться этой возможностью, тем более, что это довольно просто.

Прокрутка колесика мыши порождает сообщение `WM_MOUSEWHEEL`, а в старшем слове `wParam` возвращается количество шагов колесика, умноженное на константу `WHEEL_DELTA` (значение константы 120). Нам достаточно поделить это значение на данную константу, и мы получим количество шагов. Позаботимся о том, чтобы не выйти за границы скроллинга, установим новую позицию движка и перерисуем окно.

```
case WM_MOUSEWHEEL:
       iVscrollPos -= (short)HIWORD(wParam)/WHEEL_DELTA;
       iVscrollPos = max(0, min(iVscrollPos, COUNT));
       SetScrollPos(hWnd, SB_VERT, iVscrollPos, TRUE);
       InvalidateRect(hWnd, NULL, TRUE);
       break;
```

Примечание

Принципиально важно задать явное преобразование типа `(short)HIWORD(wParam)`, иначе отрицательное значение в старшем слове `wParam` будет интерпретироваться как очень большое положительное число.

Листинг 2.3. Окончательный вариант оконной функции программы просмотра файлов

```
#include <commdlg.h>
#include <fstream>
#include <vector>
#include <string>
#include <commctrl.h>
TBBUTTON tbb[] =
{
{STD_FILENEW, ID_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON,0,0,0,0},
{STD_FILEOPEN, ID_FILE_OPEN,TBSTATE_ENABLED, TBSTYLE_BUTTON,0,0,0,0},
{STD_FILESAVE, ID_FILE_SAVE,TBSTATE_ENABLED, TBSTYLE_BUTTON,0,0,0,0}
};
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static TCHAR name[256] = _T("");;
   static OPENFILENAME file;
   std::ifstream in;
   std::ofstream out;
   static std::vector<std::string> v;
   std::vector<std::string>::iterator it;
```

```
std::string st;
int y, k;
static int n,length,sx,sy,cx,iVscrollPos,iHscrollPos,COUNT,MAX_WIDTH;
static SIZE size = {8, 16 };
static HWND hWndToolBar;
static int size_Toolbar;
RECT rt;
static LOGFONT lf;
static CHOOSEFONT cf;
static HFONT hfont;
TEXTMETRIC tm;
switch (message)
{
case WM_CREATE:
  file.lStructSize = sizeof(OPENFILENAME);
  file.hInstance = hInst;
  file.lpstrFilter = _T("Text .txt\0 *.txt\0Все файлы\0 *.*");
  file.lpstrFile = name;
  file.nMaxFile = 256;
  file.lpstrInitialDir = _T(".\\");
  file.lpstrDefExt = _T("txt");
  hWndToolBar = CreateToolbarEx(hWnd,WS_CHILD|WS_VISIBLE|CCS_TOP,1,0,
  HINST_COMMCTRL,IDB_STD_SMALL_COLOR,tbb,3,0,0,0,0,sizeof(TBBUTTON));
  cf.lStructSize = sizeof(CHOOSEFONT);
  cf.Flags = CF_EFFECTS | CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS;
  cf.hwndOwner = hWnd;
  cf.lpLogFont = &lf;
  break;
case WM_SIZE:
  sx = LOWORD(lParam);
  sy = HIWORD(lParam);
  k = n - (sy - size_Toolbar)/size.cy;
  if (k > 0) COUNT = k; else COUNT = iVscrollPos = 0;
  SetScrollRange(hWnd, SB_VERT, 0, COUNT, FALSE);
  SetScrollPos (hWnd, SB_VERT, iVscrollPos, TRUE);
  k = length - sx/size.cx;
  if (k > 0) MAX_WIDTH = k; else MAX_WIDTH = iHscrollPos = 0;
  SetScrollRange(hWnd, SB_HORZ, 0, MAX_WIDTH, FALSE);
  SetScrollPos(hWnd, SB_HORZ, iHscrollPos, TRUE);
  SendMessage(hWndToolBar, TB_AUTOSIZE, 0, 0);
  GetWindowRect(hWndToolBar, &rt);
  size_Toolbar = rt.bottom - rt.top;
  break;
```

```
case WM_MOUSEWHEEL:
  iVscrollPos -= (short)HIWORD(wParam)/WHEEL_DELTA;
  iVscrollPos = max(0, min(iVscrollPos, COUNT));
  SetScrollPos(hWnd, SB_VERT, iVscrollPos, TRUE);
  InvalidateRect(hWnd, NULL, TRUE);
  break;
case WM_VSCROLL :
  switch(LOWORD(wParam))
     {
        case SB_LINEUP   : iVscrollPos--; break;
        case SB_LINEDOWN : iVscrollPos++; break;
        case SB_PAGEUP   : iVscrollPos -= sy/size.cy; break;
        case SB_PAGEDOWN : iVscrollPos += sy/size.cy; break;
        case SB_THUMBPOSITION : iVscrollPos = HIWORD(wParam); break;
     }
     iVscrollPos = max(0, min(iVscrollPos, COUNT));
     if (iVscrollPos != GetScrollPos(hWnd, SB_VERT))
     {
          SetScrollPos(hWnd, SB_VERT, iVscrollPos, TRUE);
          InvalidateRect(hWnd, NULL, TRUE);
     }
break;
case WM_HSCROLL :
  switch(LOWORD(wParam))
     {
         case SB_LINEUP   : iHscrollPos--; break;
         case SB_LINEDOWN : iHscrollPos++; break;
         case SB_PAGEUP   : iHscrollPos -= 8; break;
         case SB_PAGEDOWN : iHscrollPos += 8; break;
         case SB_THUMBPOSITION : iHscrollPos = HIWORD(wParam); break;
     }
     iHscrollPos = max(0, min(iHscrollPos, MAX_WIDTH));
     if (iHscrollPos != GetScrollPos(hWnd, SB_HORZ))
     {
          SetScrollPos(hWnd, SB_HORZ, iHscrollPos, TRUE);
          InvalidateRect(hWnd, NULL, TRUE);
     }
break;
case WM_COMMAND:
  switch (LOWORD(wParam))
  {
     case ID_FILE_NEW :
        if (!v.empty()) std::vector<std::string>().swap(v);
```

```
        n = length = 0;
        SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
        InvalidateRect(hWnd,NULL,TRUE);
        break;
      case ID_FILE_OPEN :
        file.lpstrTitle = _T("Открыть файл для чтения");
        file.Flags = OFN_HIDEREADONLY;
        if (!GetOpenFileName(&file)) return 1;
        in.open(name);
        while (getline(in,st))
        {
          if (length < st.length()) length = st.length();
          v.push_back(st);
        }
        in.close();
        n = v.size();
        SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
        InvalidateRect(hWnd,NULL,1);
        break;
      case ID_FILE_SAVE :
        file.lpstrTitle = _T("Открыть файл для записи");
        file.Flags = OFN_NOTESTFILECREATE;
        if (!GetSaveFileName(&file)) return 1;
        out.open(name);
        for (it = v.begin(); it != v.end(); ++it) out << *it << '\n';
        out.close();
        break;
    case ID_FONT :
        if(ChooseFont(&cf))
        {
          if (hfont) DeleteObject(hfont);
          hfont = CreateFontIndirect(&lf);
          hdc = GetDC(hWnd);
          SelectObject(hdc, hfont);
          GetTextMetrics(hdc, &tm);
          size.cx = tm.tmAveCharWidth;
          size.cy = tm.tmHeight + tm.tmExternalLeading;
          ReleaseDC(hWnd, hdc);
          SendMessage(hWnd, WM_SIZE, 0, sy << 16 | sx);
          InvalidateRect(hWnd, NULL, TRUE);
        }
        break;
  case IDM_EXIT: DestroyWindow(hWnd); break;
```

```
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
    case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      if (hfont)
      {  SelectObject(hdc, hfont);
         SetTextColor(hdc, cf.rgbColors);
      }
      for (y = size_Toolbar, it = v.begin() + iVscrollPos;
            it != v.end() && y < sy; ++it, y += size.cy)
         if (iHscrollPos < it->length())
            TabbedTextOutA(hdc, 0, y, it->data()+iHscrollPos,
                     it->length()-iHscrollPos, 0, NULL, 0);
      EndPaint(hWnd, &ps);
      break;
    case WM_DESTROY:
      if (hfont) DeleteObject(hfont);
      PostQuitMessage(0);
      break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Пожалуй, это все, что можно было сделать на данном этапе (рис. 2.5).

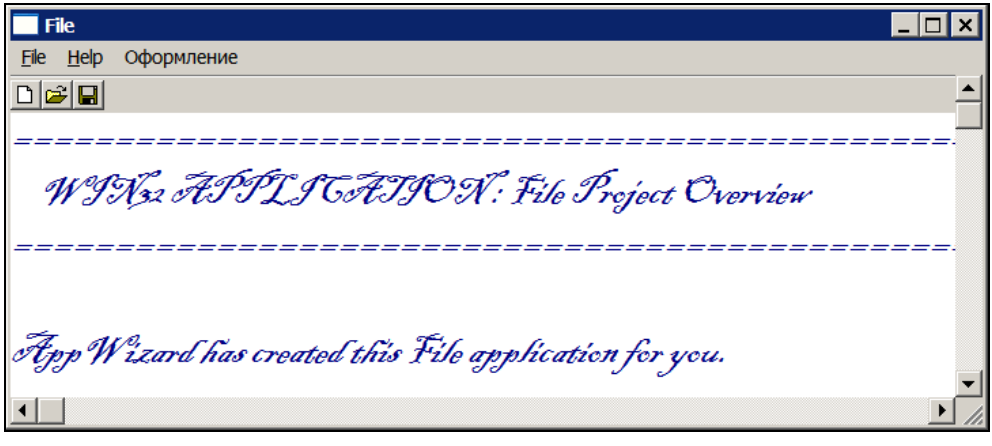


Рис. 2.5. Окно финальной версии программы-просмотрщика текстовых файлов

Однако мы должны отдавать себе отчет, что не все проблемы нашли удовлетворительное решение:

1. Так, максимальная длина строки, необходимая для организации горизонтального скроллинга, определена не совсем корректно при наличии в тексте символов табуляции.
2. Нам пришлось выводить текст до конца строки, даже если он выходит за границу окна, поскольку решение этой проблемы доступными на данном уровне средствами требует слишком много усилий.
3. Горизонтальный скроллинг будет работать удовлетворительно для моноширинных шрифтов, но для TrueType, шрифтов, когда ширина символов различна, строки текста будут "плавать" при скроллинге, и мы можем потерять "хвосты" длинных сток.

Если первые две проблемы можно как-то решить, написав дополнительный код, то в последнем случае у нас пока нет подходящего механизма.

Наиболее просто эти проблемы разрешаются при использовании виртуальных окон, о чем мы будем говорить в *главе 4*.

Чтение и запись файлов в библиотеке Win32 API

В Win32 API имеется свой набор функций, обеспечивающих работу с файлами. Мы не будем долго задерживаться на этом вопросе, поскольку нас пока вполне устраивает библиотека потоковых классов, обеспечивающая достаточно хороший сервис. Однако для полноты картины посмотрим, как можно создать программу чтения и записи текстовых файлов, используя имеющийся набор API-функций (листинг 2.4).

Нам понадобятся 3 функции:

1. Для открытия или создания файла используется функция:

```
HANDLE WINAPI CreateFileW(
 LPCWSTR lpFileName,              //имя файла
 DWORD dwDesiredAccess,           //способ доступа
 DWORD dwShareMode,               //совместный доступ
 LPSECURITY_ATTRIBUTES lpSecurityAttributes,   //атрибуты доступа
 DWORD dwCreationDisposition,     //проверка существования
 DWORD dwFlagsAndAttributes,      //атрибуты файла
 HANDLE hTemplateFile);           //дескриптор временного файла
```

2. Читать из файла будем функцией:

```
BOOL WINAPI ReadFile(
 HANDLE hFile,                    //дескриптор файла
 LPVOID lpBuffer,                 //буфер в памяти
 DWORD nNumberOfBytesToRead,      //максимальное число байтов
 LPDWORD lpNumberOfBytesRead,     //указатель переменной, возвращающей
                                  //количество фактически прочитанных байтов
 LPOVERLAPPED lpOverlapped);      //указатель на структуру OVERLAPPED
```

3. Пишем в файл функцией:

```
BOOL WINAPI WriteFile(
 HANDLE hFile,                    //дескриптор файла
 LPCVOID lpBuffer,                //буфер в памяти
 DWORD nNumberOfBytesToWrite,     //число записываемых байтов
 LPDWORD lpNumberOfBytesWritten,  //указатель переменной,
                                  //возвращающей количество фактически
                                    записанных байтов
 LPOVERLAPPED lpOverlapped);      // указатель на структуру OVERLAPPED
```

Более подробную информацию об этих функциях можно почерпнуть в *MSDN* (Microsoft Developer Network), а функцию `CreateFile()` мы рассмотрим подробнее в *главе 6*.

Листинг 2.4. Организация чтения/записи файла в библиотеке API-функций

```
#include <commdlg.h>
const DWORD MaxLength = 0x7fff;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static TCHAR name[256] = _T("");;
   static OPENFILENAME file;
   DWORD result;
   static HANDLE hFile;
   static char text[MaxLength];
   static int sx, sy;
   static DWORD nCharRead;
   RECT rt;
   switch (message)
   {
   case WM_CREATE:
      file.lStructSize = sizeof(OPENFILENAME);
      file.hInstance = hInst;
      file.lpstrFilter = _T("Text\0*.txt\0Все файлы\0*.*");
      file.lpstrFile = name;
      file.nMaxFile = 256;
      file.lpstrInitialDir = _T(".\\");
      file.lpstrDefExt = _T("txt");
      break;
   case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
```

```
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
         case ID_FILE_NEW :
            nCharRead = 0;
            InvalidateRect(hWnd, NULL, TRUE);
            break;
         case ID_FILE_OPEN :
            file.lpstrTitle = _T("Открыть файл для чтения");
            file.Flags = OFN_HIDEREADONLY;
            if (!GetOpenFileName(&file)) return 1;
            hFile = CreateFile(name, GENERIC_READ, 0, NULL,
               OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
            ReadFile(hFile, text, MaxLength, &nCharRead, NULL);
            CloseHandle(hFile);
            if (nCharRead == MaxLength)
            {
               MessageBox(hWnd, _T("Слишком большой файл"),
               _T("Неудачное открытие файла"), MB_YESNO | MB_ICONHAND);
               return 0;
            }
            InvalidateRect(hWnd, NULL,TRUE);
            break;
         case ID_FILE_SAVE :
            file.lpstrTitle = _T("Открыть файл для записи");
            file.Flags = OFN_NOTESTFILECREATE;
            if (!GetSaveFileName(&file)) return 1;
            hFile = CreateFile(name, GENERIC_WRITE, 0, NULL,
               CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
            WriteFile(hFile, text, nCharRead, &result, NULL);
            CloseHandle(hFile);
            break;
      case IDM_EXIT:  DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      SetRect(&rt, 0, 0, sx, sy);
      DrawTextA(hdc, text, nCharRead, &rt, DT_LEFT);
      EndPaint(hWnd, &ps);
      break;
```

```
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Диалог открытия файла для чтения и записи позаимствуем у предыдущей задачи (см. листинг 2.1), оттуда же возьмем обработчики сообщений `WM_CREATE` и `WM_SIZE`. Открываем файла для чтения функцией `CreateFile()`:

```
hFile = CreateFile(name, GENERIC_READ, 0, NULL, OPEN_EXISTING,
                   FILE_ATTRIBUTE_NORMAL, NULL);
```

Первый параметр `name` — полное имя файла. `GENERIC_READ` означает, что файл открывается для чтения. Следующий параметр `0` — совместного доступа к файлу нет. `NULL` — атрибуты доступа наследуются от порождающего процесса. `OPEN_EXISTING` — проверка существования файла. `FILE_ATTRIBUTE_NORMAL` — файл с "нормальными" атрибутами (т. е. файл не объявлен, как скрытый, системный и т. д.) и последний параметр `NULL` — временный файл не создается.

Возвращаемое значение функции `hFile` — дескриптор файла. Здесь было бы уместно проверить, насколько удачно завершилась операция открытия файла. Если файл не смог открыться, функция возвращает `0`.

Читаем файл одним блоком функцией `ReadFile()`:

```
ReadFile(hFile, text, MaxLength, &nCharRead, NULL);
```

Здесь `hFile` — дескриптор открытого файла. `text` — указатель символьного массива, куда считывается максимально `MaxLength` байтов. В переменной `nCharRead` мы получим фактическое число прочитанных байтов. Последний параметр `NULL` означает, что нет перекрытия читаемого блока данных.

Здесь мы для упрощения задачи ограничились максимальным размером файла в `MaxLength = 0x7fff` байтов. Вообще-то функция может читать блоки размером до 64 К. Смысл нашего ограничения будет понятен, когда мы рассмотрим элементы управления в *главе 3*.

Если же размер файла окажется больше, то будет прочитан блок в `MaxLength` байтов, а `nCharRead` вернет это же значение. В этом случае мы в диалоговом окне выводим сообщение **`"Слишком большой файл"`**.

При записи открываем файл также функцией `CreateFile()`:

```
hFile = CreateFile(name, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
                   FILE_ATTRIBUTE_NORMAL, NULL);
```

где вторым параметром стоит идентификатор `GENERIC_WRITE`, означающий, что файл открыт для записи, а параметр `CREATE_ALWAYS` означает, что создается новый файл, существующий файл будет уничтожен.

Записываем в файл также одним блоком функцией `WriteFile()`:

```
WriteFile(hFile, text, nCharRead, &result, NULL);
```

где все параметры имеют тот же смысл, что и для функции `ReadFile()`. Нам пришлось описать переменную `result`, в которую возвращается количество записанных байтов. Эту переменную можно использовать для контроля правильности записи. Если блок данных записан на диск, то ее значение совпадает с `nCharRead`.

Как при чтении, так и при записи, закрываем файл функцией `CloseHandle()`.

Поскольку в этой задаче мы работаем с символьным массивом, а не С-строкой, то в меню **New** достаточно обнулить число прочитанных байтов и перерисовать окно:

```
nCharRead = 0;
```

При выводе содержимого файла в сообщении `WM_PAINT` возникают определенные сложности. Дело в том, что мы прочитали файл как байтовый массив, имеющий ту же структуру, что и текстовый файл, в частности в конце строки присутствует пара символов — `'\r\n'`. Можно, конечно, "разобрать" файл на строки и выводить в окно построчно, но можно поступить проще, воспользовавшись функцией `DrawText()`, которая и предназначена для вывода текстового массива такой структуры. Предварительно только нужно создать прямоугольник размером с окно:

```
SetRect(&rt, 0, 0, sx, sy);
```

Текст выведем следующей конструкцией:

```
DrawTextA(hdc, text, nCharRead, &rt, DT_LEFT);
```

где мы установили выравнивание текста влево.

Мы не будем останавливаться на вопросах организации скроллинга. Предложим эту задачу для самостоятельной работы.

Вопросы к главе

1. Формат функций `GetOpenFileName()`, `GetSaveFileName()`.
2. Скроллинг как свойство окна. Настройка параметров скроллинга.
3. Обработка сообщений `WM_VSCROLL` и `WM_HSCROLL`.
4. Генерация сообщения функцией `SendMessage()`.
5. Панель инструментов, структура `TBBUTTON` и функция `CreateToolbarEx()`.
6. Как производится корректировка размеров панели инструментов?
7. Диалог выбора шрифта, функция `ChooseFont()`.
8. API-функции для чтения и записи файла.

Задания для самостоятельной работы

1. Дополнить фильтр выбора имени файла возможностью выбора файлов с расширением `dat` и без расширения имени.
2. В листинге 2.2 обработать сообщение о перемещении движка полосы скроллинга `SB_THUMBTRACK` и колесика мыши.

3. Заменить контейнер `vector`, использовавшийся для хранения текстовых строк (листинг 2.3), на контейнер `list`.
4. Дополнить программу просмотра текстового файла (листинг 2.3) пунктом меню для чтения текстового файла в кодировке DOS и Windows. Реализовать перекодировку текста для корректного вывода в окне.

   Указание: воспользоваться функциями `CharToOem()` и `OemToChar()`.
5. Сделать неактивными пункт меню **Save** и кнопку **Сохранить** на панели инструментов при загрузке приложения и по команде **New**. После открытия файла командой **Open** активизировать кнопку и пункт меню **Save**.

   Указание: необходимо послать сообщение `TB_ENABLEBUTTON` панели инструментов, подробности можно найти в справочной системе MSDN. Подобную задачу для пункта меню решает функция `EnableMenuItem()`.
6. Организовать скроллинг для задачи просмотра текстового файла, листинг 2.4. Дополнить задачу панелью инструментов и меню выбора шрифтов.

Глава 3

Окна и элементы управления

Кроме главного окна, приложение может создавать и другие окна: перекрываемые `WS_OVERLAPPED`, всплывающие `WS_POPUP`, дочерние `WS_CHILD`. *Каждое окно должно иметь свою оконную функцию, куда передаются посылаемые операционной системой сообщения.*

Начнем рассмотрение с создания *дочерних окон*, особенностью которых является то, что они располагаются в клиентской области родительского окна и автоматически уничтожаются при его закрытии.

Для создания окна используется функция `CreateWindow()` или ее расширение `CreateWindowEx()`:

```
HWND WINAPI CreateWindowExW(DWORD dwExStyle, LPCWSTR lpClassName,
LPCWSTR lpWindowName, DWORD dwStyle, int X, int Y, int nWidth,
int nHeight, HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
LPVOID lpParam);
```

Функции отличаются лишь первым параметром расширенных стилей `dwExStyle`. Если посмотреть файл включений `winuser.h`, то можно увидеть, что все вызовы функции `CreateWindow()` заменяются ее расширенной версией, где первый параметр имеет нулевое значение.

> ***Примечание***
>
> Параметр `hMenu` используется для задания уникального идентификатора дочернего окна `child_ID`. В этом случае приходится задавать явное преобразование типа `(HMENU)child_ID`.

В *главе 1* мы уже рассматривали значение аргументов этой функции при анализе "скелета" Windows-приложения. Вспомним последовательность шагов при создании окна:

1. Создается переменная типа "Класс окна" `WNDCLASSEX`.
2. Регистрируется класс окна функцией `RegisterClassEx()`.
3. Вызывается функция `CreateWindowEx()` для создания окна.
4. При помощи функции `ShowWindow()` окно отображается.
5. Поскольку отображение окна происходит асинхронно, то обычно используют функцию `UpdateWindow()` для принудительной прорисовки окна.

Примечание

Сейчас, как правило, используют расширенный класс окна `WNDCLASSEX` вместо `WNDCLASS` и, соответственно, расширенную функцию регистрации класса окна `RegisterClassEx()`.

Рассмотрим процесс создания дочерних окон на примере реализации нескольких учебных задач.

Дочерние окна

Построим приложение, имитирующее широко известную игру "крестики-нолики", где вторым партнером будет выступать компьютер (рис. 3.1). В качестве заготовки используем стандартный Win32-проект. Добавим один пункт меню **New** для запуска новой игры. Текст оконной функции приложения приведен в листинге 3.1.

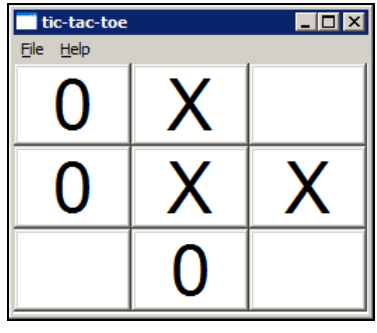


Рис. 3.1. Крестики-нолики

Основная идея, которая реализуется в этой программе, заключается в том, что вместо рисования прямоугольников и вычисления затем позиции нажатия левой кнопки мыши, мы создадим 9 дочерних окон и будем обрабатывать сообщение о щелчке мышью в соответствующем окне.

Листинг 3.1. Крестики-нолики

```
LRESULT CALLBACK ChildProc(HWND, UINT, WPARAM, LPARAM);
TCHAR ChildClassName[MAX_LOADSTRING] = _T("WinChild");
ATOM MyRegisterChildClass()
{
   WNDCLASSEX wcex    = { 0 };
   wcex.cbSize        = sizeof(WNDCLASSEX);
   wcex.lpfnWndProc   = ChildProc;
   wcex.hInstance     = hInst;
   wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
```

```
   wcex.lpszClassName = ChildClassName;
   return RegisterClassEx(&wcex);
}
static HFONT newFont;
static HWND hChild[9];
unsigned char k[9] = { 0 };
char text[] = { ' ', 'X', '0' };
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   int i;
   static int sx, sy;
   switch (message)
   {
   case WM_CREATE:
      MyRegisterChildClass();
      for (i = 0; i < 9; i++)
         hChild[i] = CreateWindow(ChildClassName, NULL, WS_CHILD |
         WS_DLGFRAME | WS_VISIBLE, 0, 0, 0, 0, hWnd, NULL, hInst, NULL);
      break;
   case WM_SIZE:
      if (wParam == SIZE_MINIMIZED) break; //Кнопка свертывания окна
         sx = LOWORD(lParam)/3;     //Ширина дочернего окна
         sy = HIWORD(lParam)/3;     //Высота дочернего окна
         if (newFont) DeleteObject(newFont);
         newFont = CreateFont(min(sx,sy), 0, 0, 0, FW_NORMAL, 0, 0, 0,
         DEFAULT_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
         DEFAULT_QUALITY, DEFAULT_PITCH | FF_DONTCARE, _T("Arial"));
         for (i = 0; i < 9; i++)
            {  MoveWindow(hChild[i],(i%3)*sx, (i/3)*sy, sx, sy, TRUE);
               UpdateWindow(hChild[i]);
            }
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_NEW:
         for (i = 0; i < 9; i++)
         {
            k[i] = 0;
            InvalidateRect(hChild[i], NULL, 1);
            UpdateWindow(hChild[i]);
         }
```

```
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
////////////////////////////////////////////////////////////////////
LRESULT CALLBACK ChildProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   RECT rt;
   int i, s;
   char *ch;
   switch (message)
   {
   case WM_LBUTTONDOWN :
         for (i = 0; hWnd != hChild[i]; i++);
         if (k[i]) break; else k[i] = 1;
         InvalidateRect(hWnd, NULL, 1);
         UpdateWindow(hWnd);
         srand(lParam);
         for(i = s = 0; i< 9; i++) if (k[i]) s++;
         if(s == 9) MessageBox(hWnd, _T("Для следующего сеанса выбирайте\
         New"), _T("Конец игры"), MB_OK | MB_ICONQUESTION);
         else
         {
            while(true)
            {  s = rand()*9/(RAND_MAX+1);
               if (k[s]) continue;
               k[s] = 2;
               InvalidateRect(hChild[s], NULL, 1);
               UpdateWindow(hChild[s]);
               break;
            }
         }
         break;
      case WM_PAINT:
```

```
        for (i = 0; hWnd != hChild[i]; i++);
        if(k[i])
        {
           ch = text+k[i];
           hdc = BeginPaint(hWnd, &ps);
           GetClientRect(hWnd, &rt);
           SelectObject(hdc, newFont);
           DrawTextA(hdc,ch,1,&rt,DT_SINGLELINE|DT_CENTER|DT_VCENTER);
           EndPaint(hWnd, &ps);
        } //Фоновая закраска окна
        else DefWindowProc(hWnd, message, wParam, lParam);
        break;
     default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При создании главного окна приложения в сообщении `WM_CREATE` создадим 9 дочерних окон. Для всех дочерних окон будем использовать один класс окна и, соответственно, одну оконную функцию `ChildProc`. В локальной функции `MyRegisterChildClass()` определим класс окна. Поскольку многие поля переменной нужно оставить со значением по умолчанию, обнулим всю структуру при описании и заполним лишь необходимые поля. После чего зарегистрируем класс окна с именем `"WinChild"`.

Так как при обработке сообщения `WM_CREATE` главное окно приложения еще не прорисовано, дочерние окна создаем нулевого размера. Укажем стиль `WS_VISIBLE` для того, чтобы окно было отображаемым. Стиль рамки `WS_DLGFRAME` запретит изменение размеров окна при буксировке его границ.

Определим на глобальном уровне массив типа `HWND` дескрипторов дочерних окон `hChild[9]`, дескриптор шрифта `newFont`, а также статический массив `k[9]` типа `unsigned char`, где мы будем хранить признак заполнения окна:

- 0 — окно не занято;
- 1 — пользователь щелкнул левой кнопкой мыши по этому окну, `'X'`;
- 2 — компьютер выбрал это окно для вывода, `'0'`.

Для корректировки размера дочерних окон используем обработчик сообщения `WM_SIZE`, поскольку это сообщение генерируется при любом изменении размера главного окна.

Если сообщение связано с кнопкой минимации и `wParam == SIZE_MINIMIZED`, перерисовывать окно нет необходимости, и мы заканчиваем обработку сообщения оператором `break`.

Ширину и высоту главного окна получим из параметра младшего и старшего слова `lParam`. Размеры дочерних окон установим в 1/3 от полученного размера.

Создадим шрифт, определив его размер по размеру созданного окна. Дескриптор шрифта определен на глобальном уровне, он будет доступен и в оконной функции дочернего окна. Однако при изменении размера окна мы создаем новый шрифт, предварительно удаляя текущий, если он уже определен.

В цикле по переменной `i` нарисуем 9 окон функцией `MoveWindow()`:

```
BOOL WINAPI MoveWindow(HWND hWnd, int x, int y, int nWidth, int nHeight, BOOL
bRepaint);
```

Функция переопределяет позицию вывода окна `(x,y)` и его размеры `nWidth`, `nHeight`. Если последний параметр отличен от `0`, генерируется сообщение `WM_PAINT` для перерисовки окна.

Координаты левого верхнего угла окна определяются формулами:

```
x = (i%3)*sx;
y = (i/3)*sy;
```

Действительно, в клиентской области главного окна координату `x` можно вычислить как остаток от деления `i` на 3, умноженный на ширину дочернего окна; а координату `y` как целую часть от деления `i` на 3, умноженную на высоту дочернего окна.

Здесь, однако, необходимо использовать функцию `UpdateWindow()`, иначе окна не успеют перерисоваться.

Для того чтобы иметь возможность запустить новый сеанс игры, добавим в меню программы пункт **New** с идентификатором `ID_NEW`. В обработчике этой команды очистим массив `k` и перерисуем все дочерние окна.

Основная же логика задачи сосредоточена в оконной функции дочерних окон. Мы будем использовать одну функцию для всех 9 окон, а различать их будем по дескрипторам, которые сохранили в массиве `hChild` на глобальном уровне.

При обработке сообщения `WM_LBUTTONDOWN` сначала мы должны определить индекс окна, пославшего сообщение. Это можно сделать, организовав в цикле перебор дескрипторов дочерних окон и сравнив их с дескриптором окна, пославшего сообщение:

```
for (i = 0; hWnd != hChild[i]; i++);
```

По завершении цикла переменная `i` и будет искомым индексом.

Если значение элемента массива `k[i]` отлично от нуля, мы игнорируем эту операцию, прекращая обработку сообщения оператором `break`. Если же окно свободно, присваиваем элементу массива `k[i]` значение `1`.

После этого нужно перерисовать дочернее окно последовательным вызовом функций `InvalidateRect()` и `UpdateWindow()`.

Сейчас компьютер должен выбрать окно, куда он выведет ответный "нолик". Для этого мы используем функцию генерации случайного числа. Однако функция будет генерировать одну и ту же последовательность случайных чисел, что нас вряд ли устроит. Для того чтобы внести некоторое разнообразие в выборе ответа, мы ввели функцию `srand()`, которая задает различные числовые последовательности, но их выбор опять же зависит от ее аргумента, в качестве которого мы взяли `lParam`,

составленный из координат курсора мыши. Так что, щелкая мышью в разных частях окна, мы получим разные последовательности случайных чисел. Прием, конечно, примитивный, но действенный.

Однако предварительно проверим, не все ли окна уже заняты? Просто подсчитаем количество ненулевых значений массива `k` и, если это количество равно 9, выведем окно сообщения, что игра завершена.

В противном случае в бесконечном цикле `while(true) {. . .}` генерируем случайное число в диапазоне от 0 до 8:

```
s = rand()*9/(RAND_MAX+1);
```

Мы учли, что функция `rand()` возвращает число от `0` до `RAND_MAX = 0x7fff`.

Проверяем, не занято ли окно с индексом `s`? Если не занято, присваиваем элементу массива `k[s]` значение `2` и перерисовываем это окно. Если же окно уже занято, то выполняем следующий шаг цикла, и так далее, пока не попадем на свободное окно. А такое окно обязательно существует, поскольку мы проверили, что занятых окон меньше 9.

Теперь осталось реализовать обработку сообщения `WM_PAINT`.

Начнем с определения индекса активного окна. Опять проверим, не является ли окно пустым? В этом случае ничего выводить не нужно, но для очистки окна лучше воспользоваться обработчиком сообщений по умолчанию `DefWindowProc()`, который закрасит окно фоновой кистью.

Если окно не является пустым, определим символ для вывода в окно. Для этого установим указатель `ch` на символ `'X'`, если `k[i] == 1`; или на символ `'0'`, если `k[i] == 2`. Мы специально ввели в массив `text` первый символ пробела, чтобы упростить выражение:

```
ch = text+k[i];
```

Определим дескриптор контекста устройства `BeginPaint()`, вычислим ограничивающий прямоугольник окна функцией `GetClientRect()` и выберем в качестве текущего шрифт `newFont`.

Функцией `DrawTextA()` выведем один символ в центр окна.

Для этого укажем флаги форматирования текста: `DT_SINGLELINE|DT_CENTER|DT_VCENTER`. Функция `EndPaint()` завершает обработку сообщения `WM_PAINT`.

Всплывающие окна

Всплывающие окна могут быть выведены в любом месте рабочего стола и всегда появляются на переднем плане, в отличие от дочерних, которые размещаются в клиентской области родительского окна. Рассмотрим технику создания всплывающих окон на примере приложения, строящего *xy*-график.

Обычно графические построения не производятся в главном окне программы, для этого создают отдельное окно. Поставим задачу: создать программу, которая читает текстовый файл с массивом числовых данных, расположенных в двух столбцах,

и строит стандартный *xy*-график. В качестве основы используем программу чтения текстового файла (см. листинг 2.3). Для упрощения кода перенесем описание вектора `v` и итератора `it` на глобальный уровень.

На глобальном уровне введем имя нового класса окна и опишем прототип оконной функции построения графика:

```
TCHAR WinClassGraphName[MAX_LOADSTRING] = _T("ChildClass");
LRESULT CALLBACK WndGraph(HWND, UINT, WPARAM, LPARAM);
```

Для вывода графика добавим пункт меню **xy-график** с идентификатором `ID_LINE`. Обработчик этой команды разместим в функции главного окна:

```
case ID_LINE :
       if (IsWindow(hGraph)) break;
       RegisterGraphClass();
       hGraph = CreateWindow(WinClassGraphName, _T("xy-график"),
       WS_SYSMENU | WS_POPUP | WS_VISIBLE | WS_THICKFRAME | WS_CAPTION,
       sx/4, sy/4, min(sx, sy), min(sx, sy), hWnd,0, hInst, NULL);
       break;
```

Стили окна обеспечат вывод всплывающего окна `WS_ POPUP`, ограниченного тонкой рамкой `WS_THICKFRAME`, с заголовком `WS_CAPTION` и системным меню `WS_SYSMENU`, отображаемое сразу после создания; стиль `WS_VISIBLE`.

Дескриптор опишем в оконной функции главного окна:

```
static HWND hGraph;
```

Для предотвращения повторного создания окна предусмотрим проверку его существования:

```
if (IsWindow(hGraph)) break;
```

Положение всплывающего окна определяется в системе координат рабочего стола. Выберем такие параметры:

```
sx/4, sy/4, min(sx, sy), min(sx, sy)
```

> ***Примечание***
>
> Для окна графика можно было бы выбрать стиль дочернего окна `WS_CHILD`. В этом случае, однако, необходим дополнительный стиль `WS_EX_NOPARENTNOTIFY`, чтобы предотвратить завершение приложения при закрытии дочернего окна.

Регистрацию *класса окна* выделим в отдельную функцию:

```
ATOM RegisterGraphClass()
{
   WNDCLASSEX wcgraph       = {0};
   wcgraph.cbSize           = sizeof(WNDCLASSEX);
   wcgraph.style            = CS_HREDRAW | CS_VREDRAW;
   wcgraph.lpfnWndProc      = WndGraph;
   wcgraph.hInstance        = hInst;
   wcgraph.hCursor          = LoadCursor(NULL, IDC_CROSS);
   wcgraph.hbrBackground    = (HBRUSH) (COLOR_WINDOW+1);
```

```
    wcgraph.lpszClassName     = WinClassGraphName;
    wcgraph.hIconSm           = LoadIcon(hInst,MAKEINTRESOURCE(IDI_ICON1));
    return RegisterClassEx(&wcgraph);
}
```

Примечание

Мы сделали присваивание `wcgraph = {0}`, чтобы не заполнять нулевые поля структуры `WNDCLASSEX`.

Для графического окна используем курсор `IDC_CROSS` и новую пиктограмму, которую мы импортировали в наш проект. Это можно сделать в контекстном меню **Add Resource…** окна **Resource View** (рис. 3.2).

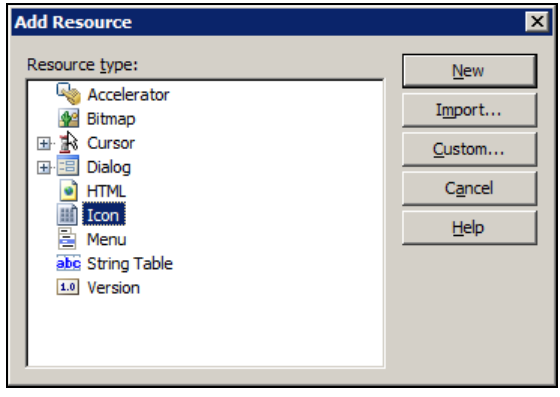


Рис. 3.2. Диалоговое окно **Add Resource**

По умолчанию новой иконке присваивается идентификатор `IDI_ICON1`.

Рассмотрим оконную функцию всплывающего окна (листинг 3.2).

Листинг 3.2. Оконная функция построения *xy*-графика

```
const int scaleX = 8;         //Метки по оси x
const int scaleY = 4;         //Метки по оси y
const int indent = 25;        //Отступ для вывода меток оси x
struct DOUDLE_POINT { double x, y; };
const int GRAPHSIZE = 1200;
const int GRAPHWIDTH = 1000;
LRESULT CALLBACK WndGraph(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static HPEN hline;
```

```
static HBRUSH hrect;
RECT rt;
static int n, sx, sy, kx, ky;
static double max_x, max_y, min_x, min_y;
int i, x, y;
static POINT *pt;
TCHAR s[20];
DOUDLE_POINT t;
double z, hx, hy;
static DOUDLE_POINT *xy;
switch (message)
{  case WM_CREATE:
      if((n = v.size()) == 0)
      {
           MessageBox(hWnd, _T("Загрузите файл"), _T("Нет данных"),
                      MB_OK | MB_ICONHAND);
           DestroyWindow(hWnd);
           return 1;
      }
      pt = new POINT[n];
      xy = new DOUDLE_POINT[n];
      for (it = v.begin(), i = 0; i < n; i++, it++)
      {
        if(sscanf(it->c_str(),"%lf %lf",&t.x, &t.y) != 2)
        {
           MessageBoxA(hWnd, it->c_str(), "Ошибка данных", MB_OK|
                       MB_ICONHAND);
           DestroyWindow(hWnd);
           return 1;
        }
        xy[i] = t;
      }
      max_x = min_x = xy[0].x;
      max_y = min_y = xy[0].y;
      for (i = 1; i < n; i++)
      {
        if (max_x < xy[i].x) max_x = xy[i].x;
           else if (min_x > xy[i].x) min_x = xy[i].x;
        if (max_y < xy[i].y) max_y = xy[i].y;
           else if (min_y > xy[i].y) min_y = xy[i].y;
      }
      hline = CreatePen(PS_SOLID, 6, RGB(0, 0, 255));
      hrect = CreateSolidBrush(RGB(255,0,0));
```

```
      hx = max_x - min_x;
      hy = max_y - min_y;
      for (i = 0; i < n; i++)
      {
        pt[i].x = int((xy[i].x - min_x)*GRAPHWIDTH/hx + 0.5);
        pt[i].y = int((xy[i].y - min_y)*GRAPHWIDTH/hy + 0.5);
      }
      break;
  case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      break;
    case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      hx = (max_x - min_x)/scaleX;
      hy = (max_y - min_y)/scaleY;
      SetMapMode(hdc, MM_ANISOTROPIC);
      SetWindowExtEx(hdc, GRAPHSIZE, -GRAPHSIZE, NULL);
      SetViewportExtEx(hdc, sx, sy, NULL);
      SetViewportOrgEx(hdc, 2*indent, sy-indent, NULL);
      SetTextAlign(hdc, TA_RIGHT | TA_TOP);
      for (z = min_x, i = 0; i <= scaleX; z += hx, i++)
      {
        x = int((z - min_x)*GRAPHWIDTH/(max_x - min_x) + 0.5);
        _stprintf(s, _T("%.1lf"), z);
        TextOut(hdc, x, 0, s, _tcslen(s));
        MoveToEx(hdc, x, -10, NULL);
        LineTo(hdc, x, 10);
      }
      MoveToEx(hdc, 0, 0, NULL);
      LineTo(hdc, GRAPHWIDTH, 0);
      SetTextAlign(hdc, TA_RIGHT | TA_BOTTOM);
      for (z = min_y, i = 0; i <= scaleY; z += hy, i++)
      {
        y = int((z - min_y)*GRAPHWIDTH/(max_y - min_y) + 0.5);
        _stprintf(s, _T("%.1lf"), z);
        TextOut(hdc, 0, y, s, _tcslen(s));
        MoveToEx(hdc, -10, y, NULL);
        LineTo(hdc, 10, y);
      }
      MoveToEx(hdc, 0, 0, NULL);
      LineTo(hdc, 0, GRAPHWIDTH);
      SelectObject(hdc, hline);
```

```
            Polyline(hdc, pt, n);
            for ( i = 0; i < n; i++)
              {
                 SetRect(&rt, pt[i].x-8, pt[i].y-8, pt[i].x+8, pt[i].y+8);
                 FillRect(hdc, &rt, hrect);
              }
            EndPaint(hWnd, &ps);
            break;
         case WM_DESTROY:
            DeleteObject(hline);
            DeleteObject(hrect);
            delete[] pt;
            delete[] xy;
            break;
         default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

Создадим текстовый файл с данными и построим график, изображенный на рис. 3.3. Предполагается, что точки графика упорядочены по возрастанию координаты *x*.

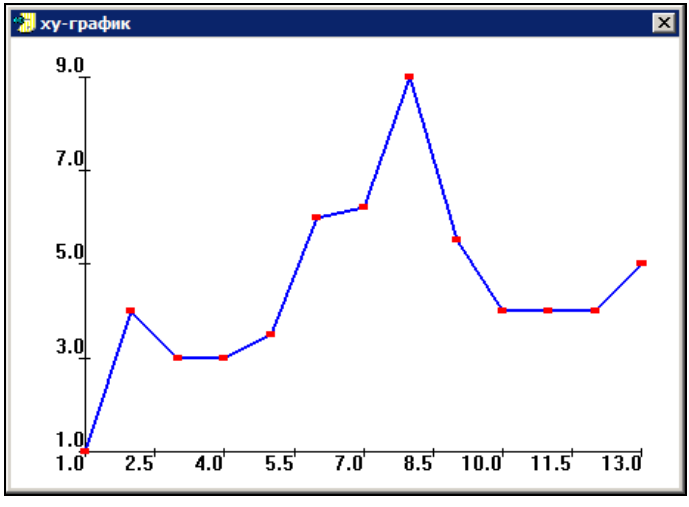


Рис. 3.3. Построение *xy*-графика

Константы, которые мы описали перед телом функции, служат для задания количества меток по осям координат, а также для выделения места для подписей осей. Мы выделили 25 логических единиц для подписи оси *x* и 50 единиц для оси *y*. Две константы `GRAPHSIZE` и `GRAPHWIDTH` определяют логические размеры окна и графика.

Начнем рассмотрение кода с сообщения `WM_CREATE`. Вначале проверим — загружены ли данные в память?

```
if((n = v.size()) == 0) . . .
```

Если данные не загружены и `n` равно `0`, выводим сообщение в диалоговом окне и закрываем окно `hGraph`.

> ***Примечание***
>
> Можно было бы решить эту проблему иначе. При создании окна сделать пункт меню построения графика **xy-graph** неактивным:
> `EnableMenuItem(GetMenu(hWnd),ID_LINE, MF_DISABLED);`
> После загрузки файла вернуть активность пункту меню:
> `EnableMenuItem(GetMenu(hWnd),ID_LINE, MF_ENABLED);`

Если же данные имеются, выделяем память под массив типа `DOUBLE_POINT`, где будем хранить данные графика в виде чисел с плавающей точкой типа `double`, и массив `POINT`, где координаты точек хранятся в логических единицах. Дело в том, что функция `Polyline()`, которую мы будем использовать для вывода графика, требует аргумента типа `POINT*`.

После чего извлекаем числовые данные из контейнера, где они хранятся в виде строк типа `string`.

Поскольку число строк текста нам известно, читаем их в цикле, и функцией `sscanf()` извлекаем по два числа из строки. Одна из причин, почему мы воспользовались этой функцией, заключается в том, что она возвращает количество преобразованных данных. Если же в строке оказалось менее двух чисел, выведем сообщение об ошибке в данных и закроем окно `hGraph`.

Здесь же определим диапазон для *x* и *y*-координат графика, т. е. минимальное и максимальное значения. Создадим перо для построения графика и кисть для выделения точек. После чего заполним массив логических координат точек графика.

При обработке сообщения `WM_SIZE` найдем размеры дочернего окна `sx` и `sy` (размер окна может быть изменен пользователем, и график должен быть перестроен).

Заранее предполагаем, что график будет строиться в логическом прямоугольнике размером 1000×1000. Размер окна определим 1200×1200, начало координат поместим в левом нижнем углу окна, выполнив смещение по оси *x* на 50, а по оси *y* на 25 единиц. Ось *x* направлена вправо, а ось *y* — вверх.

Формула преобразования:

$$x^i_{\text{лог}} = \left(\frac{x_i - x_{\min}}{x_{\max} - x_{\min}} \right) \cdot 1000; \qquad y^i_{\text{лог}} = \left(\frac{y_i - y_{\min}}{y_{\max} - y_{\min}} \right) \cdot 1000.$$

Для округления до целого добавим `0.5` и укажем явное преобразование типа.

Вывод графика осуществляется при обработке сообщения `WM_PAINT`.

Установим логическую систему координат 1200×1200 на всю клиентскую часть окна:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, GRAPHSIZE, -GRAPHSIZE, NULL);
SetViewportExtEx(hdc, sx, sy, NULL);
```

Ось *x* — вправо, а *y* — вверх. Начало координат определим относительно исходной системы координат (см. рис. 3.3).

```
SetViewportOrgEx(hdc, 2*indent, sy-indent, NULL);
```

Выведем подписи оси *x*, т. е. 8 числовых меток (как определено в константе `scaleX`), от минимального до максимального значения *x*-координаты.

При выводе горизонтальных меток хотелось бы, чтобы числовое значение своим правым краем стояло на позиции метки, но по умолчанию текст выравнивается влево. Решить эту проблему можно, установив выравнивание текста от правой верхней точки при помощи функции `SetTextAlign()`.

Теперь весь вывод будет осуществляться, исходя из заданных параметров `align`:

`TA_RIGHT | TA_TOP`.

Для преобразования числового значения в текстовый вид здесь так же можно воспользоваться функцией `sprintf()`. Зададим формат вывода `"%.1lf"`, что означает 1 знак после десятичной точки для числа с плавающей точкой типа `double (lf)`, размер поля вывода по фактической длине строки.

После вывода очередного числа ставим вертикальную риску длиной 20 логических единиц, а линию оси выведем после завершения цикла.

Подписи оси *y* сделаем аналогично, только для вывода числовых значений установим выравнивание по нижнему правому краю: `TA_RIGHT | TA_BOTTOM`.

Выбираем синее перо `hline` и строим ломаную линию функцией `Polyline()`.

После чего в цикле "заливаем" красной кистью `hrect` все точки квадратами со стороной в 16 единиц.

Нам осталось лишь освободить занятые ресурсы: при обработке сообщения `WM_DESTROY` — удалить перо, кисть и освободить память, занятую массивами `pt` и `xy`.

Диалоговые окна

Диалоговое окно является специальным типом окна, предназначенным для организации взаимодействия пользователя с программой. Диалоговое окно чаще всего используют как контейнер для элементов управления, оно предоставляет пользователю возможность выбирать или вводить данные. Начнем рассмотрение с *модальных диалоговых окон*. Модальное диалоговое окно при создании получает управление и не может возвратить его обратно вызывающему окну до завершения работы. Создается диалоговое окно при помощи вызова функции `DialogBox()`:

```
INT_PTR WINAPI DialogBoxW (HINSTANCE hInstance, LPCWSTR lpTemplate,
                           HWND hWndParent, DLGPROC lpDialogFunc);
```

- `hInstance` — дескриптор текущего приложения;
- `lpTemplate` — указатель ресурса диалогового окна;

- `hWndParent` — дескриптор "родительского" окна, породившего диалоговое окно;
- `lpDialogFunc` — указатель на функцию диалогового окна.

Диалоговое окно использует файл ресурсов, куда при помощи редактора ресурсов помещаются необходимые элементы.

> ***Примечание***
>
> На самом деле `DialogBoxW()` представляет собой макрос, который преобразуется в функцию `DialogBoxParamW()`, имеющую дополнительно параметр `dwInitParam`, равный 0 по умолчанию.

Для создания диалогового окна перейдем на подложку обозревателя ресурсов **Recourse View** и в контекстногом меню выберем **Insert Dialog** (рис. 3.4).

Окно мастера создания нового диалога показано на рис. 3.5, где по умолчанию размещены две стандартные кнопки **OK** и **Cancel**.

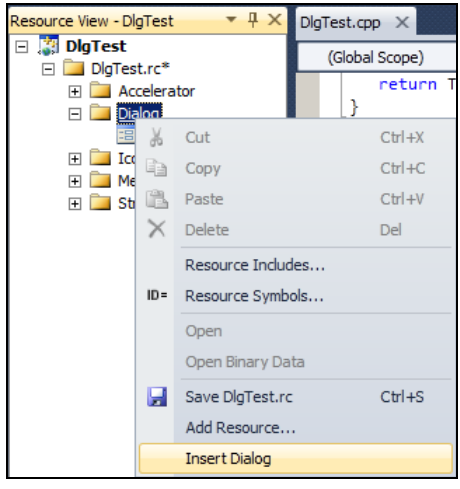


Рис. 3.4. Создание диалогового окна

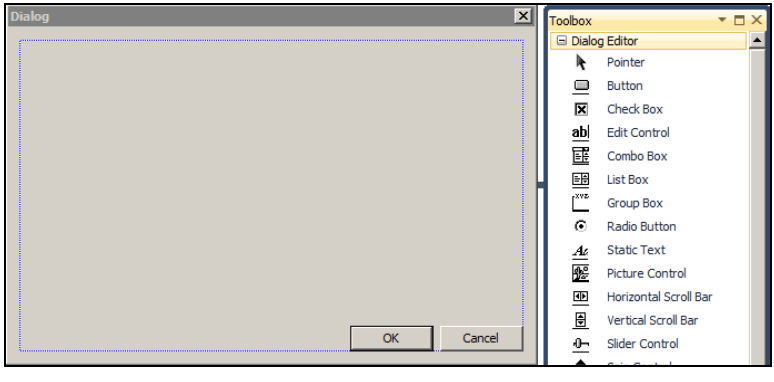


Рис. 3.5. Вид диалогового окна с панелью инструментов

Операционная система Windows имеет множество элементов управления, которые, по сути, являются окнами, зарегистрированными в системе. Это *кнопки* (**Button**), *флажоки/переключатели управления* (**Check Box**), *переключатели* (**Radio Button**), *списки* (**List Box**), *комбинированные списки с полем ввода* (**Combo Box**), *поля ввода* (**Edit Control**), *полосы прокрутки* (**Scroll Bar**), *статические элементы (надписи)* (**Static Text**) и др.

Элементы управления вставляются в диалоговое окно при помощи панели инструментов, хотя их можно вставить в приложение динамически, создавая окна элементов управления.

> ***Примечание***
>
> Каждый элемент управления имеет оконную функцию, принадлежащую операционной системе, которая используется совместно всеми приложениями. Управление элементами осуществляется посылкой их окну соответствующего сообщения. Данные передаются в параметрах сообщения, а результат возвращается функцией `SendMessage()`.

Мы можем размещать элементы на поверхности диалогового окна буксировкой при нажатой левой кнопке мыши или выделением элемента на палитре и указанием прямоугольника для его размещения. После создания элемента управления можно изменить его размеры и другие свойства.

Рассмотрим пример создания диалогового окна, на котором разместим несколько наиболее часто используемых элементов управления.

Тестирование элементов управления

Создадим тестовую программу для следующих элементов управления: *кнопка, флажок, переключатель, список, комбинированный список (список с полем ввода), полоса прокрутки, статический элемент (надпись)*. В диалоговом окне мы будем изменять их состояние, а в главном окне выведем полученные значения. В качестве основы используем стандартную заготовку, а диалоговое окно, изображенное на рис. 3.6, построим в редакторе ресурсов. По умолчанию его идентификатор `IDD_DIALOG1`.

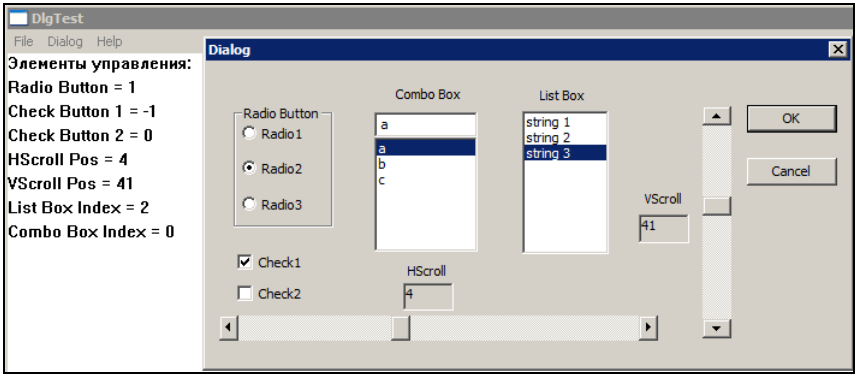


Рис. 3.6. Тест элементов управления

Каждому из элементов управления, размещенному на поверхности диалогового окна, за исключением статических полей, по умолчанию присваиваются уникальные идентификаторы, которые мы менять не будем. Заменим лишь идентификаторы статических полей, используемых для индикации состояния полос прокрутки (линеек скроллинга): `IDC_HSCR`, `IDC_VSCR`.

> ***Примечание***
>
> Для всех элементов управления установим свойство `Tabstop = true`, которое определит возможность перехода между элементами управления клавишей <Tab>. Исходный порядок соответствует последовательности размещения элементов на диалоговом окне. Для изменения порядка перехода воспользуйтесь меню **Format | Tab Order**. Щелкните мышью по элементам в необходимой последовательности и нажмите клавишу <Enter>.

Текст оконной функции приложения и функции диалогового окна, управляющей этими элементами, приведен в листинге 3.3.

Листинг 3.3. Тест элементов управления

```
INT_PTR CALLBACK Dialog1(HWND, UINT, WPARAM, LPARAM);
static int radio, check1, check2, scrlh, scrlv, lIndex, cIndex;
int *val[] = {&radio,&check1,&check2,&scrlh,&scrlv,&lIndex,&cIndex};
TCHAR *combo[100] = { _T("a"), _T("b"), _T("c") };
TCHAR *list[100] = { _T("string 1"), _T("string 2"), _T("string 3") };
TCHAR *ctrl = _T("Элементы управления:");
TCHAR *str_control[] = {_T("Radio Button"),_T("Check Button 1"),
              _T("Check Button 2"),_T("HScroll Pos"),_T("VScroll Pos"),
              _T("List Box Index"),_T("Combo Box Index")};
const int HNUM = 10, VNUM = 100;
const int List_size = 3, Combo_size = 3;
const int INTERVAL = 20;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   TCHAR str[256];
   int i;
   switch (message)
   {
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_STDDIALOG :
         DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, Dialog1);
```

```
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      TextOut(hdc, 0, 0, ctrl, _tcslen(ctrl));
      for (i = 0; i < 7;)
      {
         _stprintf(str, _T("%s = %d"), str_control[i], *val[i]);
         TextOut(hdc, 0, ++i*INTERVAL, str, _tcslen(str));
      }
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
//////////////////////////////////////////////////////////////////
INT_PTR CALLBACK Dialog1(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
   static int radio, check1, check2, scrlh, scrlv;
   static HWND hScroll, vScroll, hWndList, hWndComboBox;
   int i;
   switch (message)
   {
      case WM_INITDIALOG:
         radio = ::radio;
         CheckRadioButton(hDlg,IDC_RADIO1,IDC_RADIO3,IDC_RADIO1+radio);
         check1 = ::check1;
         SendDlgItemMessage(hDlg, IDC_CHECK1, BM_SETCHECK, check1, 0);
         check2 = ::check2;
         SendDlgItemMessage(hDlg, IDC_CHECK2, BM_SETCHECK, check2, 0);
         scrlh = ::scrlh;
         SetDlgItemInt(hDlg, IDC_HSCR, scrlh, 0);
         hScroll = GetDlgItem(hDlg, IDC_SCROLLBAR1);
         SetScrollRange(hScroll, SB_CTL, 0, HNUM, FALSE);
         SetScrollPos(hScroll, SB_CTL, scrlh, TRUE);
         scrlv = ::scrlv;
```

```
      SetDlgItemInt(hDlg, IDC_VSCR, scrlv, 0);
      vScroll = GetDlgItem(hDlg, IDC_SCROLLBAR2);
      SetScrollRange(vScroll, SB_CTL, 0, VNUM, FALSE);
      SetScrollPos(vScroll, SB_CTL, scrlv, TRUE);
      hWndList = GetDlgItem(hDlg, IDC_LIST1);
      for (i = 0; i < List_size; i++)
         SendMessage(hWndList, LB_ADDSTRING, 0, (LPARAM)list[i]);
      SendMessage(hWndList, LB_SETCURSEL, lIndex, 0);
      hWndComboBox = GetDlgItem(hDlg, IDC_COMBO1);
      for (i = 0; i < Combo_size; i++)
         SendMessage(hWndComboBox, CB_ADDSTRING, 0, (LPARAM)combo[i]);
      SendMessage(hWndComboBox, CB_SETCURSEL, cIndex, 0);
      return TRUE;
   case WM_COMMAND:
   switch(LOWORD(wParam))
   {
   case IDOK: lIndex = SendMessage(hWndList, LB_GETCURSEL, 0, 0);
              cIndex = SendMessage(hWndComboBox, CB_GETCURSEL,0,0);
              ::radio = radio;
              ::check1 = check1;
              ::check2 = check2;
              ::scrlh = scrlh;
              ::scrlv = scrlv;
              InvalidateRect(GetParent(hDlg), NULL, 1);
   case IDCANCEL : return EndDialog(hDlg, 0);
   case IDC_CHECK1 : check1 = ~check1;
         SendDlgItemMessage(hDlg, IDC_CHECK1, BM_SETCHECK, check1, 0);
         return TRUE;
   case IDC_CHECK2 : check2 = ~check2;
         SendDlgItemMessage(hDlg, IDC_CHECK2, BM_SETCHECK, check2, 0);
         return TRUE;
   case IDC_RADIO1 : radio = 0; break;
   case IDC_RADIO2 : radio = 1; break;
   case IDC_RADIO3 : radio = 2; break;
   }
      CheckRadioButton(hDlg,IDC_RADIO1,IDC_RADIO3,IDC_RADIO1+radio);
      return TRUE;
   case WM_HSCROLL:
         switch(LOWORD(wParam))
         {
         case SB_LINELEFT        : scrlh--; break;
         case SB_LINERIGHT: scrlh++; break;
         case SB_PAGELEFT        : scrlh -= HNUM/2; break;
```

```
            case SB_PAGERIGHT: scrlh += HNUM/2; break;
            case SB_THUMBPOSITION    : scrlh = HIWORD(wParam); break;
            }
            scrlh = max(0, min(scrlh, HNUM));
            if (scrlh != GetScrollPos(hScroll, SB_CTL))
            {
               SetScrollPos(hScroll, SB_CTL, scrlh, TRUE);
               SetDlgItemInt(hDlg, IDC_HSCR, scrlh, 0);
            }
            return TRUE;
      case WM_VSCROLL:
            switch(LOWORD(wParam))
            {
            case SB_LINEUP   : scrlv--; break;
            case SB_LINEDOWN : scrlv++; break;
            case SB_PAGEUP   : scrlv -= VNUM/10; break;
            case SB_PAGEDOWN : scrlv += VNUM/10; break;
            case SB_THUMBPOSITION : scrlv = HIWORD(wParam);break;
            }
            scrlv = max(0, min(scrlv, VNUM));
            if (scrlv != GetScrollPos(vScroll, SB_CTL))
            {
               SetScrollPos(vScroll, SB_CTL, scrlv, TRUE);
               SetDlgItemInt(hDlg, IDC_VSCR, scrlv, 0);
            }
            return TRUE;
      default: return FALSE;
   }
    return FALSE;
}
```

На глобальном уровне опишем переменные, которые будут отражать состояние элементов управления:

```
static int radio, check1, check2, scrlh, scrlv, lIndex, cIndex;
```

Поскольку принято, что в группе переключателей (Radio Button) может выбираться лишь один переключатель, то для описания всей группы достаточно одной переменной `radio` целого типа, значения которой `{0,1,2}` мы понимаем как включение первой, второй или третьей кнопки соответственно.

> ***Примечание***
>
> Вообще-то нет никаких проблем для включения нескольких переключателей одновременно при помощи функции `CheckDlgButton()` или же просто посылкой сообщения `BST_CHECKED` для включения и `BST_UNCHECKED` для выключения, однако лучше придерживаться стандартных соглашений.

Флажки (Check Box) могут переключаться независимо друг от друга, поэтому для их описания определим две переменные `check1` и `check2`. Признаком включения является *ненулевое* значение переменной.

Переменные `scrlh` и `scrlv` отражают позицию горизонтальной и вертикальной полосы прокрутки Scroll Bar.

> ***Примечание***
>
> Здесь полосы прокрутки являются элементами управления, в отличие от рассмотренных ранее стилей, и могут располагаться в любом месте окна.

И, наконец, переменные `lIndex`, `cIndex` необходимы для сохранения индекса выбранного элемента списка и комбинированного списка.

Эти переменные мы объявили как `static` лишь для того, чтобы они имели нулевое начальное значение.

Для начального заполнения списков мы описали два массива указателей `list[100]` и `combo[100]` типа `TCHAR*`, которым присвоили лишь 3 первых значения.

В функции главного окна `WndProc()` необходимо добавить обработчик пункта меню **StdDialog** с идентификатором `ID_STDDIALOG`:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, Dialog1);
```

Здесь будет сделан вызов диалогового окна `IDD_DIALOG1`.

Вывод значений параметров сделаем в сообщении `WM_PAINT` главного окна, где выведем в цикле значение переменных, отслеживающих состояние элементов управления.

Для преобразования числового значения в строку проще всего воспользоваться функцией форматного вывода `_stprintf()`. Строка формата позволит "склеить" текст и числовое значение: `"%s=%d"`. Выводим строки с интервалом в 20 логических единиц.

Вся логика обработки осуществляется в функции диалогового окна `Dialog1()`, рассмотрим подробнее ее работу.

Грамотно оформленное диалоговое окно должно иметь возможность закрыться как кнопкой **OK**, так и **Cancel**, поэтому нам нужен второй комплект локальных переменных для описания состояния элементов управления.

```
static int radio, check1, check2, scrlh, scrlv;
```

Для лучшей читаемости кода мы сохранили те же имена, помня, что локальная переменная всегда имеет приоритет перед глобальной. Для обращения к глобальной переменной используем явное указание области видимости `"::"`.

При открытии диалогового окна всегда генерируется сообщение WM_INITDIALOG, где мы и присвоим начальное значение переменных, например:

```
radio = ::radio;
```

и произведем начальную установку элементов управления:

- Переключатели

  Устанавливаем выбранную кнопку функцией CheckRadioButton():

  ```
  BOOL WINAPI CheckRadioButton(HWND hDlg, int nIDFirstButton,
  int nIDLastButton, int nIDCheckButton);
  ```

  Функция принимает дескриптор диалогового окна hDlg, которое является родительским для всех элементов управления, диапазон идентификаторов переключателей (IDC_RADIO1,IDC_RADIO3), и идентификатор активной кнопки IDC_RADIO1+radio. Для корректной работы данной функции необходимо создавать кнопки непрерывно одну за другой, чтобы их идентификаторы, которые эквивалентны целым числам, представляли собой последовательный ряд чисел. Должно быть понятно, что при значении переменной radio: 0,1,2; выражение IDC_RADIO1+radio будет эквивалентно одному из трех идентификаторов: IDC_RADIO1, IDC_RADIO2, IDC_RADIO3.

  Примечание

  Вставим селекторные кнопки в рамку элемента управления Group Box, который будет выполнять чисто декоративную функцию.

- Флажки

  Установим флажки, передавая им сообщение BM_SETCHECK функцией SendDlgItemMessage():

  ```
  LRESULT WINAPI SendDlgItemMessageW(HWND hDlg, int nIDDlgItem,
  UINT Msg, WPARAM wParam, LPARAM lParam);
  ```

  Функция принимает дескриптор окна диалога hDlg, идентификатор кнопки и сообщение BM_SETCHECK. Отличное от нуля значение wParam означает установку флажка кнопки, а нулевое значение снимает флажок. lParam здесь не используется.

- Полоса прокрутки

  Для отображения состояния горизонтальной и вертикальной полосы прокрутки мы вставили два статических элемента с идентификаторами IDC_HSCR и IDC_VSCR. Отобразим в них значение переменных scrlh и scrlv функцией SetDlgItemInt():

  ```
  BOOL WINAPI SetDlgItemInt(HWND hDlg, int nIDDlgItem, UINT uValue,
  BOOL bSigned);
  ```

  Функция принимает: hDlg — дескриптор окна, nIDDlgItem — идентификатор элемента управления, uValue — переменную целого типа, bSigned — признак вывода числа со знаком.

  Нужно получить дескриптор полосы прокрутки функцией GetDlgItem():

  ```
  HWND WINAPI GetDlgItem(HWND hDlg, int nIDDlgItem);
  ```

Это специализированная функция для диалогового окна, которая позволяет по идентификатору элемента управления `nIDDlgItem` получить его дескриптор.

Теперь нужно задать диапазон для горизонтальной полосы прокрутки. Это делается при помощи функции `SetScrollRange()`:

```
BOOL WINAPI SetScrollRange(HWND hWnd, int nBar, int nMinPos,
int nMaxPos, BOOL bRedraw);
```

Первым параметром указываем дескриптор полосы прокрутки. Второй параметр имеет значение `SB_CTL`, что идентифицирует полосу прокрутки как элемент управления, а не свойство окна, где он принимал значение `SB_HORZ` и `SB_VERT`. Следующие два параметра задают `min` и `max` значение позиции полос прокрутки (мы установим диапазон `[0,HNUM]`, переменная `HNUM` описана в заголовке программы), а последний параметр служит признаком перерисовки.

Затем задается текущая позиция движка функцией `SetScrollPos()`:

```
int WINAPI SetScrollPos(HWND hWnd,int nBar,int nPos,BOOL bRedraw);
```

Первые два параметра имеют тот же смысл, что и в предыдущей функции, 3-й параметр — позиция движка, а последний параметр — признак перерисовки.

Параметры вертикальной полосы прокрутки устанавливаем аналогично.

- Список

Для первичного заполнения списка мы описали на глобальном уровне массив указателей `TCHAR *list[]`, где значение присвоено только 3-м первым элементам.

Получим дескриптор списка `hWndList` и в цикле заполним список, передавая сообщение `LB_ADDSTRING`, где указатель на строку помещаем в `lParam` с явным преобразованием типа, иначе компилятор не сможет построить код этого выражения. Константа `List_size` описана на глобальном уровне и равна 3.

Для отображения выделенного элемента посылаем списку сообщение `LB_SETCURSEL`, передавая в `wParam` значение индекса `lIndex`.

- Комбинированный список

Опишем на глобальном уровне массив указателей `TCHAR *combo[]` для начальной инициализации комбинированного списка.

Далее все делается так же, как у списка, только префикс сообщений вместо `LB` будет `CB`.

> ***Примечание***
>
> Если для списка или комбинированного списка установлен стиль `Sort`, отображаться элементы будут в отсортированном по умолчанию порядке.

- Статические элементы (надписи)

Обычно статические элементы используют для вывода в окне поясняющих надписей, но в этом случае к ним нет необходимости обращаться, и мы оставим идентификаторы со значением по умолчанию `IDC_STATIC`. Однако двум элементам, которые используются для вывода состояния полос прокрутки, присвоим

уникальные идентификаторы IDC_HSCR и IDC_VSCR. Чтобы выделить эти элементы в окне установим true для стиля Border и Client Edge.

Выводят в окно статический элемент обычно функциями: SetDlgItemText() или SetDlgItemInt(), передавая им в качестве параметра указатель на строку текста или переменную целого типа.

При совершении операций с элементами управления в диалоговую функцию передается сообщение WM_COMMAND, а в LOWORD(wParam) содержится идентификатор этого элемента. Поэтому для обработки сообщений от элементов управления внутри обработчика сообщения WM_COMMAND обычно создают переключатель switch, в котором и выполняют селекцию поступающих сообщений. Причем, если оконная функция диалогового окна обрабатывает сообщение, она дожна возвращать TRUE.

```
case WM_COMMAND:
  switch(LOWORD(wParam))
      {
        ...
      }
  return TRUE;
```

Начнем рассмотрение с кнопок **OK** и **Cancel**. Мастер построения диалогового окна автоматически вставляет эти кнопки, присваивая им стандартные идентификаторы IDOK и IDCANCEL.

> ***Примечание***
>
> Идентификатор IDCANCEL имеет значение, равное 2, такое же сообщение генерируется при нажатии системной кнопки закрытия окна ☒.

При нажатии кнопки **OK** необходимо сохранить на глобальном уровне значение локальных переменных, контролирующих элементы управления, например:

```
::radio = radio;
```

а индексы элементов, выбранных в списке и комбинированном списке, необходимо получить, послав этим элементам сообщения LB_GETCURSEL и CB_GETCURSEL соответственно. Возвращаемое значение фукции SendMessage() вернет искомый индекс. Далее необходимо инициировать перерисовку главного окна вызовом функции InvalidateRect(), где дескриптор родительского окна мы получим как возвращаемое значение функции GetParent():

```
HWND WINAPI GetParent(HWND hWnd);
```

> ***Примечание***
>
> Мы намеренно не использовали оператор break в конце обработчика сообщения от кнопки **OK**. В этом случае будет выполняться код следующего оператора case и произойдет нормальное закрытие диалогового окна.

При нажатии кнопки **Cancel** выполнение кода фрагмента начнется с функции EndDialog():

```
BOOL WINAPI EndDialog(HWND hDlg, int nResult);
```

здесь мы не сохраняем локальные переменные и не перерисовываем окно.

При обработке сообщений от флажка `IDC_CHECK1` и `IDC_CHECK2` инвертируем значение переменной. Поскольку ее начальное значение `0`, то `~0` равно `-1`, а `~(-1)` равно `0`. Таким образом, переменная будет принимать одно из двух значений `0` или `-1`, что интерпретируется как состояние флажка "**выключено**" (сброшен) и "**включено**" (установлен).

Для изменения состояния флажка посылаем ему сообщение `BM_SETCHECK`. Нулевое значение `wParam` сбросит, а ненулевое — установит флажок.

А вот переключатели (селекторные кнопки) нужно обрабатывать вместе, поскольку во включенном состоянии может быть только один. Мы поместили обработку сообщений от этих переключателей в конец блока, чтобы иметь возможность после установки значения переменной `radio` передать управление функции `CheckRadioButton()` для изменения их состояния.

Обработка сообщений от полос прокрутки принципиально отличается от прочих элементов управления тем, что они не генерируют сообщения `WM_COMMAND`, а точно так же, как и полосы прокрутки — свойства окна, генерируют сообщения: `WM_HSCROLL` — для горизонтальной и `WM_VSCROLL` — для вертикальной полосы прокрутки. Обработка этих сообщений будет осуществляться точно так же, как и в листинге 2.3, однако имеется одно принципиальное отличие — в качестве идентификатора полосы прокрутки — элемента управления — используется `SB_CTL`.

Мы будем обрабатывать сообщения от полосы прокрутки: `SB_LINELEFT`, `SB_LINELEFT`, `SB_LINEUP`, `SB_LINEDOWN`, `SB_PAGELEFT`, `SB_PAGEUP`, `SB_PAGERIGHT`, `SB_PAGEDOWN`, `SB_THUMBPOSITION`.

Так, для горизонтальной полосы прокрутки мы изменяем значение переменной `scrlh`, причем, поскольку мы выбрали для `HNUM` значение `10`, то при щелчке на полосе прокрутки сделаем шаг передвижения равным `5 (HNUM/2)`.

После установки нового значения переменной, как и ранее, при помощи стандартной конструкции, позаботимся о том, чтобы не выйти за пределы интервала `[0,HNUM]`. После чего отобразим значение переменной `scrlh` в статическом элементе (надписи) `IDC_HSCR` функцией `SetDlgItemInt()`.

Вертикальный скроллинг организуем аналогично, имея в виду, что переменная `scrlv`, которая его контролирует, изменяется на отрезке `[0,VNUM]`, а шаг здесь равен `10 (VNUM/10)`. Для отображения состояния вертикального скроллинга используется статический элемент с идентификатором `IDC_VSCR`.

Итак, все сообщения от элементов управления обработаны и возвращают `TRUE`, все же остальные сообщения игнорируются и возвращают `FALSE`.

После закрытия диалогового окна главное окно перерисовывается для вывода значений переменных, контролирующих состояние элементов управления. Поскольку мы определили переменные на глобальном уровне, то они доступны и в оконной функции главного окна. Здесь же опишем массив строк с поясняющим текстом.

Чтобы не писать код для вывода каждой переменной, опишем массив указателей и присвоим его элементам адреса этих переменных:

```
int *val[] = {&radio,&check1,&check2,&scrlh,&scrlv,&lIndex,&cIndex};
```

Теперь при обработке сообщения WM_PAINT, после вывода заголовка ctrl можно организовать цикл для вывода значений переменных. При формировании строки вывода мы воспользовались функцией форматного вывода _stprintf(). Выражение ++i*INTERVAL обеспечит увеличение индекса и наращивание *y*-координаты на 20 единиц.

Общие элементы управления

Элементы управления, которые появились лишь при создании ОС Windows 95, получили название *общих элементов управления (Common Controls)*. Они расширяют возможности стандартных элементов управления и придают программам современный вид. Один из этих элементов мы уже рассмотрели в *главе* 2 — это *панель инструментов* (**Tool Bar**). Сейчас рассмотрим еще 3 элемента — *наборный счетчик* (**Spin Control**), *ползунковый регулятор* (**Slider Control**) и *индикатор выполнения* (**Progress Bar Control**).

Техника их использования мало чем отличается от стандартных элементов управления: в редакторе ресурсов перемещаем нужные элементы с палитры инструментов в диалоговое окно, присваиваем идентификаторы и устанавливаем необходимые свойства.

Особенностью же общих элементов управления является то, что они не входят в стандартную библиотеку и для них необходимо добавить файл включений commctrl.h, а также подключить к проекту библиотеку общих элементов управления comctl32.lib, как мы это делали в предыдущей главе (см. рис. 2.3).

К тому же, до использования общих элементов управления приложение должно инициализировать библиотеку функцией InitCommonControls().

> ***Примечание***
>
> Хотя панель инструментов и является общим элементом управления, для ее использования нет необходимости в вызове функции InitCommonControls().

Так же, как и стандартные элементы управления, общие элементы являются специальными элементами, которые посылают родительскому окну сообщение WM_COMMAND или WM_NOTIFY. Управление осуществляется посылкой им соответствующих сообщений. Демонстрационная задача приведена на рис. 3.7 и далее в листинге 3.4.

Создадим диалоговое окно, которое вызывается в пункте меню **StdDialog**, и разместим там элементы управления: наборный счетчик (спин), ползунковый регулятор (ползунок), индикатор выполнения и два статических элемента для вывода значений спина и ползунка. При закрытии окна кнопкой **OK** возвращаем установленные значения элементов управления и выводим в окне, при нажатии на кнопку **Cancel** ничего возвращать не будем.

> ***Примечание***
>
> Все рассмотренные общие элементы управления (Spin Control, Slider Control, Progress Bar Control) имеют внутреннюю память для хранения состояния элемента.

Для обмена данными опишем на глобальном уровне три переменные: spin, track, progress. Вывод в окно осуществим в сообщении WM_PAINT, где сформируем строку

вывода функцией форматного вывода в строку `_stprintf()`. Причем используем универсальную функцию, работающую как с С-строкой, так и со строкой Unicode. Для вывода воспользуемся функцией `DrawText()`, поскольку она позволяет форматировать выводимый текст. Так что мы сможем одним оператором вывести три строки текста, выравнивая данные табуляцией.

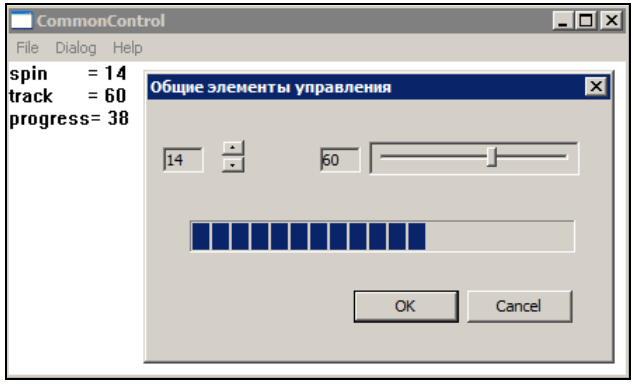


Рис. 3.7. Диалоговое окно с общими элементами управления

Slider (Track Bar Control) — ползунок

Ползунок или *ползунковый регулятор* состоит из указателя, движущегося вдоль шкалы. Он является разновидностью горизонтальной полосы прокрутки и генерирует сообщение `WM_HSCROLL`. В `lParam` передается дескриптор элемента управления, породившего сообщение, и мы можем решить вопрос о том, каким элементом управления оно вызвано, если таких элементов несколько.

В функции диалогового окна опишем переменную для хранения состояния ползунка и его дескриптор:

```
static int track;
static HWND hTrack;
```

В сообщении об инициализации диалога необходимо определить дескриптор, минимальную и максимальную позиции ползунка и установить начальное значение:

```
track = ::track;
hTrack = GetDlgItem(hDlg, IDC_SLIDER1);
SendMessage(hTrack, TBM_SETRANGEMIN, 0, 0);
SendMessage(hTrack, TBM_SETRANGEMAX, 0, 100);
SendMessage(hTrack, TBM_SETPOS, TRUE, track);
```

Здесь минимальное и максимальное значения позиции ползунка определяются посылкой ему сообщения `TBM_SETRANGEMIN` и `TBM_SETRANGEMAX`, а само значение передается в `lParam`, у нас — `[0,100]`. Текущая позиция задается посылкой сообщения `TBM_SETPOS`, где значение `TRUE` в `wParam` означает необходимость перерисовки ползунка, а позиция `track` передается в `lParam`.

Примечание

Вместо двух сообщений для установки минимального и максимального значений ползунка можно использовать сообщение `TBM_SETRANGE`, где минимум задается в младшем слове `lParam`, а максимум — в старшем.

```
SendMessage(hTrack, TBM_SETRANGE, 0, 100<<16);
```

Теперь осталось вывести значение `track` в статическом элементе `IDC_TR1`.

```
SetDlgItemInt(hDlg, IDC_TR1, track, 0);
```

Для манипуляций с ползунком в сообщении `WM_HSCROLL` передадим ему сообщение `TBM_GETPOS`. Позицию ползунка получим из младшего слова возвращаемого значения функции `SendMessage()`. После чего выведем это значение в статическом элементе `IDC_TR1` так же, как мы это делали ранее при инициализации ползунка.

Примечание

Когда ползунок находится в фокусе ввода (элемент управления, находящийся в фокусе, готов принимать команды с клавиатуры), его состоянием можно управлять клавишами управления курсором: <End> — минимальное значение, <Home> — максимальное значение, <→> и <↓> — увеличение значения на 1, <←> и <↑> — уменьшение значения на 1, <Page Up> — увеличение значения на шаг, <Page Down> — уменьшение значения на шаг. Шаг ползунка определяется автоматически в 1/5 от диапазона его изменения.

Spin (Up-Down Control) — наборный счетчик

Наборный счетчик представляет собой специальный вид полосы вертикального скроллинга, он состоит из кнопок вверх и вниз так же, как и элемент управления **Vertical Scroll Bar**, и порождает сообщение `WM_VSCROLL`. В `lParam` передается дескриптор элемента управления, породившего сообщение.

В оконной функции диалогового окна необходимо описать дескриптор спина `hSpin`, а в сообщении об инициализации диалога определить этот дескриптор и установить его начальное значение.

Опишем в функции диалогового окна дескриптор спина:

```
static HWND hSpin;
```

Для отображения значения спина в статическом элементе можно поступить как для ползунка — вывести значение функцией `SetDlgItemInt()`. Однако счетчик предоставляет пользователю дополнительный сервис. Если при построении диалогового окна установить свойство спина **Set Buddy Integer**, то можно этому спину назначить *"приятельское"* (*buddy*) окно `IDC_SP1`, а, посылая ему сообщение `UDM_SETBUDDY`, в `wParam` передать дескриптор *"приятеля"* `hBuddy`:

```
hSpin = GetDlgItem(hDlg, IDC_SPIN1);
hBuddy = GetDlgItem(hDlg, IDC_SP1);
SendMessage(hSpin, UDM_SETBUDDY, (WPARAM)hBuddy, 0);
```

В этом случае счетчик сам позаботится о выводе своего значения в приятельское окно, и нам об этом беспокоиться больше не нужно.

Сообщение `UDM_SETRANGE` устанавливает диапазон изменения счетчика так же, как для ползунка.

```
SendDlgItemMessage(hDlg, IDC_SPIN1, UDM_SETRANGE, 0, 100);
```

> ***Примечание***
>
> Если в диапазоне значений спина минимальное значение превышает максимальное, `LOWORD(lParam) > HIWORD(lParam)`, инвертируется направление изменения спина.

Сообщение `UDM_SETPOS` устанавливает позицию спина, передаваемую в `lParam`.

```
SendMessage(hSpin, UDM_SETPOS, 0, spin);
```

Поскольку мы установили для наборного счетчика (спина) приятельское окно, нет необходимости в обработке сообщения `WM_VSCROLL` для индикации числового значения.

Progress Bar Control — индикатор выполнения

Индикатор выполнения — это элемент управления, который показывает течение процесса, например, при копировании файла в файловом менеджере. Управление осуществляется передачей сообщений, сам же индикатор сообщений не генерирует.

Как и для предыдущих элементов управления, опишем в диалоговой оконной функции дескриптор индикатора:

```
static HWND hProgress;
```

и определим его при инициализации диалогового окна.

```
hProgress = GetDlgItem(hDlg, IDC_PROGRESS1);
```

Для индикатора необходимо определить диапазон изменения и задать начальную позицию. Диапазон `[0,100]` определим посылкой сообщения `PBM_SETRANGE`, где минимум задается в младшем слове `lParam`, а максимум — в старшем. Шаг индикатора, равный `1`, зададим в `wParam` сообщения `PBM_SETSTEP`. Начальную позицию, равную `t`, зададим посылкой сообщения `PBM_SETPOS`. Здесь `t` — статическая переменная целого типа, описанная в оконной функции диалога, будет контролировать состояние индикатора.

```
SendMessage(hProgress, PBM_SETRANGE, 0, 100<<16);
SendMessage(hProgress, PBM_SETSTEP, 1, 0);
SendMessage(hProgress, PBM_SETPOS, t, 0);
```

Для создания иллюзии движения создадим таймер с интервалом в 0,1 секунды:

```
SetTimer(hDlg, 1, 100, NULL);
```

Теперь в обработчике сообщения от таймера `WM_TIMER` будем увеличивать значение переменной на `1` и переустанавливать состояние индикатора.

```
    if (++t > 99) t = 0;
    SendMessage(hProgress, PBM_SETPOS, t, 0);
```

Оператором `if` через каждые 10 секунд сбрасываем индикатор в начальное состояние.

Осталось только перед закрытием диалогового окна кнопкой **OK** передать глобальной переменной значение индикатора и уничтожить таймер:

```
progress = t;
KillTimer(hDlg,1);
```

Поскольку состояние ползунка отслеживает переменная `track`, передадим ее на глобальный уровень:

```
::track = track;
```

Считываем состояние наборного счетчика, передавая ему сообщение `UDM_GETPOS`, и перерисовываем главное окно функцией `InvalidateRect()` (см. листинг 3.4). После чего диалоговое окно можно закрыть.

При нажатии кнопки **Cancel** уничтожаем таймер и закрываем диалоговое окно без перерисовки главного окна.

Листинг 3.4. Тест общих элементов управления

```
#include <commctrl.h>
INT_PTR CALLBACK Dialog1(HWND, UINT, WPARAM, LPARAM);
static int spin, track, progress;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   TCHAR str[256];
   RECT rt;
   switch (message)
   {
      case WM_CREATE:
         InitCommonControls();
         break;
      case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_COMMCTRL:
         DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, Dialog1);
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      SetRect(&rt, 0, 0, 100, 100);
      hdc = BeginPaint(hWnd, &ps);
```

```
      _stprintf(str,_T("spin\t= %d\ntrack\t= %d\nprogress= %d"),
               spin, track, progress);
      DrawText(hdc, str,_tcslen(str), &rt, DT_LEFT |DT_EXPANDTABS);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
//////////////////////////////////////////////////////////////////
INT_PTR CALLBACK Dialog1(HWND hDlg, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   static int t, track;
   static HWND hSpin, hBuddy, hTrack, hProgress;
   switch (message)
   {
      case WM_INITDIALOG:
            track = ::track;
            SetDlgItemInt(hDlg, IDC_TR1, track, 0);
            hTrack = GetDlgItem(hDlg, IDC_SLIDER1);
            SendMessage(hTrack, TBM_SETRANGE, 0, 100<<16);
            SendMessage(hTrack, TBM_SETPOS, TRUE, track);
            hSpin = GetDlgItem(hDlg, IDC_SPIN1);
            hBuddy = GetDlgItem(hDlg, IDC_SP1);
            SendMessage(hSpin, UDM_SETBUDDY, (WPARAM)hBuddy, 0);
            SendMessage(hSpin, UDM_SETRANGE, 0, 100);
            SendMessage(hSpin, UDM_SETPOS, 0, spin);
            hProgress = GetDlgItem(hDlg, IDC_PROGRESS1);
            SendMessage(hProgress, PBM_SETRANGE, 0, 100<<16);
            SendMessage(hProgress, PBM_SETSTEP, 1, 0);
            SendMessage(hProgress, PBM_SETPOS, t, 0);
            SetTimer(hDlg, 1, 100, NULL);
            return TRUE;
      case WM_TIMER :
            if (++t > 99) t = 0;
            SendMessage(hProgress, PBM_SETPOS, t, 0);
            return TRUE;
         case WM_HSCROLL:
            track = LOWORD(SendMessage(hTrack, TBM_GETPOS, 0, 0));
            SetDlgItemInt(hDlg, IDC_TR1, track, 0);
            return TRUE;
```

```
      case WM_COMMAND:
         switch(LOWORD(wParam))
         {
         case IDOK :
            progress = t;
            ::track = track;
            spin = SendMessage(hSpin, UDM_GETPOS, 0, 0);
            InvalidateRect(GetParent(hDlg),NULL,1);
         case IDCANCEL:
            KillTimer(hDlg,1);
            EndDialog(hDlg, 0);
            return TRUE;
         default: return FALSE;
         }
         default: return FALSE;
   }
    return FALSE;
}
```

Окно редактирования

Для ввода данных чаще всего используется элемент управления — *окно редактирования* (Edit Box Control). Управление окном осуществляется передачей ему сообщений или специализированными функциями диалогового окна:

- `GetDlgItemText(hDlg, IDC_EDIT, text, length)` — возвращает в `TCHAR` массив `text` не более `length` символов окна редактирования;
- `SetDlgItemText(hDlg, IDC_EDIT, text)` — заполняет окно редактирования содержимым `TCHAR`-строки `text`;
- `GetDlgItemInt(hDlg, IDC_EDIT, lpTranslated, bSigned)` — функция возвращает целое число, в которое преобразуется содержимое окна редактирования. `lpTranslated` — указатель переменной типа `BOOL`, которая устанавливается в `TRUE` при успешном преобразовании числа, и `FALSE` — в противном случае. Если `bSigned` равно `TRUE`, преобразуется число со знаком иначе, число рассматривается как беззнаковое;
- `SetDlgItemInt(hDlg, IDC_EDIT1, Value, bSigned)` — устанавливается значение переменной целого типа `Value` в окне редактирования. Если `bSigned` равно `TRUE`, рассматривается число со знаком.

Для демонстрации возможностей этого окна создадим тестовую программу (листинг 3.5), вид ее окна приведен на рис. 3.8. В диалоговом окне осуществляется ввод текстовых строк в элемент управления список, а после закрытия диалогового окна данные выводятся в главном окне. Предусмотрена возможность удаления строки списка.

Дополнительно выведем в нижней части главного окна строку состояния (строку статуса), где предусмотрим отображение количества строк списка.

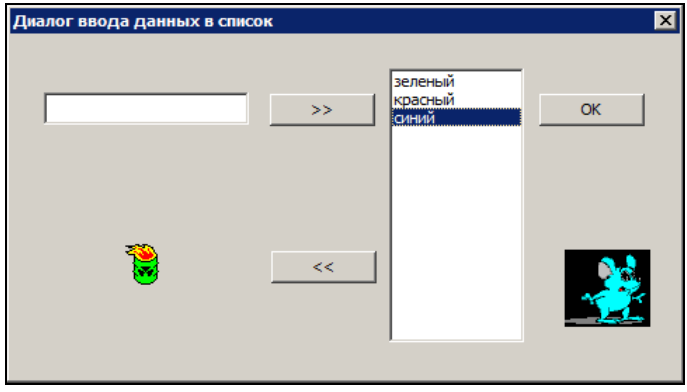


Рис. 3.8. Демонстрационная программа "Окно редактирования"

Листинг 3.5. Окно редактирования и список

```
#include <vector>
#include <xstring>
#include <commctrl.h>
INT_PTR CALLBACK Dialog1(HWND, UINT, WPARAM, LPARAM);
typedef std::basic_string<TCHAR, std::char_traits<TCHAR>,
std::allocator<TCHAR> > String;
std::vector<String> v;
HWND hWndStatusBar;   //Дескриптор строки состояния
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   TEXTMETRIC tm;
   static int cy, sx, sy;
   int y;
   RECT rt;
   std::vector<String>::iterator it;
   switch (message)
   {
   case WM_CREATE:
      hdc = GetDC(hWnd);
      GetTextMetrics(hdc,&tm);
      cy = tm.tmHeight + tm.tmExternalLeading;
```

```
      ReleaseDC(hWnd, hdc);
      hWndStatusBar = CreateStatusWindow(WS_CHILD | WS_VISIBLE |
      WS_CLIPSIBLINGS | CCS_BOTTOM | SBARS_SIZEGRIP,_T("Ready"), hWnd, 1);
      break;
   case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      GetWindowRect(hWndStatusBar, &rt);
      y = rt.bottom-rt.top;
      MoveWindow(hWndStatusBar, 0, sy - y, sx, sy, TRUE);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_DIALOG_READLISTBOX:
         DialogBox(hInst, MAKEINTRESOURCE(IDD_READ), hWnd, Dialog1);
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
         for (y = 0, it = v.begin(); it != v.end(); ++it, y += cy)
            TextOut(hdc, 0, y, it->data(), it->size());
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
//////////////////////////////////////////////////////////////////
INT_PTR CALLBACK Dialog1(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
   static HWND hList, hEdit;
   TCHAR text[256];
   int i, k;
   switch (message)
   {
      case WM_INITDIALOG:
            hList = GetDlgItem(hDlg, IDC_LIST1);
```

```
            hEdit = GetDlgItem(hDlg, IDC_EDIT1);
            return TRUE;
      case WM_COMMAND:
         switch (LOWORD(wParam))
         {
         case IDC_ADD:
            GetDlgItemText(hDlg, IDC_EDIT1, text, 256);
            SetDlgItemText(hDlg, IDC_EDIT1, _T(""));
            SendMessage(hList, LB_ADDSTRING, 0, (LPARAM)text);
            SetFocus(hEdit);
            return TRUE;
         case IDC_DEL:
            k = SendMessage(hList, LB_GETCURSEL, 0, 0);
            SendMessage(hList, LB_DELETESTRING , (WPARAM)k, 0);
            return TRUE;
         case ID_OK:
            v.clear();
            k = SendMessage(hList, LB_GETCOUNT, 0, 0);
            for (i = 0; i < k; i++)
            {
               SendMessage(hList, LB_GETTEXT, (WPARAM)i, (LPARAM)text);
               v.push_back(text);
            }
            InvalidateRect(GetParent(hDlg), NULL, 1);
            _stprintf(text,_T("Список: %d строк"), k);
            SendMessage(hWndStatusBar, WM_SETTEXT, 0, (LPARAM)text);
         case WM_DESTROY: EndDialog(hDlg, LOWORD(wParam)); return TRUE;
         }
      default: return FALSE;
   }
    return FALSE;
}
```

В качестве контейнера для хранения текстовых строк, получаемых из списка, используем контейнер `vector` для типа данных `String`, производного от шаблонного класса `basic_string` для типа `TCHAR` (см. листинг 1.3).

В заголовке программы добавим файлы включений:

```
#include <vector>
#include <xstring>
```

Опишем на глобальном уровне вектор, явно указывая область видимости:

```
std::vector<String> v;
```

Итератор вектора нет смысла описывать на глобальном уровне, поэтому опишем его в функции главного окна:

```
std::vector<String>::iterator it;
```

Нужно позаботиться о выводе результатов в главное окно, поэтому в сообщении `WM_CREATE` найдем высоту строки текущего шрифта при помощи функции `GetTextMetrics()`, как мы это делали в листинге 1.13.

Переменную `tm` типа `TEXTMETRIC` и статическую переменную `cy`, необходимую для хранения высоты шрифта, предварительно опишем в оконной функции.

Диалоговое окно будем вызывать через главное меню, где создадим пункт с идентификатором `ID_DIALOG_READLISTBOX`. `Dialog1` — имя оконной функции диалога. Именно в этой функции и реализуется вся логика работы с элементами управления.

На поверхности диалогового окна разместим следующие элементы:

- поле ввода (EditBox) с идентификатором `IDC_EDIT1`;
- список (ListBox) с идентификатором `IDC_LIST1`;
- кнопку `IDC_ADD`;
- кнопку `IDC_DEL`;
- кнопку `ID_OK`;
- иконку `IDC_STATIC` с изображением `IDI_TRASH`;
- битовый образ с изображением `IDB_MOUSE`.

Последние два элемента мы ввели, чтобы "оживить" диалоговое окно, а картинки нашли в имеющейся коллекции и импортировали в созданный проект (см. рис. 3.2). Разместим их в элементе управления Picture Control.

Начнем рассмотрение функции диалогового окна `Dialog1()`.

В сообщении `WM_INITDIALOG` определим дескрипторы окна редактирования и списка:

```
hList = GetDlgItem(hDlg, IDC_LIST1);
hEdit = GetDlgItem(hDlg, IDC_EDIT1);
```

Эти переменные описаны в заголовке функции:

```
static HWND hList, hEdit;
```

Теперь обработаем реакцию на нажатие кнопки с изображением "**>>**".

Функцией `GetDlgItemText()` читаем содержимое окна редактирования в массив `text` и функцией `SetDlgItemText()` очищаем окно, посылая "пустую" строку.

Посылая списку сообщение `LB_ADDSTRING`, добавим к нему строку `text`. Теперь установим фокус ввода обратно на окно редактирования для последующего ввода:

```
SetFocus(hEdit);
```

> ***Примечание***
>
> Сообщение `LB_ADDSTRING` позволяет добавлять строки текста в конец списка, если для него не установлено свойство `Sort`. Если же свойство установлено, то строки будут добавляться в порядке, установленном критерием сортировки по умолчанию.

Обработаем реакцию на нажатие кнопки "**<<**" для удаления выделенной в списке строки. Посылая списку сообщение `LB_GETCURSEL`, получим индекс выделенного элемента, а сообщением `LB_DELETESTRING` удалим элемент с найденным индексом.

Нажатием на кнопку **OK** (идентификатор которой заменим на `ID_OK`, и установим в `False` свойство `Default Button`, чтобы избежать стандартной реакции на нажатие клавиши <Enter>) извлечем из списка введенные строки и поместим их в контейнер `vector`, описанный на глобальном уровне.

Очистим содержимое контейнера:

```
v.clear();
```

Найдем размер списка, передавая ему сообщение `LB_GETCOUNT`:

```
k = SendMessage(hList, LB_GETCOUNT, 0, 0);
```

В цикле читаем последовательно содержимое списка, посылая ему сообщение `LB_GETTEXT`, где третьим параметром служит индекс извлекаемого элемента, а четвертый параметр — массив `text` для хранения строки текста. Прочитав строку текста, помещаем ее в контейнер методом `push_back()`. `TCHAR`-строка автоматически преобразуется к типу `String`.

```
for (i = 0; i < k; i++)
{
    SendMessage(hList, LB_GETTEXT, (WPARAM)i, (LPARAM)text);
    v.push_back(text);
}
```

> ***Примечание***
>
> В этой задаче применение контейнера для хранения строк не является необходимым, поскольку количество строк в списке известно, и можно было бы выделить массив типа `String`.

Функцией `InvalidateRect()` инициируем перерисовку главного окна программы, где дескриптор главного окна получим обращением к функции `GetParent(hDlg)`.

Мы не поставили оператор `return` перед следующим оператором `case` намеренно, поскольку выполнение кода будет продолжаться, произойдет переход на операторы закрытия диалогового окна и выхода из функции:

```
EndDialog(hDlg, LOWORD(wParam)); return TRUE;
```

что и требовалось.

Сообщение с кодом `WM_DESTROY` будет генерироваться при нажатии на кнопку завершения приложения системного меню. В этом случае приложение завершится без сохранения данных списка и перерисовки главного окна.

Мы поместили на диалоговом окне еще два элемента **Picture Control**, оставив для них идентификатор по умолчанию `IDC_STATIC`, поскольку нет необходимости доступа к ним. Для первого элемента выбрали тип `Icon`, а в качестве изображения выбрали идентификатор `IDI_TRASH` импортированной иконки, которую подобрали в специализированной библиотеке.

Для второго элемента выберем тип `Bitmap`, а в качестве битового образа — `IDB_MOUSE`. Этот идентификатор мы получим, импортируя bmp-файл с растровым

изображением. Для полноты картины заменим и иконку приложения. Для этого импортируем еще одну иконку из найденной коллекции и присвоим ей идентификатор IDI_FLGRUS.

Теперь осталось изменить стиль окна, для чего отредактируем лишь одно поле класса окна:

```
wcex.hIconSm  = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_FLGRUS));
```

Вот и все, теперь в качестве иконки приложения будет использоваться найденное нами изображение.

Строка состояния

Для того чтобы придать приложению, представленному в листинге 3.5, более современный вид, добавим в окно строку состояния. Эта строка располагается обычно в нижней части окна и выводит справочную информацию о состоянии приложения. Выведем здесь количество строк списка.

Поскольку окно состояния является общим элементом управления, необходимо добавить файл включений commctrl.h и, соответственно, библиотечный файл comctl32.lib.

Дескриптор окна состояния опишем на глобальном уровне:

```
HWND hWndStatusBar;
```

и создадим окно при обработке сообщения WM_CREATE главного окна функцией CreateStatusWindow():

```
HWND WINAPI CreateStatusWindowW(LONG style, LPCWSTR lpszText, HWND
hwndParent, UINT wID);
```

где:

- style — стиль окна;
- lpszText — текст по умолчанию;
- hwndParent — дескриптор родительского окна;
- wID — идентификатор окна.

К сожалению, окно состояния не может самостоятельно подстроиться под размер окна при его изменении, поэтому необходимо написать дополнительный код при обработке сообщения WM_SIZE:

```
GetWindowRect(hWndStatusBar, &rt);
y = rt.bottom-rt.top;
MoveWindow(hWndStatusBar, 0, sy - y, sx, sy, TRUE);
```

Теперь нужно решить вопрос — где мы будем выводить информацию в строку состояния? Проще всего это сделать при обработке сообщения о нажатии кнопки **OK** диалогового окна. Для этого мы сформируем строку вывода text:

```
_stprintf(text,_T("Список: %d строк"), k);
```

и выведем эту строку в окно состояния, послав ему сообщение WM_SETTEXT:

```
SendMessage(hWndStatusBar, WM_SETTEXT, 0, (LPARAM)text);
```

Простой текстовый редактор на элементе управления Edit Box Control

Мы рассмотрели использование окна редактирования Edit Box для ввода строки текста, однако этот элемент управления имеет гораздо больше возможностей. В качестве демонстрационного примера построим на его основе простой текстовый редактор. Как мы вскоре убедимся, все операции по вводу и редактированию текста элемент Edit Box берет на себя, нам остается лишь реализовать внешний интерфейс. Следует оговориться, что элемент управления Edit Box использует внутреннюю память для хранения текста, а это приводит к ограничениям на размер редактируемого текста в 32 767 символов (`0x7fff`).

> ***Примечание***
>
> В операционной системе Windows NT и выше размер буфера Edit Box может быть увеличен посылкой ему сообщения `EM_LIMITTEXT` до `0x7FFFFFFE` байтов.

В качестве основы используем стандартный проект Win32. Добавим три пункта меню: **New**, **Open**, **Save**, панель инструментов с кнопками, соответствующими этим пунктам меню, а также строку состояния (см. листинг 3.6). Будем создавать универсальный проект, однако, поскольку приложение ориентировано на работу с текстовыми файлами в однобайтной кодировке, будем явно указывать тип `char*` для входных и выходных массивов данных.

Листинг 3.6. Текстовый редактор с элементом управления Edit Box

```
#include <commdlg.h>
#include <commctrl.h>
#include <fstream>
TBBUTTON tbb[] =
{{STD_FILENEW, ID_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0},
 {STD_FILEOPEN, ID_FILE_OPEN,TBSTATE_ENABLED,TBSTYLE_BUTTON, 0, 0, 0, 0},
 {STD_FILESAVE, ID_FILE_SAVE,TBSTATE_ENABLED,TBSTYLE_BUTTON, 0, 0, 0, 0}
};
//////////////////////////////////////////////////////////////////////
VOID StatusOut(HWND hStatus, int count, TCHAR *str)
{
   TCHAR text[256];
   _stprintf(text,_T("Строк: %d"), count);
   SendMessage(hStatus, SB_SETTEXT, (WPARAM)0, (LPARAM)text);
   SendMessage(hStatus, SB_SETTEXT, (WPARAM)1, (LPARAM)str);
}
//////////////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   static OPENFILENAME file;
```

```
static int n, sx, sy;
static HWND hEdit, hWndToolBar, hWndStatusBar;
RECT rt;
int m, k, aWidths[2];
static TCHAR name[256];
char szText[0x7fff];
std::ifstream in;
std::ofstream out;
switch (message)
{
case WM_CREATE:
   hWndToolBar = CreateToolbarEx(hWnd, WS_CHILD|WS_VISIBLE|CCS_TOP,
   2, 0, HINST_COMMCTRL, IDB_STD_SMALL_COLOR, tbb, 3, 0, 0, 0, 0,
   sizeof(TBBUTTON));
   hEdit = CreateWindow(WC_EDIT,NULL,WS_CHILD|WS_VISIBLE|WS_HSCROLL|
         WS_VSCROLL|ES_LEFT|ES_MULTILINE|ES_AUTOHSCROLL|ES_AUTOVSCROLL,
         0, 0, 0, 0, hWnd, (HMENU) 1, hInst, NULL);
   file.lStructSize     = sizeof(OPENFILENAME);
   file.hInstance       = hInst;
   file.lpstrFilter     = _T("Text\0 *.txt\0Все файлы\0 *.*");
   file.lpstrFile       = name;
   file.nMaxFile        = 256;
   file.lpstrInitialDir = _T(".\\");
   file.lpstrDefExt     = _T("txt");
   hWndStatusBar = CreateWindow(STATUSCLASSNAME, NULL, WS_CHILD |
                   WS_VISIBLE, 0, 0, 0, 0, hWnd, NULL, hInst, NULL);
   break;
case WM_SIZE:
   sx = LOWORD(lParam);
   sy = HIWORD(lParam);
   aWidths[0] = 100;
   aWidths[1] = sx;
   GetWindowRect(hWndToolBar, &rt);
   m = rt.bottom - rt.top;
   SendMessage(hWndToolBar, TB_AUTOSIZE, 0, 0);
   GetWindowRect(hWndStatusBar, &rt);
   k = rt.bottom - rt.top;
   MoveWindow(hWndStatusBar, 0, sy - k, sx, sy, TRUE);
   SendMessage(hWndStatusBar, SB_SETPARTS, (WPARAM)2, (LPARAM)aWidths);
   StatusOut(hWndStatusBar, n, name);
   MoveWindow(hEdit, 0, m, sx, sy - m - k, TRUE);
   UpdateWindow(hEdit);
   SetFocus(hEdit);
   return 0;
```

```
case WM_COMMAND:
   switch (LOWORD(wParam))
   {
   case ID_FILE_NEW:
      szText[0] = '\0';
      SetWindowTextA(hEdit, szText);
      StatusOut(hWndStatusBar, 0, _T(""));
      break;
   case ID_FILE_OPEN:
      file.lpstrTitle = _T("Открыть файл для чтения");
      file.Flags = OFN_HIDEREADONLY;
      if (!GetOpenFileName(&file)) return 1;
      in.open(name, std::ios::binary);
      in.read(szText, 0x7fff);
      if ((m = in.gcount()) == 0x7fff)
         {
            MessageBox(hWnd, _T("Слишком большой файл"),
                        _T("Edit"),MB_OK | MB_ICONSTOP);
            in.close();
            return 0;
         }
      szText[m] = '\0';
      in.close();
      SetWindowTextA(hEdit, szText);
      n = SendMessage(hEdit, EM_GETLINECOUNT, 0, 0);
      StatusOut(hWndStatusBar, n, name);
      break;
   case ID_FILE_SAVE:
      file.lpstrTitle = _T("Открыть файл для записи");
      file.Flags = OFN_NOTESTFILECREATE | OFN_HIDEREADONLY;
      if (!GetSaveFileName(&file)) return 1;
      out.open(name, std::ios::binary);
      m = GetWindowTextA(hEdit, szText, 0x7fff);
      out.write(szText, m);
      out.close();
      n = SendMessage(hEdit, EM_GETLINECOUNT, 0, 0);
      StatusOut(hWndStatusBar, n, name);
      break;
   case IDM_EXIT: DestroyWindow(hWnd); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   break;
```

```
   case WM_DESTROY:
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Нам понадобятся файлы включений: `commdlg.h`, `commctrl.h`, `fstream`. Доступ к стандартной области имен открывать не будем, а опишем переменные с явным указанием области видимости.

Как и в программе просмотра файлов, рассмотренной в *главе 2* (см. листинг 2.3), опишем в массиве `TBBUTTON` 3 кнопки инструментальной панели. Саму панель инструментов создадим при обработке сообщения `WM_CREATE` и получим ее дескриптор `hwndToolBar`:

```
hwndToolBar = CreateToolbarEx(hWnd, WS_CHILD|WS_VISIBLE|CCS_TOP, 2, 0,
HINST_COMMCTRL,IDB_STD_SMALL_COLOR,tbb, 3, 0, 0, 0,sizeof(TBBUTTON));
```

Корректировку размера панели инструментов вставим в обработчик сообщения `WM_SIZE`, посылая ей сообщение `TB_AUTOSIZE`:

```
SendMessage(hwndToolBar, TB_AUTOSIZE, 0, 0);
```

Здесь же создадим элемент управления `Edit Box`:

```
hEdit = CreateWindow(WC_EDIT, NULL, WS_CHILD|WS_VISIBLE|WS_HSCROLL|
         WS_VSCROLL|ES_LEFT|ES_MULTILINE|ES_AUTOHSCROLL|ES_AUTOVSCROLL,
         0, 0, 0, 0, hWnd, (HMENU) 1, hInst, NULL);
```

В параметре *Класс окна* функции `CreateWindow()` указываем `WC_EDIT` — предопределенный в системе идентификатор окна Edit Box. Заголовка у окна нет, поэтому второй параметр `NULL`.

Стиль окна будет складываться из следующих компонент:

- `WS_CHILD` — дочернее окно;
- `WS_VISIBLE` — окно отображается при создании;
- `WS_HSCROLL` — имеет горизонтальную полосу скроллинга;
- `WS_VSCROLL` — имеет вертикальную полосу скроллинга;
- `ES_LEFT` — выравнивание текста влево;
- `ES_MULTILINE` — многострочное окно редактирования;
- `ES_AUTOHSCROLL` — автоматическая прокрутка текста при вводе строки;
- `ES_AUTOVSCROLL` — автоматическая прокрутка текста, когда окно заполнено.

Все четыре параметра расположения и размера окна зададим равным `0`, а реальные размеры определим позднее в сообщении `WM_SIZE`.

Далее укажем дескриптор родительского окна `hWnd`. Следующий параметр — указатель меню — в данном контексте служит для задания идентификатора окна: при-

своим ему номер 1. Укажем следующим параметром дескриптор приложения — hInst и последний параметр — NULL, поскольку дополнительные параметры отсутствуют.

Окно создано, но мы хотели бы наложить его на главное окно программы, размеры которого нужно определять в сообщении WM_SIZE. Сделаем это функцией:

```
MoveWindow(hEdit, 0, m, sx, sy - m - k, TRUE);
```

Функция MoveWindow() позволяет изменить положение (0, m) и размер окна (sx, sy – m - k). Поскольку окно редактирования объявлено как дочернее, то располагается оно в клиентской области главного окна и начало его нужно поместить в левом верхнем углу клиентской области. Но там располагается панель инструментов, высоту которой мы определим, обратившись к функции GetWindowRect():

```
GetWindowRect(hwndToolBar,& rt);
m = rt.bottom - rt.top;
```

Таким образом, левый верхний угол окна редактирования нужно сместить вниз по оси *y* на m. Высоту окна следует уменьшить на вертикальный размер панели инструментов и строки состояния, которая также располагается в клиентской области главного окна: sy – m - k.

Строку состояния создадим с нулевыми размерами функцией CreateWindow() в сообщении WM_CREATE:

```
hWndStatusBar = CreateWindow(STATUSCLASSNAME, NULL, WS_CHILD |
                WS_VISIBLE, 0, 0, 0, 0, hWnd, NULL, hInst, NULL);
```

> ***Примечание***
>
> Идентификатор строки состояния STATUSCLASSNAME определен в файле включений как: "msctls_statusbar32".

Реальный же размер зададим в сообщении WM_SIZE:

```
MoveWindow(hWndStatusBar, 0, sy - k, sx, sy, TRUE);
```

где sx, sy — ширина и высота главного окна, а k — высота окна состояния.

Воспользуемся некоторыми особенностями этого окна управления. Имеется возможность разделить строку состояния на несколько частей и выводить информацию в каждую часть отдельно. Для этого необходимо, послав сообщение SB_SETPARTS, указать в wParam количество частей, а в lParam их правые границы, заданные массивом aWidths:

```
SendMessage(hWndStatusBar, SB_SETPARTS, (WPARAM)2, (LPARAM)aWidths);
```

Поделим строку состояния на две части, установив фиксированный размер первой части, а для второй части отдадим остаток строки:

```
aWidths[0] = 100;
aWidths[1] = sx;
```

Вывод в строку состояния организуем в локальной функции StatusOut(), куда передаем количество строк текста count и имя файла str.

Сформируем вывод в первую часть строки состояния функцией _stprintf() и отправим данные, послав окну сообщение SB_SETTEXT. Номер части указывается

в WPARAM сообщения. Во вторую часть поместим имя файла str, полученное из диалогового окна открытия файла.

После установки всех размеров необходимо обратиться к функции:

```
UpdateWindow(hEdit);
```

иначе окно не успеет перерисоваться.

Что еще важно, так это обеспечить получение фокуса ввода окном редактирования:

```
SetFocus(hEdit);
```

Собственно, что касается окна редактирования, то оно уже должно работать — можно вводить текст и редактировать его, доступны скроллинг и контекстное меню.

Оставшийся код необходим для организации чтения файла в окно редактирования и записи в файл отредактированного текста.

Как и в задаче, представленной листингом 2.3, в сообщении о создании окна заполним структуру OPENFILENAME. И лишь два поля этой структуры: lpstrTitle и Flags отличающихся при чтении и записи, мы заполним при обработке команд чтения и записи файла. Диалог выбора файла реализуется функцией GetOpenFileName().

Но здесь имеется одна особенность — дело в том, что окно редактирования имеет собственную память, и нам нет необходимости вводить контейнер для хранения текста, вместо этого нужно поместить прочитанный файл в окно редактирования. Нужно учесть еще и ограничение — максимальный размер текста в окне редактирования 32К.

> ***Примечание***
>
> Размер буфера окна редактирования Edit Box может быть увеличен посылкой ему сообщения EM_SETLIMITTEXT.

И еще один момент — строки в окне редактирования разделяются парой символов "\r\n" так же, как и в файле, и это не случайно.

Можно, конечно, организовать чтение файла построчно и добавлять эти строки в окно редактирования, однако проще прочитать файл как двоичный в буфер szText и оттуда отправить весь текст в окно, так мы и поступим.

Итак, суммируя все сказанное: открываем файл как двоичный и читаем блоком не более 0x7fff символов.

```
in.open(name, std::ios::binary);
in.read(szText, 0x7fff);
```

Если прочитали ровно 0x7fff символов, то файл имеет больший размер и не может быть помещен в окно редактирования. Мы выводим предупреждающее сообщение и прекращаем чтение:

```
if ((m = in.gcount()) == 0x7fff)
{
MessageBox(hWnd,_T("Слишком большой файл"),_T("Edit"),MB_OK|MB_ICONSTOP);
in.close();
return 0;
}
```

Если же размер файла меньше критического, добавим признак завершения строки `'\0'`, поскольку метод `read()` возвращает массив символов, а нам нужна С-строка:

```
szText[m] = '\0';
```

После чего закрываем файл.

Теперь поместим текст в виде строки в окно редактирования. Это делается функцией:

```
SetWindowText(hEdit, szText);
```

Осталось только подсчитать количество текстовых строк, передавая окну редактора сообщение `EM_GETLINECOUNT`, и обновить строку состояния.

Для сохранения файла, при обработке пункта меню **Save**, нужно решить обратную задачу. Так же открываем двоичный файл для записи, но метод для записи блока требует задания его размера, поэтому при копировании текста из окна редактирования в буфер функцией `GetWindowText()` сохраняем его размер `m` в байтах. После чего записываем буфер в файл.

```
out.open(name, std::ios::binary);
m = GetWindowText(hEdit, szText, 0x7fff);
out.write(szText, m);
out.close();
```

Вновь определяем количество строк текста и обновляем строку состояния.

Еще проще выглядит обработка команды меню **New**. Мы сделаем буфер текста "пустым", просто присвоив первому байту массива символ `'\0'`. Теперь осталось послать эту "пустую" строку в окно редактирования, и окно очистится:

```
szText[0] = '\0';
SetWindowText(hEdit, szText);
```

Теперь нужно очистить строку состояния. Вот и все! Вид работающего приложения представлен на рис. 3.9. Вся логика ввода и редактирования текста, а также скроллинг и контекстное меню уже реализованы в окне редактирования, и нам осталось лишь пожинать плоды выполненной работы.

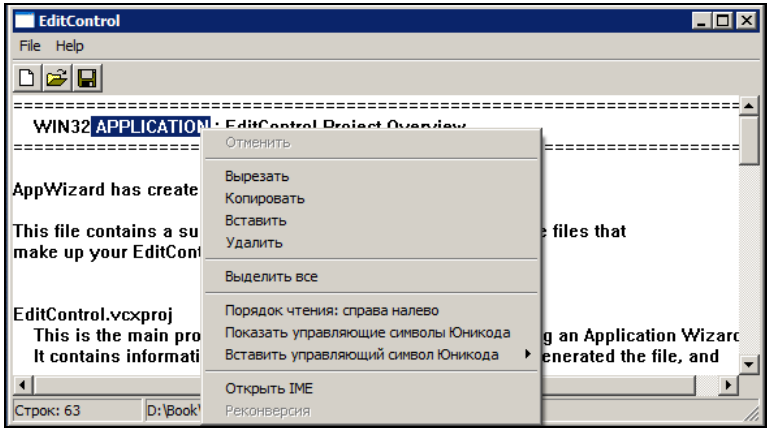


Рис. 3.9. Простейший редактор текстов с элементом управления Edit Box Control

Немодальные окна

До сих пор мы имели дело с модальными окнами, которые после создания становятся активными, и пока они открыты, управление не может быть передано другому окну приложения. Немодальные же окна работают принципиально иначе — после создания они могут как потерять управление, так и получить его вновь. Примером такого окна может служить диалоговое окно поиска **Найти и заменить** (Find and Replace) в текстовом редакторе Word.

Для демонстрации работы немодального окна построим приложение с диалоговым окном выбора цвета фона главного окна, состоящего из трех линеек скроллинга, для базовых цветов: красного, зеленого, синего (рис. 3.10).

В качестве основы используем стандартную заготовку, куда добавим пункт меню **Color**, при обработке которого создадим диалоговое окно, но, поскольку нам нужно немодальное окно, создавать окно будем функцией `CreateDialog()`:

```
hDlgColor = CreateDialog(hInst,MAKEINTRESOURCE(IDD_COLOR),hWnd,Dialog);
```

Функция имеет тот же набор параметров, что и `DialogBox()`, но возвращает дескриптор окна, который мы опишем на глобальном уровне. На самом деле это макрос, который преобразуется в функцию `CreateDialogParam()`.

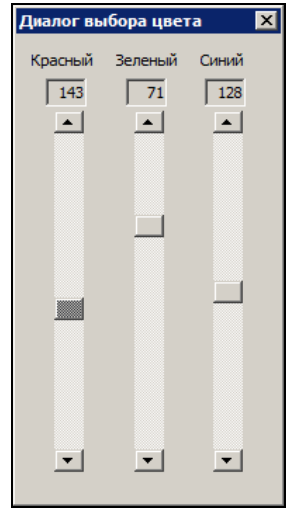


Рис. 3.10. Немодальное диалоговое окно

> ***Примечание***
>
> При создании диалогового окна нужно установить для него свойство `Visible`. Иначе окно, созданное функцией `CreateDialog()`, не будет отображаться.

Поскольку немодальное окно может потерять управление, то цикл обработки сообщений головной функции `WinMain()` должен определить источник сообщения, которое может исходить либо от главного, либо немодального окна. Для этого добавим оператор `if` в цикл обработки сообщений:

```
while (GetMessage(&msg, NULL, 0, 0))
{
   if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
      if (!IsDialogMessage(hDlgColor, &msg))
        {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
        }
}
```

Функция `IsDialogMessage()` определяет, предназначено ли сообщение для указанного диалогового окна и, если это так, обрабатывает сообщение, возвращая `TRUE`, иначе обработка сообщения происходит традиционным образом оконной функцией главного окна. Так происходит "диспетчеризация" сообщений и, например, щелчок

мыши на диалоговом окне приведет к обработке этого сообщения функцией диалогового окна, а такой же щелчок в главном окне будет обрабатываться функцией главного окна. *В связи с этим принципиально важно, чтобы для всех сообщений, обрабатываемых в оконной функции немодального диалога, возвращаемое значение было* `TRUE`*, а для всех прочих сообщений —* `FALSE`*.*

Далее приведем листинг 3.7 с текстом функции окна немодального диалогового окна и отметим, что закрываться немодальное окно должно функцией `DestroyWindow()`.

Листинг 3.7. Оконная функция немодального диалогового окна

```
INT_PTR CALLBACK Dialog(HWND, UINT, WPARAM, LPARAM);
//////////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   switch (message)
   {
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_DIALOG_COLOR:
         hDlgColor = CreateDialog(hInst, MAKEINTRESOURCE(IDD_COLOR),
            hWnd, Dialog);
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
//////////////////////////////////////////////////////////////////
INT_PTR CALLBACK Dialog(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
   static int ID_SCROLL[3] = {IDC_RED, IDC_GREEN, IDC_BLUE};
   static int ID[3] = {IDC_R, IDC_G, IDC_B};
   static HWND hWnd, hScroll[3];           //{hRed, hGreen, hBlue}
   static int color[3] = {255, 255, 255};  //{red, green, blue}
   HBRUSH hBrush;
   int index;
   switch (message)
   {
```

```
   case WM_INITDIALOG:
      for (index = 0; index < 3; index++)
      {
         hScroll[index] = GetDlgItem(hDlg, ID_SCROLL[index]);
         SetScrollRange(hScroll[index], SB_CTL, 0, 255, FALSE);
         SetScrollPos (hScroll[index], SB_CTL, color[index], TRUE);
         SetDlgItemInt(hDlg, ID[index], color[index], 0);
      }
      return TRUE;
   case WM_VSCROLL :
      for (index = 0; index < 3; index++)
         if ((HWND)lParam == hScroll[index]) break;
      SetFocus(hScroll[index]);
      switch(LOWORD(wParam))
      {
      case SB_LINEUP   : color[index]--; break;
      case SB_LINEDOWN : color[index]++; break;
      case SB_PAGEUP   : color[index] -= 16; break;
      case SB_PAGEDOWN : color[index] += 16; break;
      case SB_THUMBTRACK:
      case SB_THUMBPOSITION : color[index] = HIWORD(wParam); break;
      }
      color[index] = max(0, min(color[index], 255));
      if (color[index] != GetScrollPos(hScroll[index], SB_CTL))
      {
         SetDlgItemInt(hDlg,ID[index],color[index],0);
         SetScrollPos(hScroll[index], SB_CTL, color[index], TRUE);
         hBrush = CreateSolidBrush(RGB(color[0],color[1],color[2]));
         hWnd = GetParent(hDlg);
         DeleteObject((HBRUSH)GetClassLong(hWnd, GCL_HBRBACKGROUND));
         SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBrush);
         InvalidateRect(hWnd, NULL, TRUE);
      }
      return TRUE;
   case WM_COMMAND:
      if (LOWORD(wParam) == IDCANCEL) {DestroyWindow(hDlg); return TRUE;}
      return FALSE;
   }
   return FALSE;
}
```

Рассмотрим оконную функцию диалогового окна. Поскольку диалоговое окно содержит 3 линейки скроллинга и 3 статических элемента для отображения состоя-

ния скроллинга, то нам удобнее описать эти элементы массивами, присвоив их элементам значение соответствующих идентификаторов:

```
static int ID_SCROLL[3] = {IDC_RED, IDC_GREEN, IDC_BLUE};
static int ID[3] = {IDC_R, IDC_G, IDC_B};
```

Так же опишем массив дескрипторов для линеек скроллинга и массив для хранения трех базовых цветов:

```
static HWND hScroll[3];                    //{hRed, hGreen, hBlue}
static int color[3] = {255, 255, 255};     //{red, green, blue}
```

Опишем дескриптор окна `hWnd` и кисти `hBrush`, а также переменную целого типа `index`.

Далее организуем обработку сообщений.

При инициализации диалогового окна в цикле определяем дескрипторы линеек скроллинга:

```
hScroll[index] = GetDlgItem(hDlg, ID_SCROLL[index]);
```

Устанавливаем диапазон скроллинга `[0;255]`:

```
SetScrollRange(hScroll[index], SB_CTL, 0, 255, FALSE);
```

и начальную позицию в `255`, поскольку начальный цвет окна — белый, соответствует значению `RGB(255, 255, 255)`.

```
SetScrollPos (hScroll[index], SB_CTL, color[index], TRUE);
```

После чего выведем значение в статическом элементе:

```
SetDlgItemInt(hDlg, ID[index], color[index], 0);
```

Теперь рассмотрим сообщение `WM_VSCROLL`, генерируемое при воздействии на линейку скроллинга. Вначале нужно определить, от какой линейки пришло сообщение. Сделаем это в цикле, сравнивая дескриптор элемента, породившего сообщение, и передаваемого в `lParam`, с дескрипторами линеек скроллинга:

```
for (index = 0; index < 3; index++)
    if ((HWND)lParam == hScroll[index]) break;
```

При выходе из цикла переменная `index` будет соответствовать индексу линейки скроллинга, которая сгенерировала сообщение.

Установим фокус ввода на выбранную линейку:

```
SetFocus(hScroll[index]);
```

Это не обязательно, но позволит нам использовать для управления клавиатуру.

Обработку сообщений скроллинга осуществим стандартным образом так же, как в листинге 2.3, только вместо переменной будем оперировать элементом массива `color[index]`:

Убедившись, что состояние скроллинга изменилось, установим новое значение в статическом элементе:

```
SetDlgItemInt(hDlg, ID[index], color[index],0);
```

а также новую позицию движка линейки:

```
SetScrollPos(hScroll[index], SB_CTL, color[index], TRUE);
```

Теперь создаем новую кисть:

```
hBrush = CreateSolidBrush(RGB(color[0],color[1],color[2]));
```

и уничтожаем старую. Дескриптор старой кисти получим обращением к функции `GetClassLong()`:

```
DWORD WINAPI GetClassLong(HWND hWnd, int nIndex);
```

если передать в качестве параметра `GCL_HBRBACKGROUND`:

```
DeleteObject((HWND)GetClassLong(hWnd, GCL_HBRBACKGROUND));
```

Уничтожить старую кисть необходимо, чтобы не "замусорить" память.

> ***Примечание***
>
> Здесь нам понадобился дескриптор главного окна `hWnd`. Можно перенести его определение на глобальный уровень или воспользоваться функцией `GetParent()`.

Осталось установить новое значение параметра класса окна функцией `SetClassLong()`, эта функция, напротив, позволит изменить свойства класса окна, зарегистрированного в системе:

```
DWORD WINAPI SetClassLong(HWND hWnd, int nIndex, LONG dwNewLong);
```

передавая ей тот же параметр `GCL_HBRBACKGROUND` и дескриптор новой кисти.

```
SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBrush);
```

После чего объявляем окно недействительным для перерисовки фона:

```
InvalidateRect(hWnd, NULL, TRUE);
```

> ***Примечание***
>
> Функция `GetClassLong()` позволяет получить, а `SetClassLong()` установить большинство параметров класса окна, которые определяются значением передаваемого индекса: `GCL_MENUNAME`, `GCL_HBRBACKGROUND`, `GCL_HCURSOR`, `GCL_HICON`, `GCL_HMODULE`, `GCL_CBWNDEXTRA`, `GCL_CBCLSEXTRA`, `GCL_WNDPROC`, `GCL_STYLE`.

Нам осталось обработать сообщение о закрытии окна. Мы убрали кнопку закрытия диалогового окна и воспользовались кнопкой системного меню, которая приводит к тому же результату.

Все остальные сообщения оконная функция диалога игнорирует, возвращая `FALSE`.

Вот собственно и все: запустив программу на выполнение, мы можем регулировать цвет фона, передвигая движки линеек скроллинга для трех базовых цветов.

Стандартное диалоговое окно выбора цвета

Картина будет неполной, если мы не рассмотрим стандартное диалоговое окно выбора цвета, реализованное в Windows. Прототипы функции `ChooseColor()` и структуры `CHOOSECOLOR`, необходимых для создания диалогового окна, как и для всех других диалоговых окон, описаны в файле включений `commdlg.h`, который нужно добавить в заголовке программы.

Для создания диалогового окна создадим пункт меню **StdColor** с идентификатором `ID_STDCOLOR`. При обработке этого пункта меню вызовем диалоговое окно выбора

цвета, которое изображено на рис. 3.11, а код оконной функции, вызывающий это окно, в листинге 3.8.

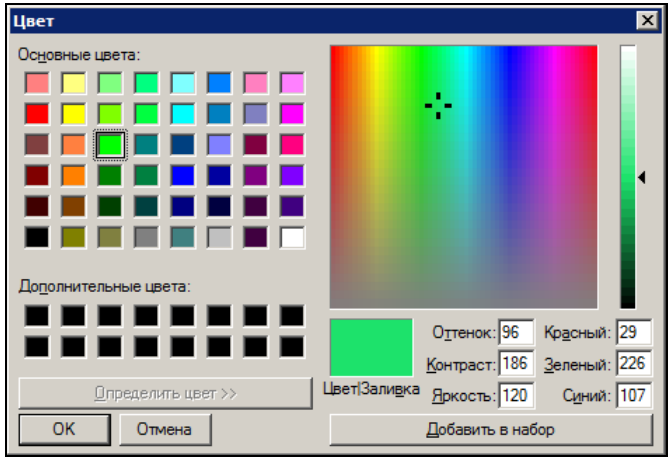


Рис. 3.11. Стандартное диалоговое окно выбора цвета

Листинг 3.8. Оконная функция задачи выбора цвета фона с помощью стандартного диалогового окна

```
#include <commdlg.h>
COLORREF stdColor = RGB(255,255,255);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
    static CHOOSECOLOR ccs;
    static COLORREF acrCustClr[16];
    static HBRUSH hBrush;
    switch (message)
    {
    case WM_CREATE:
      ccs.lStructSize = sizeof(CHOOSECOLOR);
      ccs.hwndOwner = hWnd;
      ccs.rgbResult = stdColor;
      ccs.Flags = CC_RGBINIT | CC_FULLOPEN;
      ccs.lpCustColors = (LPDWORD)acrCustClr;
      break;
    case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_STDCOLOR:
```

```
        if (ChooseColor(&ccs))
        {
           stdColor = ccs.rgbResult;
           if (hBrush) DeleteObject(hBrush);
           hBrush = CreateSolidBrush(stdColor);
           SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBrush);
           InvalidateRect(hWnd, NULL, TRUE);
        }
        break;
     case IDM_EXIT: DestroyWindow(hWnd); break;
     default: return DefWindowProc(hWnd, message, wParam, lParam);
     }
     break;
  case WM_DESTROY: PostQuitMessage(0); break;
  default: return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

Для создания диалогового окна выбора цвета необходимо описать переменную типа структуры `CHOOSECOLOR`, массив `acrCustClr` типа `COLORREF` для хранения 16 дополнительных цветов, а также переменную `color`, которую окно будет использовать для задания начального цвета.

> ***Примечание***
>
> В диалоговом окне выбора цвета имеется возможность кроме основного цвета задать еще 16 дополнительных цветов. Причем даже если они не нужны, массив `acrCustClr` должен быть определен.

Мы задали начальное значение `stdColor = RGB(255, 255, 255)`, чтобы начать выбор с белого цвета.

```
static CHOOSECOLOR ccs;
static COLORREF acrCustClr[16];
```

Здесь же опишем дескриптор кисти.

```
static HBRUSH hBrush;
```

Все переменные класса памяти `static`, чтобы они не потеряли значения при выходе из оконной функции.

Необходимые поля структуры `CHOOSECOLOR` заполняются при обработке сообщения `WM_CREATE`.

```
ccs.lStructSize = sizeof(CHOOSECOLOR); //Задается размер структуры
ccs.hwndOwner = hWnd;                  // Дескриптор родительского окна
ccs.rgbResult = stdColor;              //Начальный цвет диалога
ccs.Flags = CC_RGBINIT | CC_FULLOPEN;
```

Флаг работы `CC_RGBINIT` заставляет диалоговое окно использовать цвет, указанный в `rgbResult`, как начальный цветовой выбор; флаг `CC_FULLOPEN` — отображает дополнительные средства управления для создания дополнительных цветов в массиве `acrCustClr`.

```
ccs.lpCustColors = (LPDWORD)acrCustClr;
```

Диалоговое окно выбора цвета вызывается функцией `ChooseColor()`, аргументом которой является адрес структуры `CHOOSECOLOR`. При удачном результате работы функции она возвращает `TRUE`, а выбранный цвет мы получаем из поля `ccs.rgbResult`.

Теперь, чтобы не оставлять "мусор" в памяти, уничтожим старую кисть:

```
if (hBrush) DeleteObject(hBrush);
```

Оператор `if` используем, чтобы обойти функцию `DeleteObject()` при первом обращении к диалогу, когда дескриптор `hBrush` равен `NULL`.

Создаем новую кисть:

```
hBrush = CreateSolidBrush(stdColor);
```

и используем ее в качестве фонового цвета окна:

```
SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBrush);
```

Осталось только объявить окно недействительным для смены фона:

```
InvalidateRect(hWnd, NULL, TRUE);
```

и задача будет решена.

Вопросы к главе

1. Техника создания окна, аргументы функции `CreateWindow()`.
2. Чем отличаются дочерние и всплывающие окна?
3. Назначение функции `UpdateWindow()`?
4. Создание диалогового окна, функции `DialogBox()` и `CreateDialog()`.
5. Стандартные и общие элементы управления. В чем заключается особенность общих элементов управления?
6. Как осуществляется обмен данными с элементами управления?
7. Приятельские окна для спина.
8. Полоса прокрутки — свойство окна и элемент управления.
9. Как происходит передача управления от немодального окна?
10. Стандартное диалоговое окно выбора цвета, функция `ChooseColor()`.

Задания для самостоятельной работы

1. Написать программу, изображающую шахматную доску, где каждая клетка будет дочернем окном, которое меняет цвет при щелчке левой кнопкой мыши на его поверхности, а при нажатии на правую кнопку появляется всплывающее окно с координатами клетки в шахматной нотации. Всплывающее окно можно создать на элементе управления `STATIC`.
2. Модифицировать программу построения графика (листинг 3.2), предусмотрев блокировку пункта меню **xy-graph** до тех пор, пока не будет загружен файл с данными. После загрузки файла блокировку снять.
3. В программе построения графика предусмотреть вывод координат точки во всплывающем окне после нажатия правой кнопки мыши. При отпускании клавиши мыши окно убрать.

   Указание: воспользуйтесь элементом управления `STATIC`. В случае перекрытия курсора мыши всплывающим окном воспользуйтесь стандартным приемом "захвата мыши".
4. Предусмотрите возможность интерактивного редактирования графика путем "буксировки" точек данных левой кнопкой мыши. После закрытия окна графика сохраните отредактированные данные.
5. Написать тестовую программу для работы со спином при помощи "приятельского" окна **Edit Box**.
6. Написать программу для построения круговой диаграммы в отдельном всплывающем окне. Реализовать диалоговое окно для настройки цветов заполнения.
7. Для предыдущей задачи организовать ввод исходных данных для построения круговой диаграммы в диалоговом окне.
8. Добавить к окну программы просмотра текстовых файлов (листинг 2.3) строку состояния, вывести в нее количество строк текста, максимальную длину строки и полное имя файла.
9. С помощью немодального диалогового окна реализовать функцию поиска слова для программы просмотра текстовых файлов (листинг 2.3). Если слово найдено, обеспечить прокрутку текста до необходимой позиции и выделить найденное слово цветом.
10. Решить предыдущую задачу, используя элемент управления Find.

Глава 4

Растровая графика

В *главе 1* мы рассмотрели технику создания простейших графических примитивов, таких как: точка, линия, прямоугольник, эллипс и т. п. Однако часто возникает необходимость вывести в окне растровое изображение, представленное bmp-файлом, которое состоит из набора точек, каждой из которых сопоставлен битовый образ. Так, для 24-битного изображения точка описывается 3 байтами, где каждый байт задает интенсивность соответствующего цвета — красного, зеленого и синего.

Функция *BitBlt()*

Простейшим вариантом использования растрового изображения является включение его в ресурс приложения. Для этого необходимо на этапе создания проекта импортировать bmp-файл в проект как ресурс Bitmap в меню **Project | Add Resource… | Bitmap | Import…**. Полученному ресурсу автоматически присваивается идентификатор `IDB_BITMAP1`.

Когда приложение начинает работать, изображение загружается функцией `LoadBitmap()`, что обычно происходит при обработке сообщения о создании окна.

```
HBITMAP WINAPI LoadBitmapW(HINSTANCE hInstance,  LPCWSTR lpBitmapName);
```

Первый параметр `hInstance` — дескриптор приложения, второй `lpBitmapName` — идентификатор битового ресурса, который необходимо привести к типу `LPCWTSTR` макросом `MAKEINTRESOURCE()`, например:

```
hBitmap = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_BITMAP1));
```

Функция возвращает дескриптор битового образа `hBitmap`.

После этого можно получить информацию о загруженном изображении обращением к функции `GetObject()`:

```
int WINAPI GetObjectW(HANDLE h, int c, LPVOID pv);
```

Функция принимает дескриптор загруженного изображения `h`, размер `c` и адрес структуры `BITMAP pv`. В этой структуре мы получаем информацию об изображении:

```
struct BITMAP
{   LONG        bmType;      //тип
    LONG        bmWidth;     //ширина изображения в пикселах
    LONG        bmHeight;    //высота изображения в пикселах
```

```
    LONG        bmWidthBytes;//число байтов в строке изображения
    WORD        bmPlanes;    //количество цветов
    WORD        bmBitsPixel; //число битов отображения цвета
    LPVOID      bmBits;      //указатель на область памяти битового образа
};
```

Возвращаемое значение функции — размер битового образа в байтах.

Нас чаще всего интересует содержимое двух полей структуры `BITMAP`, в которых возвращаются размеры загруженного изображения в пикселях — `bmWidth` и `bmHeight`.

После того как мы загрузили изображение и получили его дескриптор, необходимо создать в памяти контекст устройства `memBit`, совместимый с текущим контекстом устройства вывода `hdc`. Для решения этой задачи используется функция `CreateCompatibleDC()`:

```
HDC WINAPI CreateCompatibleDC(HDC hdc);
```

Далее контексту `memBit` вызовом функции `SelectObject()` ставится в соответствие битовый образ.

> ***Примечание***
>
> Контекст памяти образует в оперативной памяти область хранения изображения в том же виде, что и в контексте устройства вывода в окно.

Вывод битового изображения осуществляется обращением к функции `BitBlt()`:

```
BOOL WINAPI BitBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth, int
nHeight, HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop );
```

Функция имеет 9 параметров:

- `hdcDest` — контекст устройства приемника изображения;
- `nXDest`, `nYDest` — *x*, *y*-координата левого верхнего угла приемника;
- `nWidth`, `nHeight` — ширина и высота изображения;
- `hdcSrc` — контекст устройства источника изображения;
- `nXSrc`, `nYSrc` — *x*, *y*-координата левого верхнего угла источника;
- `dwRop` — код растровой операции.

При обработке сообщения `WM_PAINT` получаем контекст устройства вывода `hdc` и вызываем функцию `BitBlt()`, где последним аргументом выбираем операцию `SRCCOPY`, которая осуществляет побитовое копирование изображения из устройства-источника `memBit` в устройство-приемник `hdc`.

```
BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, memBit, 0, 0, SRCCOPY);
```

Другие растровые операции рассмотрим позднее, а сейчас приведем оконную функцию рассмотренной задачи (листинг 4.1).

Листинг 4.1. Вывод в окне растрового изображения из ресурса приложения

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   HBITMAP hBitmap;
   static HDC memBit;
   static BITMAP bm;
   switch (message)
   {
   case WM_CREATE:
      hBitmap = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_BITMAP1));
      GetObject(hBitmap, sizeof(bm), &bm);
      hdc = GetDC(hWnd);
      memBit = CreateCompatibleDC(hdc);
      SelectObject(memBit, hBitmap);
      ReleaseDC(hWnd, hdc);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, memBit, 0, 0, SRCCOPY);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Результат работы программы показан на рис. 4.1. Размеры окна пришлось изменять "вручную", поскольку они никак не связаны с размерами изображения.

Рис. 4.1. Вывод растрового изображения в окне

Вывод изображения в заданный прямоугольник

Растровое изображение в предыдущей задаче можно было бы вывести по размеру созданного окна, растягивая или сжимая исходное изображение, но в этом случае нужно использовать другую функцию `StretchBlt()`:

```
BOOL WINAPI StretchBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth,
                       int nHeight, HDC hdcSrc, int nXSrc, int nYSrc,
                       int nWidthSrc, int nHeightSrc, DWORD dwRop);
```

Отличием этой функции от `BitBlt()` является наличие двух дополнительных параметров `nWidthSrc` и `nHeightSrc`, которые определяют ширину и высоту изображения-источника, остальные параметры совпадают.

Функция `StretchBlt()` осуществляет вывод изображения из источника `hdcSrc` с левым верхним углом `(nXSrc, nYSrc)` шириной `nWidthSrc` и высотой `nHeightSrc` в прямоугольную область приемника `hdcDest` с левым верхним углом `(nXDest, nXDest)` шириной `nWidth` и высотой `nHeight`.

В рассмотренную ранее задачу (листинг 4.1) добавим обработчик сообщения об изменении размеров окна:

```
case WM_SIZE:
       sx = LOWORD(lParam);
       sy = HIWORD(lParam);
       break;
```

Статические переменные `sx, sy` объявим в оконной функции:

```
static int sx, sy;
```

Заменим в сообщении `WM_PAINT` обращение к функции `BitBlt()` на функцию:

```
StretchBlt(hdc,0,0,sx,sy,memBit,0,0,bm.bmWidth,bm.bmHeight,SRCCOPY);
```

Теперь при запуске приложения изображение будет подстраиваться под реальный размер окна.

> ***Примечание***
>
> Изменение размеров растрового изображения приводит к существенной потере качества.

Обычно поступают наоборот — подстраивают размер окна под размер изображения. Это можно сделать, переопределив размер окна в сообщении `WM_SIZE` функцией `MoveWindow()`. Нужно только учесть, что ширина окна должна быть увеличена относительно изображения на ширину бордюра окна, а высота окна включает также высоту заголовка и меню: `caption`, `menu`, `border`. Эти значения можно получить обращением к функции `GetSystemMetrics()`:

```
int WINAPI GetSystemMetrics(int nIndex);
```

с соответствующим значением индекса.

```
caption = GetSystemMetrics(SM_CYCAPTION);
menu = GetSystemMetrics(SM_CYMENU);
border = GetSystemMetrics(SM_CXFIXEDFRAME);
MoveWindow(hWnd, 0, 0, bm.bmWidth + 2*border, bm.bmHeight + caption +
menu + border, TRUE);
```

> ***Примечание***
>
> Если размер изображения превысит размер экрана, необходимо организовать прокрутку изображения "скроллинг".

Загрузка изображения из файла

Не всегда бывает удобно и возможно помещать изображение в ресурс приложения, поэтому в листинге 4.2 рассмотрим задачу о загрузке растрового изображения из bmp-файла во время выполнения приложения.

Листинг 4.2. Загрузка изображения из файла

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static int caption, menu, border;
   static HDC memBit;
   static HBITMAP hBitmap;
   static BITMAP bm;
   switch (message)
```

```
   {
   case WM_CREATE:
      caption = GetSystemMetrics(SM_CYCAPTION);
      menu = GetSystemMetrics(SM_CYMENU);
      border = GetSystemMetrics(SM_CXFIXEDFRAME);
      hBitmap = (HBITMAP)LoadImage(NULL, _T("test.bmp"), IMAGE_BITMAP,
              0, 0, LR_LOADFROMFILE | LR_CREATEDIBSECTION);
      if (hBitmap == NULL)
      {
         MessageBox(hWnd,_T("Файл не найден"),_T("Загрузка изображения"),
                     MB_OK | MB_ICONHAND);
         DestroyWindow(hWnd);
         return 1;
      }
      GetObject(hBitmap, sizeof(bm), &bm);
      hdc = GetDC(hWnd);
      memBit = CreateCompatibleDC(hdc);
      SelectObject(memBit, hBitmap);
      ReleaseDC(hWnd, hdc);
      break;
   case WM_SIZE:
      MoveWindow(hWnd,100, 50, bm.bmWidth+2*border, bm.bmHeight + caption
                     + menu + border, TRUE);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT:  DestroyWindow(hWnd);  break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, memBit, 0, 0, SRCCOPY);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Все, что нам нужно для работы с изображением — это его дескриптор, который можно получить обращением к функции `LoadImage()`:

```
HANDLE WINAPI LoadImageW(HINSTANCE hinst, LPCWTSTR lpszName, INT uType,
int cxDesired, int cyDesired, UINT fuLoad);
```

- `hinst` — дескриптор приложения для загрузки изображения из ресурса, `NULL` — если изображение загружается из файла;
- `lpszName` — идентификатор изображения, если `hinst != NULL`; иначе имя bmp-файла;
- `uType` — тип загружаемого изображения: `IMAGE_BITMAP`, `IMAGE_CURSOR`, `IMAGE_ICON`;
- `cxDesired` — ширина в пикселах для иконки или курсора;
- `cyDesired` — высота в пикселах для иконки или курсора;
- `fuLoad` — флаги чтения изображения: `LR_LOADFROMFILE` — bmp-файл; `LR_CREATEDIBSECTION` — изображение считывается в память в совместимом с текущим дисплеем представлении.

Возвращается дескриптор изображения, но, поскольку функция предназначена для чтения не только растровых изображений, а также иконок и курсоров, то для возвращаемого значения необходимо указывать явное преобразование типа, в нашем случае `(HBITMAP)`.

Предусмотрим случай, когда файл с изображением не найден в текущей папке, тогда функция `LoadImage()` возвращает дескриптор, равный `NULL`. Выводим предупреждающее сообщение функцией `MessageBox()` и завершаем работу `DestroyWindow()`.

Сейчас подойдем к проблеме согласования размеров изображения и окна. Вместо того чтобы менять размер изображения, построим окно, исходя из размеров загруженного изображения (в этом случае в функции `CreateWindow()` необходимо указать нулевые размеры окна).

Размер изображения мы получим обращением к функции `GetObject()`, но для построения окна необходимо учесть наличие заголовка окна, строки меню и бордюра. Эти размеры получим обращением к функции `GetSystemMetrics()`.

После того как вся необходимая информация получена, можно изменить размер окна функцией `MoveWindow()` при обработке сообщения `WM_SIZE`.

> ***Примечание***
>
> Если при создании окна определить стиль рамки `WS_DLGFRAME`, пользователь не сможет изменить размер окна.

Вывести изображение в окно можно функцией `BitBlt()`.

Нетрудно добавить к этой программе диалоговое окно выбора файла, но эту задачу мы оставим для самостоятельной работы.

Растровые операции

Функция `BitBlt()` может работать с 15 различными растровыми операциями, которые осуществляются над каждой точкой изображения:

- `SRCCOPY` — копирование источника в приемник;
- `SRCPAINT` — побитовая операция "ИЛИ" (OR) с предыдущим изображением;
- `SRCAND` — побитовая операция "И" (AND) с предыдущим изображением;
- `SRCINVERT` — побитовая операция "Исключающее ИЛИ" (XOR) с предыдущим изображением;
- `SRCERASE` — побитовая операция "И" (AND) цвета источника с инвертированным цветом приемника;
- `NOTSRCCOPY` — инвертированный цвет источника;
- `NOTSRCERASE` — инверсия цвета, полученного битовой операцией AND над цветом источника и приемника;
- `MERGECOPY` — побитовая операция "И" (AND) цвета источника с текущей кистью;
- `MERGEPAINT` — побитовая операция "ИЛИ" (OR) инвертированного цвета источника с цветом приемника;
- `PATCOPY` — область заполняется текущей кистью;
- `PATPAINT` — побитовая операция "ИЛИ" (OR) текущей кисти и инвертированного цвета источника, после чего побитовая операция "ИЛИ" (OR) с цветом приемника;
- `PATINVERT` — побитовая операция "Исключающее ИЛИ" (XOR) для кисти и цвета приемника;
- `DSTINVERT` — инвертирование цвета приемника;
- `BLACKNESS` — область заполняется черным цветом;
- `WHITENESS` — область заполняется белым цветом.

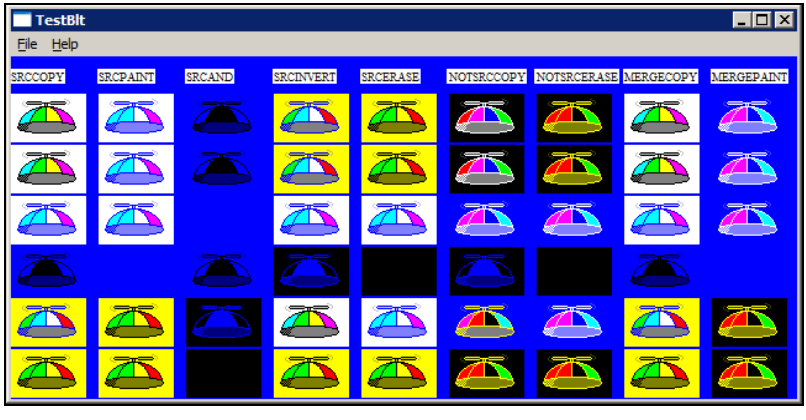


Рис. 4.2. Вывод тестового изображения

Для демонстрации эффектов, возникающих при различных сочетаниях растровых операций, рассмотрим простой пример (листинг 4.3), где мы на синем фоне выведем простое изображение с различными кодами операций, после чего применим к полученному образцу весь набор операций. На рис. 4.2 показан фрагмент окна вывода.

Листинг 4.3. Тест растровых операций для функции `BitBlt()`

```
DWORD Op[15] = {SRCCOPY, SRCPAINT, SRCAND, SRCINVERT, SRCERASE,
      NOTSRCCOPY, NOTSRCERASE, MERGECOPY, MERGEPAINT, PATCOPY,
      PATPAINT, PATINVERT, DSTINVERT, BLACKNESS, WHITENESS};
TCHAR *Name_Op[15]={_T("SRCCOPY"), _T("SRCPAINT"), _T("SRCAND"),
      _T("SRCINVERT"),_T("SRCERASE"), _T("NOTSRCCOPY"), _T("NOTSRCERASE"),
      _T("MERGECOPY"), _T("MERGEPAINT"), _T("PATCOPY"), _T("PATPAINT"),
      _T("PATINVERT"), _T("DSTINVERT"), _T("BLACKNESS"), _T("WHITENESS")};
////////////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   int i, j, x;
   static HFONT hFont;
   static HDC memDC;
   static HBITMAP hPicture;
   static BITMAP bm;
   switch (message)
   {
      case WM_CREATE:
         hPicture = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_BITMAP1));
         GetObject(hPicture, sizeof(bm), &bm);
         hdc = GetDC(hWnd);
         memDC = CreateCompatibleDC(hdc);
         SelectObject(memDC, hPicture);
         ReleaseDC(hWnd, hdc);
         hFont = CreateFont(12,0,0,0,FW_NORMAL,0,0,0,DEFAULT_CHARSET,
               OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,
               DEFAULT_PITCH | FF_DONTCARE, _T("Times New Roman"));
         break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
```

```
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      SelectObject(hdc,hFont);
      for (i = x = 0; i < 15; i++, x += bm.bmWidth + 10)
      {
         TextOut(hdc, x, 10, Name_Op[i], _tcslen(Name_Op[i]));
         BitBlt(hdc, x, 30, bm.bmWidth, bm.bmHeight, memDC, 0, 0, Op[i]);
         for (j = 0; j < 15; j++) BitBlt(hdc, x, 30 + (j + 1)*
            (bm.bmHeight+2), bm.bmWidth, bm.bmHeight, hdc, x, 30, Op[j]);
      }
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      DeleteDC(memDC);
      DeleteObject(hFont);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Набор изображений выводим в строку, последовательно перебирая коды растровых операций из массива `Op`. Для наглядности в заголовке таблицы выводим список операций из массива `Name_Op`, всего таких операций 15.

Следующие строки получаются последовательным применением тех же операций, но в качестве источника используем выведенное в первой строке изображение. Здесь в качестве приемника и источника изображения используется одно и то же устройство `hdc`, отличающееся лишь координатами:

```
BitBlt(hdc, x, 30 + (j + 1)*(bm.bmHeight+2), bm.bmWidth, bm.bmHeight,
hdc, x, 30, Op[j]);
```

Так, координаты начала изображения первой строки таблицы `(x,30)`, а текущей строки `(x, 30 + (j + 1)*(bm.bmHeight + 2))`, где `j` — ее индекс. Мы сдвинули начало вывода на 30 единиц, учитывая выведенную строку заголовка.

Чтобы задать фон окна, мы создали синюю кисть и установили ее в свойствах класса окна до регистрации в системе:

```
wcex.hbrBackground = CreateSolidBrush(RGB(0,0,255));
```

Анимация

Функция `BitBlt()` позволяет более эффективно использовать графическую систему. Можно не перерисовывать каждый раз окно, а изменять лишь небольшой фрагмент изображения, представленный в виде битового образа.

На рис. 4.3 показан пример программы, которая демонстрирует движение упругого шарика в замкнутой области, ограниченной рамкой окна.

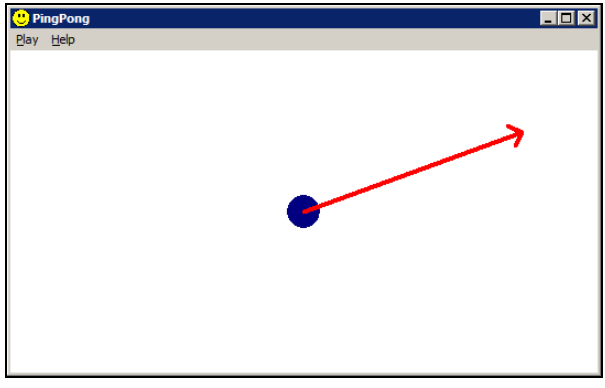


Рис. 4.3. Движение абсолютно упругого шарика с отражением от стенок

Для наглядности скорость движения будем задавать визуально, щелкая мышью в нужном направлении от центра окна, где изображен шарик. Величина скорости будет пропорциональна длине вектора.

Изображение шарика создадим в редакторе ресурсов, для простоты выбирая изображение заданного размера 32×32 пиксела. Фон картинки и окна выберем одинаковым — белым.

Текст оконной функции задачи приведен в листинге 4.4. Еще мы изменили иконку приложения, но ранее мы это уже проделывали и трудностей здесь возникнуть не должно.

Листинг 4.4. Оконная функция задачи "Ping-pong"

```
const int SPAN = 10;
#include <math.h>
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   int mx, my;
   static double mod, vx, vy, xt, yt;
   static HDC memDC;
```

```
static HPEN hpen;
static int x, y, cx, cy, scrx, scry;
static HBITMAP hCircle;
static bool play;
switch (message)
{
case WM_CREATE:
   hpen = CreatePen(PS_SOLID, 4, RGB(255, 0, 0));
   hCircle = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_CIRCLE));
   hdc = GetDC(hWnd);
   memDC = CreateCompatibleDC(hdc);
   SelectObject(memDC, hCircle);
   ReleaseDC(hWnd, hdc);
   break;
case WM_SIZE:
   scrx = LOWORD(lParam);
   scry = HIWORD(lParam);
   x = (cx = scrx/2) - 16;
   y = (cy = scry/2) - 16;
   break;
case WM_LBUTTONDOWN:
   if (!play)
   {  mx = LOWORD(lParam);
      my = HIWORD(lParam);
      vx = mx - cx;
      vy = my - cy;
      mod = sqrt(vx*vx+vy*vy);
      vx = vx/mod;
      vy = vy/mod;
      hdc = GetDC(hWnd);
      SelectObject(hdc, hpen);
      MoveToEx(hdc, cx, cy, 0);
      LineTo(hdc, mx, my);
      LineTo(hdc, mx - (vx - vy)*SPAN, my - (vy + vx)*SPAN);
      MoveToEx(hdc, mx - (vx + vy)*SPAN, my - (vy - vx)*SPAN, 0);
      LineTo(hdc, mx, my);
      ReleaseDC(hWnd, hdc);
      play = true;
    }
    break;
case WM_TIMER:
      hdc = GetDC(hWnd);
      BitBlt(hdc, x, y, 32, 32, NULL, 0, 0, PATCOPY);
```

```
        if (x + 31 > scrx || x < 1) vx = -vx;
        if (y + 31 > scry || y < 1) vy = -vy;
        xt += vx*10;
        yt += vy*10;
        x = int(xt + 0.5);
        y = int(yt + 0.5);
        BitBlt(hdc, x, y, 32, 32, memDC, 0, 0, SRCCOPY);
        ReleaseDC(hWnd, hdc);
        break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_PLAY_BEGIN:
         SetTimer(hWnd, 1,(int)(sqrt(double(cx*cx+cy*cy))/mod)*10 , NULL);
         xt = x;
         yt = y;
         InvalidateRect(hWnd, NULL, TRUE);
         break;
      case ID_PLAY_END:
         KillTimer(hWnd, 1);
         x = cx - 16;
         y = cy - 16;
         InvalidateRect(hWnd, NULL, TRUE);
         play = false;
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
         BitBlt(hdc, x, y, 32, 32, memDC, 0, 0, SRCCOPY);
         EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      DeleteDC(memDC);
      DeleteObject(hpen);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При обработке сообщения WM_CREATE создаем красное перо hpen для прорисовки вектора скорости:

```
hpen = CreatePen(PS_SOLID, 4, RGB(255, 0, 0));
```

Загружаем изображение:

```
hCircle = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_CIRCLE));
```

Создаем контекст устройства в памяти memDC и ставим ему в соответствие загруженное изображение hCircle:

```
hdc = GetDC(hWnd);
memDC = CreateCompatibleDC(hdc);
SelectObject(memDC, hCircle);
ReleaseDC(hWnd, hdc);
```

Начальное значение логической переменной play = false — игра еще не началась.

В сообщении WM_SIZE вычисляем координаты x и y для вывода изображения шарика по центру окна, попутно сохраняя координаты центра окна в переменных cx и cy:

```
scrx = LOWORD(lParam);
scry = HIWORD(lParam);
x = (cx = scrx/2) - 16;
y = (cy = scry/2) - 16;
```

Если скорость движения шарика еще не определена (переменная play == false), в сообщении WM_LBUTTONDOWN вычисляем размер вектора для ее определения. Вектор строится от центра окна до позиции курсора:

```
mx = LOWORD(lParam);
my = HIWORD(lParam);
vx = mx - cx;
vy = my - cy;
mod = sqrt(vx*vx+vy*vy);
```

Нам удобнее работать в относительных единицах, поэтому нормируем координаты вектора:

```
vx = vx/mod;
vy = vy/mod;
```

Мы не случайно определили переменные vx и vy как double, иначе, для целого типа, результатом этой операции был бы 0.

Сейчас можно нарисовать вектор. Для этого получаем контекст устройства hdc, строим линию из центра окна в точку с позицией курсора:

```
hdc = GetDC(hWnd);
SelectObject(hdc, hpen);
MoveToEx(hdc, cx, cy, 0);
LineTo(hdc, mx, my);
```

и прорисовываем наконечник стрелки двумя маленькими линиями, размер которых определяется константой SPAN. Для того чтобы написать эти формулы, необходимы элементарные сведения из аналитической геометрии:

```
LineTo(hdc, mx - (vx - vy)*SPAN, my - (vy + vx)*SPAN);
MoveToEx(hdc, mx - (vx + vy)*SPAN, my - (vy - vx)*SPAN, 0);
LineTo(hdc, mx, my);
```

После чего освобождаем контекст устройства:

```
ReleaseDC(hWnd, hdc);
```

Теперь переменной play присваивается значение true для того, чтобы предотвратить переопределение скорости при случайном щелчке в области окна.

Ритм движения шарика будет определяться таймером, который мы создадим при обработке пункта меню **Begin** с идентификатором ID_PLAY_BEGIN:

```
SetTimer(hWnd, 1, (int)(sqrt(double(cx*cx+cy*cy))/mod)*10, NULL);
```

Третий параметр функции определяет частоту генерации сообщения WM_TIMER и, соответственно, скорость движения.

> ***Примечание***
>
> Нужно иметь в виду, что таймер не может отсчитать интервал времени меньше 58 мсек, поэтому мы подобрали еще и масштабный множитель для ускорения движения.

Скорость движения будет максимальной, если мы зададим максимальный размер вектора, щелкнув в углу окна.

Начальные координаты шарика (x,y) поместим в переменные xt,yt типа double и инициируем перерисовку окна, объявив его недействительным.

Перерисовка окна здесь необходима, чтобы убрать стрелку, показывающую направление и величину скорости. В сообщении WM_PAINT мы ограничиваемся лишь обращением к функции BitBlt() для вывода изображения шарика и перерисовываем фон окна:

```
BitBlt(hdc, x, y, 32, 32, memDC, 0, 0, SRCCOPY);
```

Вся работа по прорисовке движения шарика будет осуществляться при обработке сообщения WM_TIMER. Идея метода заключается в том, что мы в позицию "старого" положения шарика выводим цвет фона, используя растровую операцию PATCOPY:

```
BitBlt(hdc, x, y, 32, 32, NULL, 0, 0, PATCOPY);
```

Таким образом, предыдущее изображение будет "затерто" текущей кистью, т. е. белым цветом. Теперь нужно решить вопрос — а не вышли ли мы за пределы окна, двигаясь в исходном направлении? Мы проверим отдельно x- и y-координаты:

```
if (x + 31 > scrx || x < 1) vx = -vx;
if (y + 31 > scry || y < 1) vy = -vy;
```

Если текущий шаг приводит к пересечению границы окна, меняем знак приращения координаты vx или vy. Несложно убедиться, что отражение от стенки по зако-

нам геометрической оптики требует именно такого преобразования. Теперь наращиваем координаты:

```
xt += vx*10;
yt += vy*10;
```

Мы ввели масштабный коэффициент, равный `10`, для увеличения скорости движения. Результат округляем до целого:

```
x = int(xt + 0.5);
y = int(yt + 0.5);
```

и выводим изображение в новой позиции:

```
BitBlt(hdc, x, y, 32, 32, memDC, 0, 0, SRCCOPY);
```

Так мы обеспечили движение шарика и "отражение" от стенок окна.

Для прекращения демонстрации предусмотрен пункт меню **End** с идентификатором `ID_PLAY_END`. При обработке этого сообщения уничтожаем таймер:

```
KillTimer(hWnd, 1);
```

приводим переменные к исходному положению:

```
x = cx - 16;
y = cy - 16;
play = false;
```

и перерисовываем окно, объявляя его недействительным.

Теперь можно установить новое значение скорости и вновь запустить процесс.

В сообщении `WM_DESTROY` перед завершением работы удаляем перо и контекст устройства в памяти.

Функция *PlgBlt()*

Основным недостатком функции `BitBlt()` является невозможность изменения размеров выводимого изображения и его деформации. Эту задачу решает функция `PlgBlt()`, появившаяся впервые в Windows NT, она осуществляет перенос точек из прямоугольника-источника в заданный параллелограмм-приемник:

```
BOOL WINAPI PlgBlt(HDC hdcDest,CONST POINT *lpPoint,HDC hdcSrc,int nXSrc, int
nYSrc,int nWidth,int nHeight,HBITMAP hbmMask,int xMask,int yMask);
```

- `hdcDest` — дескриптор контекста устройства приемника;
- `lpPoint` — указатель на массив из трех углов параллелограмма;
- `hdcSrc` — дескриптор исходного устройства-источника;
- `nXSrc, nYSrc` — `x`-, `y`-координаты левого верхнего угла исходного прямоугольника;
- `nWidth,nHeight` — ширина и высота исходного прямоугольника;
- `hbmMask` — дескриптор монохромного битового образа-маски;
- `xMask, yMask` — `x,y`-координаты левого верхнего угла маски.

Примечание

При использовании файла-маски нужно учитывать, что это должно быть монохромное изображение такого же размера, что и выводимый рисунок. Точки рисунка, соответствующие черным точкам маски, не отображаются.

Чтобы представить работу новой функции, напишем программу, которая выводит в окно изображение "как есть" и предоставляет возможность изменять координаты параллелограмма-приемника. Для наглядности будем изменять координаты трех его углов, "буксируя" их мышью в нужную позицию (очевидно, что для однозначного задания параллелограмма достаточно задать только три его угла). Результат работы программы изображен на рис. 4.4.

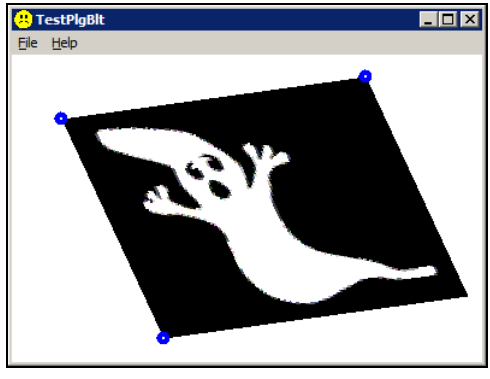


Рис. 4.4. Деформация изображения функцией `PlgBlt()`

Углы прямоугольника, задаваемые в массиве `lpPoint`, для наглядности выделили небольшими кружками.

Текст оконной функции приведен в листинге 4.5. Демонстрационный bmp-файл мы импортируем в ресурсы приложения.

Листинг 4.5. Демонстрационная задача для функции `PlgBlt()`

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   RECT rt;
   int i, x, y, p, q;
   static int k;
   static bool Capture;
   static POINT pts[3];
   static HDC memDC;
   static HBITMAP hPicture;
```

```
static BITMAP bm;
static HPEN hPen;
switch (message)
{
   case WM_CREATE:
      hPicture = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_SPOOK));
      GetObject(hPicture, sizeof(bm), &bm);
      hPen = CreatePen(PS_SOLID, 4, RGB(0, 0, 255));
      GetClientRect(hWnd, &rt);
      x = (rt.right - bm.bmWidth)/2;
      y = (rt.bottom - bm.bmHeight)/2;
      pts[0].x = pts[2].x = x;
      pts[0].y = pts[1].y = y;
      pts[1].x = x + bm.bmWidth;
      pts[2].y = y + bm.bmHeight;
      hdc = GetDC(hWnd);
      memDC = CreateCompatibleDC(hdc);
      SelectObject(memDC, hPicture);
      ReleaseDC(hWnd, hdc);
      break;
case WM_COMMAND:
   switch (LOWORD(wParam))
   {
   case IDM_EXIT: DestroyWindow(hWnd); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   break;
   case WM_LBUTTONDOWN:
      x = LOWORD(lParam);
      y = HIWORD(lParam);
      for (k = 0; k < 3; k++)
      {
         p = x - pts[k].x;
         q = y - pts[k].y;
         if (p*p + q*q < 16)
         {
            SetCapture(hWnd);
            Capture = true;
            return 0;
         }
      }
      break;
   case WM_MOUSEMOVE:
      if (Capture)
      {
```

```
            pts[k].x = LOWORD(lParam);
            pts[k].y = HIWORD(lParam);
            InvalidateRect(hWnd, NULL, TRUE);
         }
         break;
      case WM_LBUTTONUP:
         if (Capture)
         {
            ReleaseCapture();
            Capture = false;
         }
         break;
      case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);
         PlgBlt(hdc, pts, memDC, 0, 0, bm.bmWidth, bm.bmHeight, 0, 0, 0);
         SelectObject(hdc, (HPEN)hPen);
         for (i = 0; i < 3; i++)
            Ellipse(hdc, pts[i].x-4, pts[i].y-4, pts[i].x+4, pts[i].y+4);
         EndPaint(hWnd, &ps);
         break;
      case WM_DESTROY:
         DeleteDC(memDC);
         DeleteObject(hPen);
         PostQuitMessage(0);
         break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При обработке сообщения `WM_CREATE` мы, как и в предыдущей задаче, загрузим картинку и сохраним ее дескриптор в статической переменной:

```
hPicture = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_SPOOK));
```

Определим параметры изображения, помещая их в структуру `BITMAP bm`:

```
GetObject(hPicture, sizeof(bm), &bm);
```

Нам нужно создать перо для выделения углов параллелограмма:

```
hPen = CreatePen(PS_SOLID, 4, RGB(0, 0, 255));
```

После этого определим размер клиентской области окна для вывода изображения по центру:

```
GetClientRect(hWnd, &rt);
x = (rt.right - bm.bmWidth)/2;
y = (rt.bottom - bm.bmHeight)/2;
```

Координаты `x` и `y` будут определять верхний левый угол изображения.

Примечание

Хотя при обработке сообщения WM_CREATE окно еще не отображается, его размеры уже известны системе и могут быть получены функцией GetClientRect().

Вначале задаем прямоугольник с исходными размерами изображения, определяя массив pts, где 3 элемента типа POINT — координаты углов параллелограмма-приемника:

```
pts[0].x = pts[2].x = x;
pts[0].y = pts[1].y = y;
pts[1].x = x + bm.bmWidth;
pts[2].y = y + bm.bmHeight;
```

Теперь осталось получить контекст в памяти memDC, совместимый с контекстом окна, и связать его с загруженным изображением hPicture:

```
memDC = CreateCompatibleDC(hdc);
SelectObject(memDC, hPicture);
```

При обработке сообщения о нажатии левой кнопки мыши WM_LBUTTONDOWN мы должны определить, указывает ли курсор мыши на угол параллелограмма и на какой именно. Поступим следующим образом — в цикле по трем точкам (углам параллелограмма) измерим расстояние между координатами точек и указателем мыши:

```
x = LOWORD(lParam);
y = HIWORD(lParam);
for (k = 0; k < 3; k++)
{ p = x - pts[k].x;
  q = y - pts[k].y;
  if (p*p + q*q < 16)
```

Если расстояние не превышает 4 пикселов (квадрат расстояния, соответственно, 16), считаем, что точка выбрана, производим *"захват мыши"*, присваиваем переменной Capture значение true и завершаем обработку сообщения:

```
SetCapture(hWnd);
Capture = true;
return 0;
```

Можно было бы обойтись и без *"захвата мыши"*, но тогда при выходе курсора за пределы окна оконная функция перестанет получать сообщения мыши, что вряд ли хорошо (мы уже обсуждали эту проблему при построении кривой Безье, листинг 1.15). Функция SetCapture() обеспечит пересылку всех сообщений мыши приложению, "захватившему мышь", даже при выходе курсора за пределы окна, а возвращаемые координаты курсора мыши будут определяться относительно клиентской области окна. Однако необходимо помнить, что координаты могут быть отрицательными.

Примечание

Прием с "захватом мыши" является стандартным в подобной ситуации.

Рассмотрим обработку сообщения о перемещении мыши `WM_MOUSEMOVE`. Сначала мы должны определить, был ли "захвачен" один из углов параллелограмма. Если это не так, переменная `Capture` имеет значение `false`, и обработка сообщения прекращается. Если же угол параллелограмма "захвачен", считываем текущие координаты мыши в соответствующий элемент массива `pts` (его индекс сохранился в статической переменной `k`):

```
pts[k].x = LOWORD(lParam);
pts[k].y = HIWORD(lParam);
```

и инициируем перерисовку окна функцией `InvalidateRect()`.

При сообщении об отпускании левой кнопки мыши `WM_LBUTTONUP`, если мышь была "*захвачена*", мы должны ее "*освободить*" функцией без параметров `ReleaseCapture()` и присвоить значение `false` логической переменной `Capture`.

> ***Примечание***
>
> Нужно очень осторожно обращаться с операцией *"захвата мыши"* и обязательно *"освобождать"* мышь. Обычная практика такова — мышь *"захватывается"* по нажатию кнопки, а *"освобождается"* при ее отпускании.

Осталось рассмотреть обработку сообщения `WM_PAINT`. Получаем контекст устройства `hdc` и выводим изображение:

```
PlgBlt(hdc, pts, memDC, 0, 0, bm.bmWidth, bm.bmHeight, 0, 0, 0);
```

из контекста памяти `memDC`, начиная с левого верхнего угла `(0,0)`, размером `bm.bmWidth, bm.bmHeight`, в параллелограмм с координатами, указанными в массиве `pts`, на устройстве `hdc`.

Маску изображения мы не использовали, поэтому три последних параметра функции имеют нулевое значение.

После чего выберем созданное ранее перо:

```
SelectObject(hdc, (HPEN)hPen);
```

и нарисуем три маленьких окружности радиуса 4 единицы вокруг углов параллелограмма:

```
for (i = 0; i < 3; i++)
    Ellipse(hdc, pts[i].x-4, pts[i].y-4, pts[i].x+4, pts[i].y+4);
```

Остальные сообщения в этой программе обрабатываются стандартным образом, нужно только не забыть освободить контекст в памяти `memDC` и перо `hPen` при завершении работы в сообщении `WM_DESTROY`.

Функция *MaskBlt()*

Таким же нововведением в Windows NT стала функция `MaskBlt()`, позволяющая маскировать одно изображение другим:

```
BOOL WINAPI MaskBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth,
                    int nHeight, HDC hdcSrc, int nXSrc, int nYSrc,
                    HBITMAP hbmMask, int xMask, int yMask, DWORD dwRop);
```

- `hdcDest` — дескриптор контекста устройства приемника;
- `nXDest`, `nYDest` — *x*, *y*-координата левого верхнего угла приемника;
- `nWidth`, `nHeight` — ширина и высота изображения;
- `hdcSrc` — контекст устройства источника изображения;
- `nXSrc`, `nYSrc` — *x*, *y*-координата левого верхнего угла источника;
- `hbmMask` — дескриптор монохромного битового образа-маски;
- `xMask`, `yMask` — *x*, *y*-координаты левого верхнего угла маски;
- `dwRop` — код растровой операции, аналогичен функции `BitBlt()`.

Полный набор растровых операций позволяет получить различные эффекты с использованием маскирующего изображения. Причем, в отличие от функции `PlgBlt()`, здесь изображение-маска накладывается на исходное изображение с соответствующим эффектом, определенным выбранной растровой операцией.

> ***Примечание***
>
> Функция `MaskBlt()` используется редко, поскольку практически все эффекты могут быть достигнуты в функции `BitBlt()`, к тому же, в ее реализации была сделана ошибка, и функция работает правильно только тогда, когда размер маски точно совпадает с размером изображения.

В качестве примера рассмотрим достаточно традиционную задачу (листинг 4.6 и рис. 4.5): отобразим в окне сигарету, а при нажатии левой кнопки мыши перечеркнем изображение стандартным знаком **Ø**; при отпускании кнопки изображение должно восстанавливаться. Импортируем два изображения в ресурс приложения с идентификаторами `IDB_SMOKES` и `IDB_NO`.

Листинг 4.6. Тест для функции `MaskBlt()`

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static int x, y;
   static HDC memDC;
   static HBITMAP hSmokes, hNo;
   static BITMAP bm;
   switch (message)
   {
   case WM_CREATE:
      hNo = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_NO));
      hSmokes = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_SMOKES));
      GetObject(hSmokes, sizeof(bm), &bm);
      hdc = GetDC(hWnd);
      memDC = CreateCompatibleDC(hdc);
```

```
      SelectObject(memDC, hSmokes);
      ReleaseDC(hWnd, hdc);
      break;
   case WM_SIZE:
      x = (LOWORD(lParam) - bm.bmWidth)/2;
      y = (HIWORD(lParam) - bm.bmHeight)/2;
      break;
   case WM_LBUTTONDOWN:
      hdc = GetDC(hWnd);
      MaskBlt(hdc,x,y,bm.bmWidth,bm.bmHeight,memDC,0,0,hNo,0,0,SRCCOPY);
      ReleaseDC(hWnd, hdc);
      break;
   case WM_LBUTTONUP:
      InvalidateRect(hWnd, NULL, TRUE);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      BitBlt(hdc, x, y, bm.bmWidth, bm.bmHeight, memDC, 0, 0, SRCCOPY);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

В оконной функции опишем переменные целого типа `x,y` для хранения координат вывода изображения по центру окна, контекст устройства в памяти `memDC`, а также дескрипторы изображения `hSmokes` и маски `hNo`. Переменная `bm` типа `BITMAP` необходима для определения размеров загруженного изображения.

При создании окна в сообщении `WM_CREATE` загрузим из ресурса изображение и маску, получим их дескрипторы:

```
hNo = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_NO));
hSmokes = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_SMOKES));
```

Получаем информацию об изображении:

```
GetObject(hSmokes, sizeof(bm), (LPSTR)&bm);
```

Создаем совместимый контекст устройства в памяти и выбираем для него загруженное изображение:

```
memDC = CreateCompatibleDC(hdc);
SelectObject(memDC, hSmokes);
```

В сообщении `WM_SIZE` вычисляем координаты для вывода изображения:

```
x = (LOWORD(lParam) - bm.bmWidth)/2;
y = (HIWORD(lParam) - bm.bmHeight)/2;
```

Функцией `BitBlt()` прорисовываем изображение в сообщении `WM_PAINT`:

```
BitBlt(hdc, x, y, bm.bmWidth, bm.bmHeight, memDC, 0, 0, SRCCOPY);
```

А при нажатии левой кнопки мыши в сообщении `WM_LBUTTONDOWN` перерисуем изображение, наложив на него маску (см. рис. 4.5):

```
MaskBlt(hdc,x,y,bm.bmWidth,bm.bmHeight,memDC,0,0,hNo,0,0,SRCCOPY);
```

Здесь мы выбрали растровую операцию копирования.

При отпускании кнопки мыши восстановим исходное изображение, перерисовывая окно.

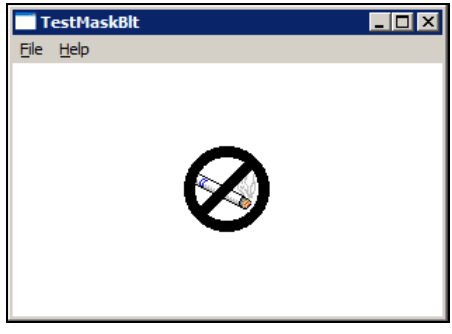


Рис. 4.5. Демонстрация работы функции `MaskBlt()`

Вращение графического образа

Еще одна функция `SetWorldTransform()`, реализованная в Windows NT, позволяет производить вращение изображения. Строго говоря, функция обеспечивает поворот не самого изображения, а системы координат:

```
BOOL WINAPI SetWorldTransform(HDC hdc, CONST XFORM *lpXform);
```

Здесь `hdc` — контекст устройства, `lpXform` — структура, содержащая данные для преобразования координат.

```
struct XFORM {FLOAT eM11, eM12, eM21, eM22, eDx, eDy;};
```

Формулы преобразования координат имеют следующий вид:

```
x' = x * eM11 + y * eM21 + eDx
y' = x * eM12 + y * eM22 + eDy
```

Из аналитической геометрии нам известны формулы преобразования координат:

$$x' = x_0 + x \cdot \cos(\phi) - y \cdot \sin(\phi)$$
$$y' = y_0 + x \cdot \sin(\phi) + y \cdot \cos(\phi)$$

Из сравнения этих формул следует, что первые четыре параметра структуры — это косинусы и синусы угла поворота, а два последних — сдвиг системы координат.

Для демонстрации работы функции `SetWorldTransform()` напишем программу (листинг 4.7 и рис. 4.6), где отобразим небольшую картинку из bmp-файла, над которой поместим текст. При каждом нажатии на левую кнопку мыши обеспечим поворот на 45°.

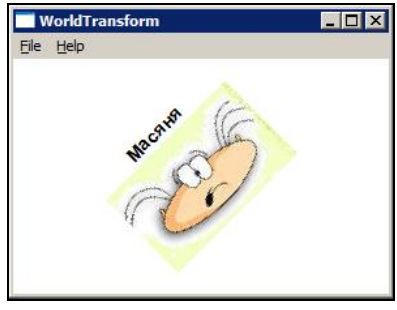


Рис. 4.6. Вращение мировых координат

Листинг 4.7. Демонстрация вращения графического образа

```
#define _USE_MATH_DEFINES
#include <math.h>
TCHAR *text = _T("Масяня");
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static int x, y, cx, cy;
   static double angle;
   static HDC memBit;
   static HBITMAP hBitmap;
   static BITMAP bm;
   static XFORM xf;
   switch (message)
   {
   case WM_CREATE:
      hBitmap = (HBITMAP)LoadImage(NULL, _T("mas1.bmp"), IMAGE_BITMAP, 0,
```

```
                    0, LR_LOADFROMFILE | LR_CREATEDIBSECTION);
      GetObject(hBitmap, sizeof(bm), &bm);
      hdc = GetDC(hWnd);
      memBit = CreateCompatibleDC(hdc);
      SelectObject(memBit, hBitmap);
      ReleaseDC(hWnd, hdc);
      break;
   case WM_SIZE:
      cx = LOWORD(lParam)/2; //Координаты центра окна
      cy = HIWORD(lParam)/2;
      xf.eDx = cx;    //Точка будет являться и новым началом
      xf.eDy = cy;    //координат для графических функций
      x = - bm.bmWidth/2;
      y = - bm.bmHeight/2;
      break;
   case WM_LBUTTONDOWN: angle += M_PI*45.0/180.0;
      InvalidateRect(hWnd, NULL, TRUE);
      break;
   case WM_PAINT:
      xf.eM22 = xf.eM11 = cos(angle);
      xf.eM12 = -(xf.eM21 = sin(angle));
      hdc = BeginPaint(hWnd, &ps);
      SetGraphicsMode(hdc, GM_ADVANCED);
      SetWorldTransform(hdc, &xf);
      TextOut(hdc,x+35, y-20, text, _tcsclen(text));
      BitBlt(hdc, x, y, bm.bmWidth, bm.bmHeight, memBit, 0, 0, SRCCOPY);
      EndPaint(hWnd, &ps);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Картинку загружаем функцией `LoadImage()` из файла `mas1.bmp` при обработке сообщения о создании окна `WM_CREATE`. Здесь же создаем контекст в памяти, совместимый с контекстом устройства вывода, так же, как мы делали в предыдущих задачах.

В сообщении WM_SIZE определяем размеры окна и заполняем поля eDx и eDy структуры XFORM. *Эта точка будет началом новой системы координат*.

При обработке сообщения WM_LBUTTONDOWN увеличим угол поворота системы координат angle на 45° и инициируем перерисовку окна функцией InvalidateRect(). Поскольку переменная angle — статическая, начальное значение угла 0°.

В сообщении WM_PAINT определяем остальные поля структуры XFORM (они определены в библиотеке как FLOAT). Далее получаем контекст устройства вывода, обеспечиваем поворот системы координат функцией SetWorldTransform(), выводим текст и графику, учитывая, что начало координат сейчас находится в центре окна.

> ***Примечание***
>
> Функция реализована только для операционной системы Windows NT и выше и работает только при включенном режиме GM_ADVANCED (продвинутый графический режим), поэтому предварительно нужно вызвать функцию SetGraphicsMode(hdc, GM_ADVANCED).

Виртуальное окно

Нередко возникают ситуации, когда обновление окна либо требует больших вычислений, либо же просто невозможно традиционными способами. В этом случае используют так называемые *"виртуальные окна"*. Идея заключается в том, что в памяти создается контекст, совместимый с контекстом окна, куда и отправляется весь вывод. Когда же необходимо обновить окно, содержимое виртуального окна копируется в окно вывода функцией BitBlt().

Реализация этой методики похожа на ту, что мы использовали при отображении растрового изображения, только вместо готовой картинки мы помещаем в контекст памяти растровое изображение заданного размера, совместимое с контекстом устройства, которое создается функцией CreateCompatibleBitmap():

```
HBITMAP WINAPI CreateCompatibleBitmap(HDC hdc, int width, int height);
```

Для реализации этой задачи понадобится еще одна функция PatBlt():

```
BOOL WINAPI PatBlt(HDC hdc, int x, int y, int width, int height,
                   DWORD dwRaster);
```

которая закрашивает прямоугольную область с координатами (x,y), шириной width и высотой height, используя текущую кисть и растровую операцию dwRaster:

- PATCOPY — область заполняется текущей кистью;
- PATINVERT — логическая операция "Исключающее ИЛИ" (XOR) для цвета кисти и области заполнения;
- DSTINVERT — инвертирование цвета области заполнения;
- BLACKNESS — область заполняется черным цветом;
- WHITENESS — область заполняется белым цветом.

Рассмотрим в качестве примера задачу о заполнении окна "случайными" прямоугольниками "случайного" же цвета (рис. 4.7, листинг 4.8). Очевидно, что для пе-

рерисовки такого окна было бы необходимо повторить весь вывод с самого начала, а в случае "случайного" расположения объектов в окне решение задачи становится весьма проблематичным. Поэтому мы направим весь вывод в контекст памяти, а для отображения в окне будем осуществлять копирование изображения из памяти.

Рис. 4.7. Случайные прямоугольники

Листинг 4.8. Демонстрация работы виртуального окна

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   int r, g, b, maxX, maxY;
   static int sx, sy;
   static HDC memBit;
   HBRUSH hBrush;
   RECT rt;
   HBITMAP hBitmap;
   SYSTEMTIME tm;
   switch (message)
   {
   case WM_CREATE:
      SetTimer(hWnd, 1, 100, NULL);
      maxX = GetSystemMetrics(SM_CXSCREEN);
      maxY = GetSystemMetrics(SM_CYSCREEN);
      hdc = GetDC(hWnd);
      memBit = CreateCompatibleDC(hdc);
```

```
      hBitmap = CreateCompatibleBitmap(hdc, maxX, maxY);
      SelectObject(memBit, hBitmap);
      PatBlt(memBit, 0, 0, maxX, maxY, WHITENESS);
      ReleaseDC(hWnd, hdc);
      GetSystemTime(&tm);
      srand(tm.wMilliseconds);
      break;
   case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      break;
   case WM_TIMER:
      rt.right = (rt.left = rand()*sx/RAND_MAX) + rand()*sx/RAND_MAX/2;
      rt.top = (rt.bottom = rand()*sy/RAND_MAX) - rand()*sy/RAND_MAX/2;
      r = rand()*255/RAND_MAX;
      g = rand()*255/RAND_MAX;
      b = rand()*255/RAND_MAX;
      hBrush = CreateSolidBrush(RGB(r,g,b));
      FillRect(memBit, &rt, hBrush);
      DeleteObject(hBrush);
      InvalidateRect(hWnd, NULL, 0);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      BitBlt(hdc, 0, 0, sx, sy, memBit, 0, 0, SRCCOPY);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      DeleteObject(memBit);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Заполнение окна будем осуществлять при обработке сообщений таймера, который создадим в сообщении `WM_CREATE`:

```
SetTimer(hWnd, 1, 100, NULL);
```

Здесь же определим максимальный размер окна, вычисляя горизонтальный и вертикальный размер экрана в пикселах:

```
maxX = GetSystemMetrics(SM_CXSCREEN);
maxY = GetSystemMetrics(SM_CYSCREEN);
```

Такой размер обеспечит необходимый объем памяти при "распахивании окна" на весь экран.

> ***Примечание***
>
> На самом деле мы задаем несколько больший размер, поскольку клиентская часть окна будет меньше на размер бордюра, заголовка окна и строки меню.

Теперь, определив контекст устройства вывода `hdc`, создадим совместимый контекст в памяти `memBit`:

```
memBit = CreateCompatibleDC(hdc);
```

Создадим изображение в памяти заданного размера:

```
hBitmap = CreateCompatibleBitmap(hdc, maxX, maxY);
```

После чего выбираем это изображение для контекста памяти:

```
SelectObject(memBit, hBitmap);
```

и закрашиваем область белой кистью:

```
PatBlt(memBit, 0, 0, maxX, maxY, WHITENESS);
```

Здесь же задаем стартовую точку генерации случайных чисел, используя в качестве аргумента функции `srand()` системное время, полученное обращением к функции `GetSystemTime()`:

```
VOID WINAPI GetSystemTime(LPSYSTEMTIME lpSystemTime);
```

В сообщении `WM_SIZE` получим размеры клиентской области `(sx,sy)`.

Вывод в контекст памяти будем осуществлять в сообщении `WM_TIMER`, где координаты выводимого прямоугольника `rt`, а также его цвет `RGB(r,g,b)` определены случайным образом. Определив кисть соответствующего цвета:

```
hBrush = CreateSolidBrush(RGB(r,g,b));
```

закрасим прямоугольник:

```
FillRect(memBit, &rt, hBrush);
```

после чего кисть уничтожаем, дабы не "мусорить в памяти":

```
DeleteObject(hBrush);
```

и перерисовываем окно, объявляя его недействительным:

```
InvalidateRect(hWnd, NULL, 0);
```

> ***Примечание***
>
> Здесь гораздо эффективнее использовать для закрашивания прямоугольника функцию `FillRect()`, поскольку нет необходимости устанавливать кисть в качестве текущей.

Нет смысла перерисовывать фон окна, поэтому последний параметр функции равен `0` и в сообщении `WM_PAINT` просто копируем изображение из контекста памяти в окно:

```
BitBlt(hdc, 0, 0, cx, cy, memBit, 0, 0, SRCCOPY);
```

Метафайлы

Если в bmp-файлах изображения хранятся в виде битовых образов, то метафайлы содержат набор инструкций по построению изображения, точнее, в них записывается последовательность вызовов графических функций. Поэтому для изображения, воспроизводимого из метафайла, изменение масштаба не приводит к потере качества, поскольку изображение строится каждый раз заново в новой системе координат. Большую часть графических функций можно использовать в метафайлах, и, казалось бы, они относятся скорее к векторной графике, однако возможно использование и растровых изображений, правда, с некоторыми оговорками. Рассмотрим вначале простейшие варианты построения метафайлов, а затем перейдем к их расширениям.

Для создания метафайла необходимо вызвать функцию `CreateMetaFile()`:

```
HDC WINAPI CreateMetaFileW(LPCWSTR lpszFile);
```

с одним параметром `lpszFile` — это имя файла с расширением wmf (*Windows Metafile)*. Если вместо имени задать `NULL`, файл создается в памяти.

Возвращаемым значением функции является контекст устройства, куда и направляется вывод графических функций. В этом случае изображение в окно не выводится, а последовательность обращений к GDI-функциям записывается в метафайл. По завершении записи необходимо закрыть метафайл обращением к функции `CloseMetaFile()`:

```
HMETAFILE WINAPI CloseMetaFile(HDC hdc);
```

Возвращаемым значением будет дескриптор метафайла, который используется в функции `PlayMetaFile()` для его *"проигрывания"*.

```
BOOL WINAPI PlayMetaFile(HDC hdc, HMETAFILE hmf);
```

Рассмотрим простой пример построения графика в виде метафайла в памяти и его тиражирование в окне (листинг 4.9 и рис. 4.8).

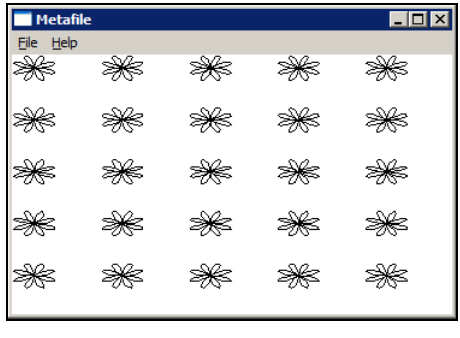


Рис. 4.8. Построение изображения из метафайла

Листинг 4.9. Создание метафайла в памяти

```
#define _USE_MATH_DEFINES
#include <math.h>
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc, hdcMetafile;
   static HMETAFILE hmf;
   static int sx, sy;
   int x, y;
   double angle;
   switch (message)
   {
   case WM_CREATE:
      hdcMetafile = CreateMetaFile(NULL);
      for (x = y = 50, angle = 0.0; angle < 2*M_PI; angle += 0.1)
      {
         MoveToEx(hdcMetafile, x, y, NULL);
         x = int(50.0*cos(angle)*sin(angle*4.0)+50.5);
         y = int(50.0*sin(angle)*sin(angle*4.0)+50.5);
         LineTo(hdcMetafile, x, y);
      }
      hmf = CloseMetaFile(hdcMetafile);
      break;
   case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      SetMapMode(hdc, MM_ANISOTROPIC);
      SetWindowExtEx(hdc, 1000, 1000, NULL);
      SetViewportExtEx(hdc, sx, sy, NULL);
      for(x = 0; x < 5; x++)
```

```
        for(y = 0; y < 5; y++)
           {
             SetWindowOrgEx(hdc, -200 * x, -200 * y, NULL);
             PlayMetaFile(hdc, hmf);
           }
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      DeleteMetaFile(hmf);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Для построения геометрической фигуры нам понадобится математическая библиотека `math.h`. При обработке сообщения `WM_CREATE` создадим метафайл в памяти:

```
hdcMeta = CreateMetaFile(NULL);
```

В цикле `for` по углу `angle` от `0.0` до `2*M_PI` с шагом `0.1` построим фигуру размером 100×100 логических единиц ломаной линией, используя в качестве устройства вывода контекст созданного метафайла `hdcMeta`.

После чего закроем метафайл и получим его дескриптор:

```
hmf = CloseMetaFile(hdcMeta);
```

В сообщении `WM_SIZE` получим клиентские размеры окна `sx,sy`.

А при обработке сообщения `WM_PAINT`, получив контекст устройства `hdc`, установим режим `MM_ANISOTROPIC` и логические размеры окна:

```
SetWindowExtEx(hdc, 1000, 1000, NULL);
```

Физические размеры области вывода установим по размеру окна:

```
SetViewportExtEx(hdc, sx, sy, NULL);
```

Теперь для вывода изображения в виде таблицы размером 5×5 организуем двойной цикл, где на каждом шаге будем менять начало координат, указывая на левый верхний угол прямоугольной области размером 200×200 (здесь нужно задавать отрицательные координаты), и выводить изображение в эту область, *проигрывая* метафайл функцией `PlayMetaFile()`:

```
SetWindowOrgEx(hdc, -200 * x, -200 * y, NULL);
PlayMetaFile(hdc, hmf);
```

Таким образом, мы получим в окне 25 изображений, хранящихся в метафайле.

Остается только при закрытии окна `WM_DESTROY` освободить ресурсы, занимаемые метафайлом: `DeleteMetaFile(hmf)`:

```
BOOL WINAPI DeleteMetaFile(HMETAFILE hmf);
```

Создание дискового файла

Кроме этого имеется возможность сохранить метафайл в виде дискового файла, который может быть просмотрен в графическом редакторе. Для демонстрации этой возможности внесем некоторые изменения в листинг 4.9.

При создании метафайла на диске мы должны указать его имя:

```
hdcMetafile = CreateMetaFile(_T("graph.wmf"));
```

Теперь нет необходимости запоминать дескриптор метафайла `hmf` при его закрытии:

```
CloseMetaFile(hdcMetafile);
```

В сообщении `WM_PAINT` обращением к функции `GetMetaFile()`:

```
HMETAFILE WINAPI GetMetaFileW(LPCWSTR lpName);
```

получим дескриптор метафайла:

```
hmf = GetMetaFile(_T("graph.wmf"));
```

Функция имеет один параметр — имя метафайла (файл создаем в текущей папке и читаем также из текущей папки).

Перед выходом из обработчика сообщения `WM_PAINT` необходимо освободить ресурсы, занятые метафайлом:

```
DeleteMetaFile(hmf);
```

Результатом выполнения этой программы будет та же картинка, что мы получили в предыдущем примере (см. рис. 4.8), однако здесь метафайл сохранится на диске.

Растровое изображение в метафайле

Хотя это и не приветствуется по причине расточительного расходования памяти, рассмотрим возможность записи растрового изображения в метафайл. Для этого построим новый пример (листинг 4.10), где считаем картинку из ресурса приложения, поместим ее в метафайл памяти, а затем *проиграем* для вывода в окно.

> ***Примечание***
>
> Растровое изображение можно поместить только в метафайл памяти. Для вывода его в дисковый файл потребуется формат *расширенного метафайла*, который мы рассмотрим далее.

Листинг 4.10. Создание метафайла в памяти для растрового изображения

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc, hdcMeta, memBit;
   static HMETAFILE hmf;
   HBITMAP hBitmap;
```

```
    BITMAP bm;
    switch (message)
    {
    case WM_CREATE:
      hdcMeta = CreateMetaFile(NULL);
      hBitmap = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_MOUSE));
      GetObject(hBitmap, sizeof(bm), &bm);
      hdc = GetDC(hWnd);
      memBit = CreateCompatibleDC(hdc);
      ReleaseDC(hWnd, hdc);
      SelectObject(memBit, hBitmap);
      BitBlt(hdcMeta, 0, 0, bm.bmWidth, bm.bmHeight, memBit,0,0,SRCCOPY);
      hmf = CloseMetaFile(hdcMeta);
      break;
    case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
    case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      PlayMetaFile(hdc, hmf);
      EndPaint(hWnd, &ps);
      break;
    case WM_DESTROY:
      DeleteMetaFile(hmf);
      PostQuitMessage(0);
      break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

При создании окна в сообщении `WM_CREATE` получим дескриптор метафайла в памяти:

```
hdcMeta = CreateMetaFile(NULL);
```

Загрузим изображение из ресурса приложения (для этого импортируем картинку в виде bmp-файла на этапе разработки) и получим ее размеры:

```
hBitmap = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_MOUSE));
GetObject(hBitmap, sizeof(bm), &bm);
```

Создадим совместимый контекст в памяти:

```
memBit = CreateCompatibleDC(hdc);
```

Выберем изображение для этого контекста

```
SelectObject(memBit, hBitmap);
```

и выведем полученное изображение в метафайл:

```
BitBlt(hdcMeta, 0, 0, bm.bmWidth, bm.bmHeight, memBit,0,0,SRCCOPY);
```

При закрытии метафайла получим его дескриптор:

```
hmf = CloseMetaFile(hdcMeta);
```

Поскольку дескриптор метафайла `hmf` определен в статической переменной, мы можем использовать его в сообщении `WM_PAINT` для *проигрывания*:

```
PlayMetaFile(hdc, hmf);
```

Результат работы приложения получим как показано на рис. 4.9.

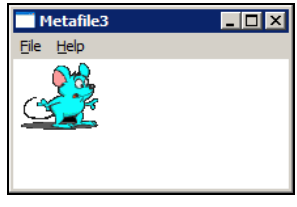


Рис. 4.9. Вывод растрового изображения из метафайла

Расширенные метафайлы

Метафайлы, рассмотренные ранее, представляют устаревший формат хранения изображений, поскольку имеют некоторые проблемы при их воспроизведении. Они не хранят информации о графическом режиме, поэтому размеры изображения могут определяться неоднозначно, а также не позволяют хранить растровое изображение.

Для устранения отмеченных недостатков был разработан формат *расширенного метафайла* (расширение имени файла — `emf`).

На примере вывода графика функции посмотрим, как можно реализовать подобную задачу, используя расширенный метафайл.

Здесь используется новый набор функций:

1. Создание расширенного метафайла, функция `CreateEnhMetaFile()`.

   ```
   HDC WINAPI CreateEnhMetaFileW(HDC hdcRef, LPCWTSTR lpFilename,
   CONST RECT* lpRect, LPCWTSTR lpDescription);
   ```

   - `hdcRef` — описатель контекста устройства, если `NULL`, GDI берет эту метрическую информацию из контекста устройства дисплея;
   - `lpFilename` — имя файла, если `NULL`, метафайл создается в памяти;

- `lpRect` — адрес структуры `RECT`, описывающей размеры метафайла, если `NULL`, GDI сама определит размеры;
- `lpDescription` — текстовая строка, описывающая метафайл, состоит из двух частей. Первая часть — имя приложения, заканчивается символом `'\0'`, вторая часть — имя визуального образа, также заканчивающееся символом `'\0'`. Например: `"Demo\0Mouse\0"`. Если вместо параметра задан `NULL`, заголовок будет построен по умолчанию.

  Возвращаемое значение — дескриптор контекста устройства метафайла.

2. Закрытие расширенного метафайла, функция `CloseEnhMetaFile()`.

   ```
   HENHMETAFILE WINAPI CloseEnhMetaFile(HDC hdcEMF);
   ```

   Функция принимает контекст метафайла и возвращает его дескриптор, который может использоваться для *проигрывания* метафайла.

3. Проигрывание расширенного метафайла, функция `PlayEnhMetaFile()`.

   ```
   BOOL WINAPI PlayEnhMetaFile(HDC hdc, HENHMETAFILE hemf, CONST RECT
   *lpRect);
   ```

   где `hdc` — контекст устройства вывода, `hemf` — дескриптор метафайла, `lpRect` — адрес структуры типа `RECT`, определяющей прямоугольник для вывода. Метафайл при выводе подстраивается под размер принимающего прямоугольника.

4. Освобождение ресурсов памяти метафайла, функция `DeleteEnhMetaFile()`.

   ```
   BOOL WINAPI DeleteEnhMetaFile(HENHMETAFILE hemf);
   ```

   `hemf` — дескриптор метафайла на диске. Функция освобождает ресурсы, занятые метафайлом в памяти, причем освобождаются только ресурсы в памяти — дисковый файл сохраняется. Возвращается ненулевое значение при успешном результате работы.

В качестве примера приведем код (листинг 4.11) оконной функции с использованием расширенного метафайла для задачи, рассмотренной в листинге 4.9.

Листинг 4.11. Расширенный метафайл в памяти

```
#define _USE_MATH_DEFINES
#include <math.h>
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc, hdcEMF;
   RECT rt;
   static HENHMETAFILE hmf;
   static int sx, sy;
   int x, y;
   double angle;
   switch (message)
```

```
{
case WM_CREATE:
   hdcEMF = CreateEnhMetaFile(NULL, NULL, NULL, NULL);
   for (x = y = 50, angle = 0.0; angle < 2*M_PI; angle += 0.1)
   {
      MoveToEx(hdcEMF, x, y, NULL);
      x = int(50.0*cos(angle)*sin(angle*4.0)+50.5);
      y = int(50.0*sin(angle)*sin(angle*4.0)+50.5);
      LineTo(hdcEMF, x, y);
   }
   hmf = CloseEnhMetaFile(hdcEMF);
   break;
case WM_SIZE:
   sx = LOWORD(lParam);
   sy = HIWORD(lParam);
   break;
case WM_COMMAND:
   switch (LOWORD(wParam))
   {
   case IDM_EXIT: DestroyWindow(hWnd); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   break;
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
      for(rt.left = 0; rt.left < sx; rt.left += sx/5)
         for(rt.top = 0; rt.top < sy; rt.top += sy/5)
            {
                  rt.right = rt.left + sx/5;
                  rt.bottom = rt.top + sy/5;
                  PlayEnhMetaFile(hdc, hmf, &rt);
            }
   EndPaint(hWnd, &ps);
   break;
case WM_DESTROY:
   DeleteEnhMetaFile(hmf);
   PostQuitMessage(0);
   break;
default: return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

В сообщении `WM_CREATE` создаем метафайл:

```
hdcEMF = CreateEnhMetaFile(NULL, NULL, NULL, NULL);
```

Все параметры функции зададим равными нулю — это означает, что метрическая информация будет взята из текущих установок видеосистемы, файл будет создан в памяти, размеры метафайла определены автоматически, а его описание построено по умолчанию.

При закрытии метафайла

```
hmf = CloseEnhMetaFile(hdcEMF);
```

будет получен его дескриптор `hmf`, который определен как статическая переменная типа `HENHMETAFILE`, поэтому может использоваться в сообщении `WM_PAINT` при *проигрывании* метафайла в прямоугольной области, определенной переменной `rt`:

```
PlayEnhMetaFile(hdc, hmf, &rt);
```

Для тиражирования изображения, так же, как и в листинге 4.9, организуем двойной цикл, но в качестве переменных используем поля структуры `RECT`:

```
for(rt.left = 0; rt.left < sx; rt.left += sx/5)
    for(rt.top = 0; rt.top < sy; rt.top += sy/5)
      {
        . . .
      }
```

В цикле мы создадим таблицу размером 5×5 из прямоугольников в клиентской области окна, определяя поля `left` и `top` структуры `RECT` как (*x*, *y*)-координаты левого верхнего угла области вывода. Поля `right` и `bottom` вычисляем в цикле, добавляя на каждом шаге `1/5` ширины и высоты окна вывода:

```
rt.right = rt.left + sx/5;
rt.bottom = rt.top + sy/5;
```

Остальной же текст программы не отличается от рассмотренного в листинге 4.9, за исключением того, что ресурсы метафайла мы освободим функцией

```
DeleteEnhMetaFile(hmf);
```

при закрытии окна.

Однако вывод в окне будет несколько иным, поскольку сейчас изображение занимает всю выделенную прямоугольную область (рис. 4.10).

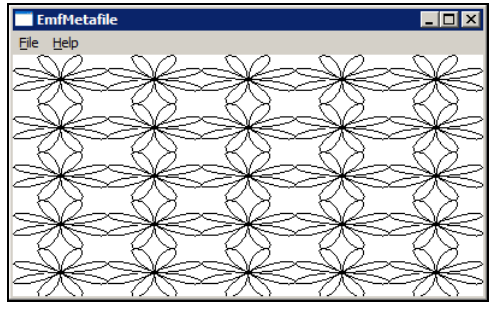


Рис. 4.10. Расширенный метафайл в памяти

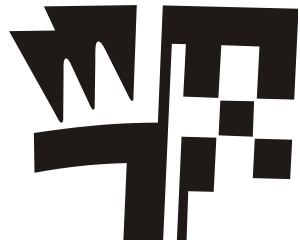
Вопросы к главе

1. Как загрузить растровое изображение из ресурса приложения?
2. Получение информации о размерах изображения.
3. Создание контекста устройства вывода в памяти.
4. Формат функции вывода растрового изображения `BitBlt()`.
5. Чем отличается функция `StretchBlt()` от `BitBlt()`?
6. Чтение растрового изображения из файла, формат функции `LoadImage()`.
7. Растровые операции.
8. Назначение функции `PlgBlt()`.
9. Формат функции `MaskBlt()`, наложение изображения в функции `BitBlt()`.
10. Поворот координат, функция `SetWorldTransform()`.
11. Создание "виртуального" окна.
12. Создание метафайла в памяти и на диске.
13. Расширенные метафайлы.
14. Как получить информацию из заголовка метафайла?

Задания для самостоятельной работы

1. Написать программу, которая делит окно на 4 равные части и выводит в каждой четверти растровое изображение, растягивая его на весь выделенный прямоугольник. При изменении размеров окна размеры изображений должны корректироваться.
2. Создать программу "просмотрщика" графических файлов. Использовать диалоговое окно выбора имени bmp-файлов. Организовать скроллинг, если изображение не помещается в окне.
3. Изменить задачу теста функции `MaskBlt()` (листинг 4.6), реализуя наложение изображений функцией `BitBlt()`.
4. В стандартном диалоговом окне выбрать шрифт и вывести по центру окна произвольный текст, повторяя базовой линией письма контур синусоиды.

   Указание: использовать поворот системы координат для наклона каждого выводимого символа.
5. Вывести в расширенный метафайл растровое изображение и сравнить размер emf-файла с исходным bmp-файлом.



Глава 5

Библиотеки динамической компоновки DLL

Одной из особенностей Windows-приложений является их способность подключать во время работы необходимые функции и ресурсы, которые размещены в так называемых библиотеках динамической компоновки (Dynamic-Link Libraries, *DLL*). Все функции Windows содержатся в dll-файлах, например, графические функции размещены в файле `gdi32.dll`. Преимущество использования библиотек динамической компоновки перед статическими библиотеками проявляется в том, что приложение, состоящее даже из нескольких модулей, использует лишь один экземпляр функции, тогда как из статической библиотеки каждый программный модуль присоединяет свою копию функции на этапе компоновки. Рассмотрим подробно способы создания динамических библиотек и их использование.

Создание DLL

В среде Visual Studio создание новой DLL-библиотеки не представляет больших сложностей. Нужно при создании нового Win32-проекта выбрать тип DLL. Будет создан файл реализации `dllmain.cpp`, который может служить заготовкой новой DLL-библиотеки, а также установятся необходимые ключи компоновки:

```
#include "stdafx.h"
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved )
{
  switch (ul_reason_for_call)
  {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
        break;
  }
  return TRUE;
}
```

Следует обратить внимание на то, что имя головной функции фиксировано — `DllMain()`. Эта функция будет вызываться при загрузке и выгрузке DLL-библиотеки. Все, что нам пока требуется знать об этой функции, это то, что она должна вернуть `TRUE` при успешной загрузке библиотеки.

Далее в файле реализации проекта должны располагаться функции, которые мы хотели бы поместить в данную библиотеку.

Чтобы процесс создания DLL-библиотеки был более ясен, рассмотрим его на примере (листинг 5.1), в котором создается библиотека, состоящая всего лишь из одной функции, строящей треугольник. Поскольку пример небольшой, то приведем его полностью.

Листинг 5.1. Описание DLL-библиотеки (dllmain.cpp)

```
#include "stdafx.h"
__declspec(dllexport) BOOL WINAPI Triangle(HDC, POINT*);
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
  return TRUE; //переключатель switch нам пока не нужен
}
BOOL WINAPI Triangle(HDC hdc, POINT *pt)
{
  MoveToEx(hdc, pt[0].x, pt[0].y,NULL);
  LineTo(hdc, pt[1].x, pt[1].y);
  LineTo(hdc, pt[2].x, pt[2].y);
  return LineTo(hdc, pt[0].x, pt[0].y);
}
```

Функция `Triangle()` принимает контекст устройства `hdc` и массив из трех точек типа `POINT`. Треугольник строится вызовом стандартных функций `MoveToEx()` и `LineTo()`. В качестве возвращаемого значения используется значение, которое вернет последний вызов функции `LineTo()`. Разумно предположить, что если этот вызов был удачным, то и наша функция должна сообщить об удачном завершении. Спецификатор `WINAPI` означает стандартное соглашение Windows для порядка передачи параметров GDI-функций.

Особый интерес вызывает описание прототипа функции:

```
__declspec(dllexport) BOOL WINAPI Triangle(HDC, POINT*);
```

Директива `__declspec(dllexport)` используется компоновщиком для построения раздела экспорта dll-файла.

Примечание

DLL, как и EXE-файл, имеет несколько разделов, в частности разделы экспорта и импорта, которые используются при его загрузке для поиска экспортируемых из библиотеки функций, а также функций, импортируемых из других библиотек.

При обнаружении хотя бы одного экспортируемого элемента компоновщик кроме dll строит и lib-файл, структура которого отличается от соответствующего файла статической библиотеки. Здесь lib-файл содержит только имена экспортируемых функций и их адреса в библиотеке динамической компоновки.

Функция `Triangle()` использует две стандартные функции, прототипы которых размещены в файле включений `wingdi.h`:

```
WINGDIAPI BOOL  WINAPI MoveToEx(HDC, int, int, LPPOINT);
WINGDIAPI BOOL WINAPI LineTo(HDC, int, int);
```

Если посмотреть, что такое `WINGDIAPI`, то можно найти определение:

```
#define WINGDIAPI DECLSPEC_IMPORT
```

Отслеживая определение `DECLSPEC_IMPORT`, мы находим:

```
#define DECLSPEC_IMPORT __declspec(dllimport)
```

Фактически перед функциями `MoveToEx()` и `LineTo()` расположен спецификатор **`__declspec(dllimport)`.**

Он служит указанием компоновщику для создания раздела импорта, куда помещаются ссылки на эти функции.

Использование DLL

Итак мы создали DLL-библиотеку из одной функции, но это не принципиально — функций может быть сколько угодно много. Библиотеку нужно скомпилировать и убедиться, что папка проекта содержит lib- и dll-файлы. Теперь возникает вопрос, как все это применить?

Существует два способа использования DLL-библиотек: *неявное* (*implicit linking*) и *явное* (explicit linking) *связывание*.

Неявное связывание

Неявное связывание (Implicit linking) — самый распространенный в настоящее время способ загрузки DLL-библиотек. Суть его заключается в том, что компоновщик при построении исполняемого exe-файла не включает код функций в тело программы, а создает раздел импорта, где перечисляются символические имена функций и переменных для каждой из DLL-библиотек. Для этого к проекту необходимо подключить соответствующий lib-файл. При запуске программы загрузчик операционной системы анализирует раздел импорта, загружает все необходимые DLL-библиотеки и, что важно, проецирует их на адресное пространство загружаемого приложения. Причем, если в загружаемой DLL-библиотеке существует свой раздел импорта, то загружаются и те dll-файлы, которые там указаны. Однако если биб-

лиотека один раз уже загружена, то второй раз она не загружается, а строится ссылка на уже существующий экземпляр.

Поиск DLL-библиотек осуществляется по стандартной схеме:

1. В папке, откуда запущено приложение.
2. В текущей папке.
3. В системной папке Windows\system32.
4. В папке Windows.
5. В папках, которые перечислены в переменной окружения PATH.

Если библиотека не найдена, загрузка приложения прекращается и выводится сообщение об ошибке.

Посмотрим, как же будет загружаться наша DLL-библиотека (листинг 5.1).

Поскольку она имеет раздел экспорта, то библиотека загружается в память, но функция `Triangle()` использует еще две стандартные функции, которые помещены в разделе импорта DLL-библиотеки. Если говорить точнее, то в нашей библиотеке хранятся не коды этих функций, а лишь ссылка на них из библиотеки `gdi32.dll`. Таким образом, при загрузке приложения, когда дело дойдет до загрузки DLL-библиотек, вначале загрузится библиотека `Triangle.dll`, затем будет загружена `gdi32.dll`. Эта библиотека, как правило, уже загружена, поскольку используется операционной системой, поэтому будут использованы ссылки на функции `MoveToEx()` и `LineTo()` из уже имеющейся в памяти библиотеки.

Рассмотрим задачу для демонстрации разработанной нами библиотеки DLL (листинг 5.2). Создадим новый проект, скопируем в папку проекта файлы `Triangle.dll` и `Triangle.lib`. Последний файл необходимо явно подключить к проекту, что можно сделать в меню **Project | Properties…** на вкладке **Linker | Input** с помощью диалогового окна **Additional Dependencies** (см. рис. 2.3).

Листинг 5.2. Пример неявного связывания DLL-библиотеки

```
WINGDIAPI BOOL WINAPI Triangle(HDC, POINT*);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static HPEN hPen;
   static int sx, sy;
   POINT pt[3];
   switch (message)
   {
   case WM_CREATE:
     hPen = CreatePen(PS_SOLID, 4, RGB(0, 0, 255));
     break;
```

```
    case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      break;
    case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
    case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      SelectObject(hdc, hPen);
      pt[0].x = sx/4;
      pt[0].y = pt[1].y = sy/4;
      pt[1].x = sx*3/4;
      pt[2].x = sx/2;
      pt[2].y = sy/2;
      Triangle(hdc, pt);
      EndPaint(hWnd, &ps);
      break;
    case WM_DESTROY:
      DeleteObject(hPen);
      PostQuitMessage(0);
      break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Для созданной нами DLL-библиотеки желательно подготовить заголовочный h-файл с прототипами функций. Однако можно описать прототип функции в тексте программы. Обратите внимание, что в качестве директивы компоновщика используется `WINGDIAPI`, поскольку, как мы выяснили ранее, это то же самое, что и `__declspec(dllimport)`.

Из файла `Triangle.cpp` мы приводим лишь оконную функцию, в которой опишем дескриптор пера `hPen`, две переменных целого типа для хранения размеров окна `sx`, `sy`, а также массив точек `pt[3]`.

Сплошное синее перо толщиной 4 пиксела создадим в сообщении `WM_CREATE`:

```
hPen = CreatePen(PS_SOLID, 4, RGB(0, 0, 255));
```

В сообщении `WM_SIZE` определим размер клиентской области окна `sx`, `sy`.

Все рисование сосредоточим в сообщении `WM_PAINT`, где получим контекст устройства и выберем для него перо:

```
SelectObject(hdc, hPen);
```

Теперь заполним поля массива `pt` для рисования треугольника в верхней части окна:

```
pt[0].x = sx/4;
pt[0].y = pt[1].y = sy/4;
pt[1].x = sx*3/4;
pt[2].x = sx/2;
pt[2].y = sy/2;
```

Обращаемся к функции `Triangle()` как к обычной GDI-функции:

```
Triangle(hdc, pt);
```

Задача решена. Нужно не забыть уничтожить перо при закрытии окна.

После запуска программы будет выведено окно с изображением треугольника (рис. 5.1). При изменении размеров окна треугольник будет перестраиваться.

> ***Примечание***
>
> Если запустить созданный exe-файл в командной строке, то приложение не сможет выполниться, поскольку в текущей папке не будет DLL-библиотеки. Решением проблемы может быть размещение библиотеки динамической компоновки в той же папке, где находится файл приложения, либо в одной из папок поиска DLL-библиотек.

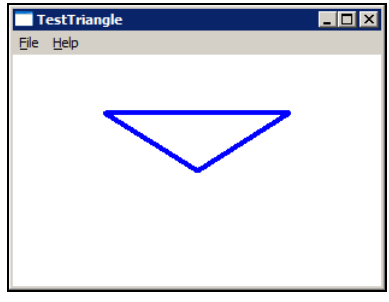


Рис. 5.1. Графические построения с применением пользовательской DLL-библиотеки

DLL общего использования

Созданная нами DLL-библиотека работает с программами, написанными на языке C++. Если же попытаться использовать ее в программе, написанной на другом языке программирования, мы выясним, что в ней отсутствует функция с именем `Triangle`. Но имеется функция `? Triangle@@YGHPAUHDC__@@PAUtagPOINT@@@Z`.

Дело в том, что по соглашению языка C++ функция идентифицируется не только именем, но и типами формальных параметров, что называется сигнатурой функции. Нетрудно сообразить, что именно эта особенность C++ позволяет определять несколько функций с одним именем, но с разным числом и типом параметров.

В нашей ситуации это является скорее помехой, поскольку вряд ли кому понравится строить такие специфические имена функций.

Однако когда приложение создается в C++, то имя так и строится, поэтому проблем с поиском функций не возникает. Другое дело, если приложение пишется на другом языке, который "ничего не знает" о соглашениях C++.

Одним из выходов в данной ситуации является использование спецификатора компоновки для классического языка C, где искажения имени функции не происходит:

```
extern "язык" прототип функции
```

Такая конструкция позволяет строить обращение к функции в соответствии со стандартом выбранного языка. Например, в нашем случае функцию можно было бы описать так:

```
extern "C" __declspec(dllexport) BOOL Triangle(HDC, POINT*);
```

> ***Примечание***
>
> К сожалению, имеющиеся в настоящее время компиляторы фирмы Microsoft поддерживают спецификатор компоновки только для языка C.

Обратите внимание, что в этом случае функция `Triangle()` использует стандартный для классического C порядок передачи параметров. Если необходимо создать библиотеку функций обратного вызова `CALLBACK`, необходимо использовать иной подход.

Более приемлемый путь решения данной задачи заключается в использовании def-файла (*файла определений* с расширением def — от англ. definition). Его можно добавить к проекту в меню **Project | Add New Item… | Code | Module-Definition File (.def)** (рис. 5.2).

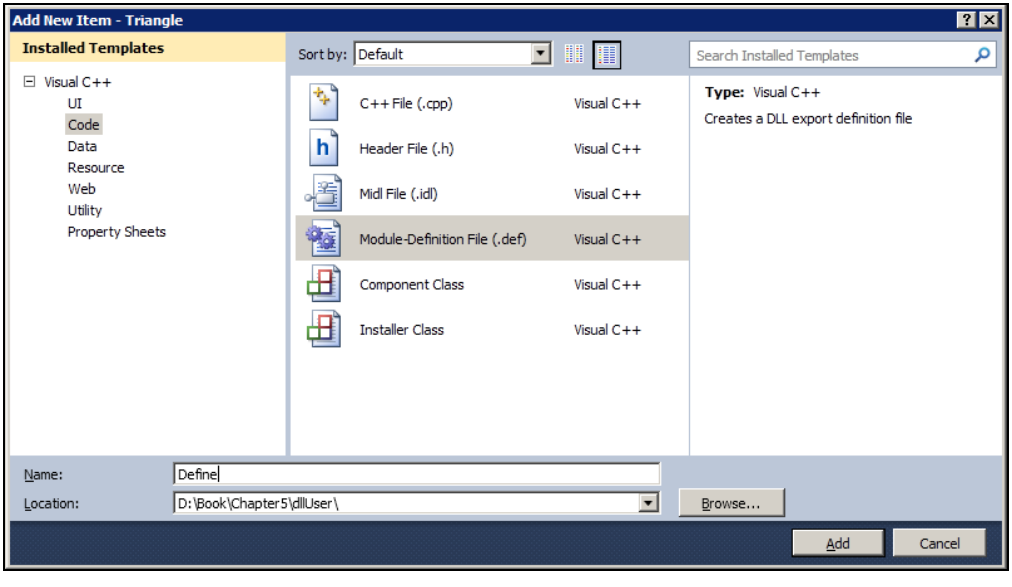


Рис. 5.2. Добавление к проекту файла определений

Если при разработке проекта DLL-библиотеки включить в него def-файл с парой строк, которые представляют раздел экспорта:

```
EXPORTS
Triangle
```

то компоновщик помещает в раздел экспорта библиотеки функцию с именем `Triangle` без искажений. После этого библиотека станет доступной для использования любому Windows-приложению.

> ***Примечание***
>
> Если поставить цель написать DLL-библиотеку общего использования, нужно всегда применять def-файл и только те типы данных, которые определены в Windows. Имя файла произвольно, лишь бы он был включен в проект.

Явная загрузка DLL

Неявное связывание, несмотря на свою очевидную простоту, имеет и определенные недостатки. Основным из них является одновременная (с приложением) загрузка и последующее удаление из памяти всех DLL-библиотек, независимо от того, что большая часть из них может и не понадобиться. Однако имеется возможность во время работы приложения загружать только необходимые DLL, освобождая занимаемую ими память, когда потребность в них отпала. Такой способ называют *явным связыванием* (explicit linking).

Каждое приложение имеет свое адресное пространство, и чтобы получить доступ к функциям и ресурсам, размещенным в DLL-библиотеке, нужно спроецировать библиотеку на адресное пространство приложения. Это можно сделать функцией `LoadLibrary()`:

```
HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);
```

аргументом которой является имя DLL-библиотеки.

Функция возвращает дескриптор спроецированного в память dll-файла.

`HMODULE` — тип возвращаемого значения, это просто другое обозначение дескриптора приложения `HINSTANCE`.

Функция `FreeLibrary()` выгружает DLL-библиотеку из памяти:

```
BOOL WINAPI FreeLibrary(HMODULE hLibModule);
```

Эта функция принимает дескриптор, полученный при загрузке DLL, и возвращает `TRUE` при успешном завершении.

> ***Примечание***
>
> На самом деле функция `FreeLibrary()` лишь уменьшает счетчик пользователей библиотеки на 1, а выгружается библиотека из памяти тогда, когда счетчик пользователей становится равным 0.

Рассмотрим реализацию библиотеки для загрузки ее явно. Мы должны обеспечить неизменность имен функций, входящих в библиотеку. Этого проще всего добиться использованием def-файла. Добавим к проекту библиотеки def-файл и перекомпилируем проект.

Создадим новый проект ExplicitLinking (листинг 5.3) для использования этой библиотеки и скопируем в этот проект единственный файл `Triangle.dll`. При явной загрузке lib-файл нам не понадобится, и не нужно включать его в проект.

Листинг 5.3. Явная загрузка DLL-функций

```
typedef BOOL (WINAPI* PFN) (HDC, POINT*);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static HINSTANCE hLibrary;
   static PFN pfnTriangle;
   static HPEN hPen;
   static int sx, sy;
   POINT pt[3];
   switch (message)
   {
   case WM_CREATE:
      hLibrary = LoadLibrary(_T("Triangle"));
      if (hLibrary)
      {
        pfnTriangle = (PFN)GetProcAddress(hLibrary, "Triangle");
        if (pfnTriangle == NULL)
        {
           MessageBox(hWnd, _T("Функция Triangle не найдена"),
                     _T("LoadLibrary"), MB_OK | MB_ICONQUESTION);
           DestroyWindow(hWnd);
           return 0;
        }
        hPen = CreatePen(PS_SOLID, 4, RGB(0, 0, 255));
      }
      else
      {
        MessageBox(hWnd, _T("Библиотека не найдена"), _T("LoadLibrary"),
                     MB_OK | MB_ICONQUESTION);
        DestroyWindow(hWnd);
        return 0;
      }
      break;
   case WM_SIZE:
      sx = LOWORD(lParam);
      sy = HIWORD(lParam);
      break;
```

```
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      SelectObject(hdc, hPen);
      pt[0].x = sx/4;
      pt[0].y = pt[1].y = sy/4;
      pt[1].x = sx*3/4;
      pt[2].x = sx/2;
      pt[2].y = sy/2;
      pfnTriangle(hdc, pt);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      FreeLibrary(hLibrary);
      DeleteObject(hPen);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При создании окна в сообщении `WM_CREATE` загружаем библиотеку:

```
hLibrary = LoadLibrary(_T("Triangle"));
```

> ***Примечание***
>
> В аргументе функции `LoadLibrary()` не следует указывать расширение имени dll, поскольку в этом случае поиск библиотеки будет осуществляться, начиная с каталога "известных DLL", прописанного в разделе KnownDLLs реестра Windows.

Возвращаемое значение сохраняем в переменной `hLibrary` типа `HINSTANCE`. В случае успешной загрузки `hLibrary` отлично от нуля, и мы можем получить адрес библиотечной функции при помощи `GetProcAddress()`:

```
FARPROC WINAPI GetProcAddress(HMODULE hModule, LPCSTR lpProcName);
```

Здесь второй параметр `lpProcName` — С-строка.

> ***Примечание***
>
> Адрес библиотечной функции отыскивается по ее имени, поэтому, даже после модификации библиотеки, если имя и параметры функции не изменились, она будет найдена. Кодировка Unicode в именах функций DLL-библиотеки не используется.

Для возвращаемого значения необходимо создать новый тип данных `PFN` — указатель на функцию с двумя параметрами. Воспользуемся оператором `typedef` и на глобальном уровне определим этот тип:

```
typedef BOOL (WINAPI* PFN) (HDC, POINT*);
```

Переменную `pfnTriangle` типа `PFN` опишем как статическую, это и будет указатель на функцию `Triangle` из динамической библиотеки. Сейчас ее можно использовать как имя функции, что мы и сделаем в сообщении `WM_PAINT`:

```
pfnTriangle(hdc, pt);
```

Директивой `FreeLibrary(hLibrary)` выгрузим библиотеку перед закрытием окна.

> ***Примечание***
>
> Функция `LoadLibrary()`, перед тем как загрузить библиотеку, проверяет наличие такой библиотеки в памяти, и, если библиотека уже загружена, просто увеличивает счетчик обращений к ней на `1` и отображает ее на адресное пространство приложения.

Загрузка ресурсов из DLL

Помимо функций из DLL-библиотеки можно загрузить и ресурсы. Для примера создадим простенькую DLL, содержащую в ресурсе одну иконку, и рассмотрим два способа ее извлечения при явном и неявном связывании.

Создадим проект DLL-библиотеки и импортируем туда иконку `IDI_ICON1`.

Листинг 5.4. Библиотека динамической компоновки, содержащая ресурсы

```
#include "stdafx.h"
#include "resource.h"
__declspec(dllexport) HICON hIcon;
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                      )
{
   switch (ul_reason_for_call)
   {
   case DLL_PROCESS_ATTACH:
         hIcon = LoadIcon(hModule, MAKEINTRESOURCE(IDI_ICON1));
         break;
   case DLL_PROCESS_DETACH:
         DestroyIcon(hIcon);
         break;
   }
   return TRUE;
}
```

Программный код библиотеки не содержит ни одной функции, а только описание глобальной переменной `hIcon`, объявленной как *"экспортируемая"*:

```
__declspec(dllexport) HICON hIcon;
```

Функция `DllMain()`, которую называют функцией входа/выхода, вызывается операционной системой при загрузке и выгрузке DLL-библиотеки и имеет три параметра:

- `HModule` — дескриптор библиотеки, присваивается при загрузке;
- `ul_reason_for_call` — код уведомления;
- `lpReserved` — зарезервировано для дальнейшего применения.

Код уведомления `ul_reason_for_call` может принимать одно из четырех значений:

- `DLL_PROCESS_ATTACH` — при создании нового процесса;
- `DLL_PROCESS_DETACH` — при завершении процесса;
- `DLL_THREAD_ATTACH` — при создании нового потока;
- `DLL_THREAD_DETACH` — при завершении потока.

Обычно при обработке этих уведомлений в DLL-библиотеке осуществляются операции по выделению и освобождению необходимых для работы ресурсов операционной системы. Например, при загрузке библиотеки требуется выделить память, а при ее выгрузке — освободить.

Иконку можно загрузить из ресурса библиотеки функцией `LoadIcon()`:

```
hIcon = LoadIcon(hModule, MAKEINTRESOURCE(IDI_ICON1));
```

Мы сохраняем в глобальной переменной дескриптор иконки, чтобы при выгрузке библиотеки из памяти по уведомлению `DLL_PROCESS_DETACH` освободить память:

```
DestroyIcon(hIcon);
```

Поскольку потоков создавать не планируется, то уведомления `DLL_THREAD_ATTACH` и `DLL_THREAD_DETACH` обрабатывать не будем.

Рассмотрим два способа загрузки иконки:

1. Создадим проект демонстрационной задачи. В папку этого проекта скопируем созданные файлы библиотеки `DllIcon.dll` и `DllIcon.lib`.

   В свойствах проекта добавим имя библиотеки `DllIcon.lib`. На глобальном уровне опишем импортируемую переменную:

   ```
   __declspec(dllimport) HICON hIcon;
   ```

   а в сообщении `WM_CREATE` переопределим малую иконку класса окна:

   ```
   SetClassLong(hWnd, GCL_HICONSM, (LONG)hIcon);
   ```

   Поскольку окно прорисовывается после обработки этого сообщения, мы увидим в заголовке окна новую пиктограмму приложения.

2. Для явной загрузки библиотеки создадим новый проект, нам понадобится лишь dll-файл созданной библиотеки, подключать lib-файл не нужно. В сообщении `WM_CREATE` необходимо получить дескриптор библиотеки:

   ```
   hDll = LoadLibrary(_T("DllIcon"));
   ```

передавая ей в качестве параметра имя DLL-файла. Далее, функцией `GetProcAddress()` находим дескриптор иконки, уже загруженной в библиотеке, передавая ей дескриптор иконки как текстовую строку.

```
hIcon = *((HICON*)GetProcAddress(hDll, "hIcon"));
```

Переменная `hIcon` описана как `HICON`.

После этого мы можем, как и в предыдущем случае, изменить малую иконку класса окна:

```
SetClassLong(hWnd, GCL_HICONSM, (LONG)hIcon);
```

Результат работы этого варианта программы будет идентичен предыдущему. lib-файл в проекте нам не нужен, но нужно иметь в виду, что DLL-библиотека должна быть создана с использованием def-файла:

```
EXPORTS
hIcon
```

Примечание

В Visual Studio имеется специальная утилита `dumpbin`, которая позволяет просмотреть содержимое разделов dll-файла. Протокол будет выглядеть так:

```
dumpbin /exports DllIcon.dll
Section contains the following exports for DllIcon.dll
  00000000 characteristics
  48F621AA time date stamp Wed Oct 15 23:00:26 2008
      0.00 version
         1 ordinal base
         1 number of functions
         1 number of names
  ordinal hint RVA      name
        1    0 00017138 hIcon
```

Весь набор ключей `dumpbin` можно получить командой: `dumpbin /?`. Нужно только иметь в виду, что для работы данной утилиты требуется, чтобы в пределах доступных путей поиска находились два модуля: `link.exe`, `mspdb100.dll`.

При использовании командного процессора Total — либо Windows Commander, можно воспользоваться стандартным просмотрщиком, который "умеет видеть" структуру dll-файла.

DLL, содержащие только ресурсы

Можно использовать DLL-библиотеку, как контейнер для хранения ресурсов, например: курсоров, иконок и пр. В этом случае головная функция `DllMain()` ничего не делает и будет выглядеть так:

```
BOOL APIENTRY DllMain( HMODULE, DWORD, LPVOID)
{
   return TRUE;
}
```

Поскольку в такой библиотеке не будет экспортируемых переменных, lib-файл не создается и библиотека может быть загружена только явно. Для доступа к ресурсам необходимо сохранить значение их идентификаторов, которые определены в файле `resource.h`, например:

```
#define IDI_ICON1    101
```

Эти определения лучше всего разместить в h-файле и поставлять вместе с dll-файлом.

Если создать такую библиотеку, то обработчик сообщения `WM_CREATE` мог бы выглядеть так:

```
HMODULE hDll;
HICON hIcon;
. . .
case WM_CREATE:
  hDll = LoadLibrary(_T("DllIcon"));
  hIcon = LoadIcon(hDll, MAKEINTRESOURCE(IDI_ICON1));
  SetClassLong(hWnd, GCL_HICONSM, (LONG)hIcon);
  break;
```

Вопросы к главе

1. В чем отличие библиотек динамической компоновки от статических библиотек?
2. Какие действия выполняет функция `DllMain()`, коды уведомлений?
3. Объявление экспортируемых переменных и функций. Проблема "искажения" имен, спецификатор компоновки `extern "C"` и def-файл.
4. Явное и неявное связывание DLL-библиотек.
5. Пути поиска dll-файла.
6. Формат функций `LoadLibrary()`, `GetProcAddress()`, `FreeLibrary()`.
7. Как найти адрес функции и переменной в DLL-библиотеке?
8. Как создать DLL-библиотеку для хранения ресурсов?

Задания для самостоятельной работы

1. Создайте DLL-библиотеку `UserString` для работы с С-строкой, включив в нее аналоги стандартных функций: `strlen()`, `strcpy()`, `strcat()`, `strrev()`...
2. Постройте демонстрационную задачу для использования созданной библиотеки при неявном и явном связывании.
3. Создайте библиотеку, содержащую набор ресурсов: иконку, курсор, растровое изображение. Постройте демонстрационную задачу, использующую ресурсы этой DLL-библиотеки.
4. Создайте DLL-библиотеку с галереей рисунков одинакового размера. В число экспортируемых переменных включите название галереи и количество изображений. Окно создаваемого приложения заполните этими рисунками.
5. При помощи утилиты `dumpbin` просмотрите разделы экспорта и импорта созданных библиотек.

Глава 6

Процессы и потоки

Операционная система Windows является многозадачной системой и позволяет одновременно выполнять несколько приложений. Разумеется, при наличии одного процессора многозадачность может быть реализована лишь переключением между различными приложениями, которым выделяется определенный квант времени. При достаточной скорости обработки это создает иллюзию одновременного выполнения нескольких приложений. О механизме передачи управления поговорим чуть позднее, а сейчас рассмотрим технику создания процесса.

Создание процесса

Любое приложение, запущенное на выполнение, представляет собой *процесс*. Каждому процессу при загрузке выделяются ресурсы операционной системы:

- объект ядра, представляющий собой небольшой блок памяти, через который операционная система управляет выполнением процесса;
- адресное пространство, содержащее код и данные.

Приложение может создать новый процесс, проще говоря, может запустить другое приложение. В любой момент вновь созданный процесс может быть при необходимости уничтожен. Для создания нового процесса используется функция `CreateProcess()`:

```
BOOL WINAPI CreateProcessW(
  LPCWSTR lpApplicationName,                 //имя приложения
  LPWSTR lpCommandLine,                      //командная строка
  LPSECURITY_ATTRIBUTES lpProcessAttributes, //атрибуты доступа процесса
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  //и потока
  BOOL bInheritHandles,                      //наследование дескрипторов
  DWORD dwCreationFlags,                     //флаги
  LPVOID lpEnvironment,                      //параметры среды
  LPCWSTR lpCurrentDirectory,                //текущая папка процесса
  LPSTARTUPINFO lpStartupInfo,               //структура стартовых полей
  LPPROCESS_INFORMATION lpProcessInformation //возвращаемые значения
  );
```

Функция имеет 10 полей, однако многие из них можно задавать по умолчанию (параметр NULL).

- Если вместо имени программы lpApplicationName задать NULL, под именем программы будет пониматься первое имя, стоящее в командной строке lpCommandLine. Этот вариант предпочтительнее, поскольку, во-первых, к имени приложения автоматически добавляется расширение "exe", во-вторых, приложение ищется по стандартной схеме, аналогично поиску dll-файлов. Если же имя приложения задать первым параметром, то обязательно нужно указать полное имя с расширением, и если приложения с таким именем не существует, то работа функции закончится неудачей.

  Примечание

  Unicode-версия функции CreateProcess() завершится неудачей, если CommandLine — строка константного типа.

- Если параметры lpProcessAttributes и lpThreadAttributes установлены в NULL, используются текущие права доступа. Обычно эти параметры применяются при создании серверных приложений, однако можно установить поле структуры SECURITY_ATTRIBUTES bInheritHandle равным TRUE для определения дескриптора как наследуемого.

```
struct SECURITY_ATTRIBUTES {
    DWORD nLength;               //размер структуры
    LPVOID lpSecurityDescriptor; //указатель описателя защиты
    BOOL bInheritHandle;};       //признак наследования
```

  Все дескрипторы объектов ядра, порождаемые с таким параметром, могут наследоваться порожденными процессами, однако это не означает, что они передаются автоматически. На самом деле порождаемый процесс "ничего не знает" о наследуемых дескрипторах, поэтому их нужно как-то передать в порождаемый процесс, например, в параметре командной строки или через переменную окружения.

- Обычно процессы создают с нулевым значением параметра dwCreationFlags, что означает "*нормальный*" режим выполнения процесса, но при задании других значений флага можно, например, запустить процесс в отладочном режиме или же задать класс приоритета процесса. Справку о допустимых флагах можно получить в справочной системе MSDN (Microsoft Developer Network).

- Параметр lpEnvironment — указатель на буфер с параметрами среды; если этот параметр равен NULL, порождаемый процесс наследует среду окружения родительского процесса.

- Если параметр lpCurrentDirectory установлен в NULL, то текущая папка родительского процесса будет унаследована порождаемым процессом.

- Параметр lpStartupInfo — указатель на структуру STARTUPINFO, поля которой определяют режим открытия нового процесса:

```
struct STARTUPINFO {
    DWORD   cb;              //размер структуры
    LPSTR   lpReserved;      //NULL
```

```
    LPSTR   lpDesktop;      //имя "рабочего стола"
    LPSTR   lpTitle;        //заголовок консоли
    DWORD   dwX;            //левый верхний угол
    DWORD   dwY;            //нового окна
    DWORD   dwXSize;        //ширина
    DWORD   dwYSize;        //и высота нового консольного окна
    DWORD   dwXCountChars;  //размер буфера
    DWORD   dwYCountChars;  //консоли
    DWORD   dwFillAttribute; //цвет текста (в консольном приложении)
    DWORD   dwFlags;        //флаг определяет разрешенные поля
    WORD    wShowWindow;    //способ отображения окна
    WORD    cbReserved2;    //NULL
    LPBYTE  lpReserved2;    //NULL
    HANDLE  hStdInput;      //дескрипторы стандартных
    HANDLE  hStdOutput;     //потоков ввода/вывода
    HANDLE  hStdError;      //потока ошибок
};
```

Поле `dwFlags` обычно устанавливают в `STARTF_USESHOWWINDOW`, что позволяет задать способ отображения окна. Значение `SW_SHOWNORMAL` в поле `wShowWindow` позволяет Windows самостоятельно определить размер и положение открываемого окна, а `SW_SHOWMINIMIZED` означает, что приложение открывается в "свернутом" виде.

- И, наконец, параметр `lpProcessInformation` — указатель на структуру `PROCESS_INFORMATION`, в которой возвращается значение дескриптора и идентификатора порождаемого процесса и потока:

```
struct PROCESS_INFORMATION {
    HANDLE hProcess;   //дескриптор нового процесса
    HANDLE hThread;    //дескриптор главного потока
    DWORD dwProcessId; //идентификатор нового процесса
    DWORD dwThreadId;  //идентификатор главного потока
};
```

Несмотря на то, что созданный процесс абсолютно независим от родителя, родительский процесс может принудительно завершить его в любой момент функцией `TerminateProcess()`:

```
BOOL WINAPI TerminateProcess(HANDLE hProcess, UINT fuExitCode);
```

В качестве первого параметра этой функции используется дескриптор процесса, который возвращается в поле `hProcess` структуры `PROCESS_INFORMATION`. Второй параметр `fuExitCode` — код возврата.

Обе функции возвращают ненулевое значение при успешном завершении.

Процесс может быть завершен и "изнутри" вызовом функции `ExitProcess()`:

```
VOID WINAPI ExitProcess(UINT fuExitCode);
```

Рассмотрим простейший пример создания процесса (листинг 6.1). Приложение имеет два пункта меню: **Открыть блокнот** и **Закрыть блокнот**. Мы запустим стандартное приложение notepad в виде процесса, а затем выгрузим его из памяти. Чтобы показать возможность передачи параметров создаваемому процессу, передадим ему в командной строке имя файла для открытия.

Листинг 6.1. Создание процесса

```
TCHAR CommandLine[256] = _T("notepad ReadMe.txt");
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   static STARTUPINFO tin;
   static PROCESS_INFORMATION pInfo;
   static DWORD exitCode;
   switch (message)
   {
   case WM_CREATE:
      tin.cb = sizeof(STARTUPINFO);
      tin.dwFlags = STARTF_USESHOWWINDOW;
      tin.wShowWindow = SW_SHOWNORMAL;
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_FILE_OPEN:
         GetExitCodeProcess(pInfo.hProcess, &exitCode);
         if (exitCode != STILL_ACTIVE) CreateProcess(NULL, CommandLine,
                     NULL, NULL, FALSE, 0, NULL, NULL, &tin, &pInfo);
         break;
      case ID_FILE_DELETE:
         GetExitCodeProcess(pInfo.hProcess, &exitCode);
         if (exitCode==STILL_ACTIVE) TerminateProcess(pInfo.hProcess, 0);
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При описании переменных, необходимых для создания нового процесса, мы объявили их статическими:

```
static STARTUPINFO tin;
static PROCESS_INFORMATION pInfo;
```

Поэтому все поля переменных `tin`, `pInfo` имеют начальное нулевое значение, и нам достаточно определить лишь 3 поля структуры `STARTUPINFO`. Что мы и сделаем при обработке сообщения `WM_CREATE`:

```
tin.cb = sizeof(STARTUPINFO);
tin.dwFlags = STARTF_USESHOWWINDOW;
tin.wShowWindow = SW_SHOWNORMAL;
```

Так мы установим *"нормальный"* режим открытия приложения.

Процесс же создадим при обработке пункта меню **Открыть блокнот**, его идентификатор `ID_FILE_OPEN`:

```
GetExitCodeProcess(pInfo.hProcess, &exitCode);
if (exitCode != STILL_ACTIVE) CreateProcess(NULL, CommandLine, NULL,
NULL, FALSE, 0, NULL, NULL, &tin, &pInfo);
```

В качестве параметров функции зададим всего три отличных от нуля поля:

- командную строку для вызова приложения и передачи ему имени файла в качестве параметра `CommandLine`;
- указатель на структуру `tin`, поля которой мы определили при создании окна;
- указатель на структуру `pInfo`, где мы получим дескриптор созданного процесса;

Обращение к функции `CreateProcess()` заключили в условный оператор

```
if (exitCode != STILL_ACTIVE) . . .
```

чтобы избежать создания нескольких экземпляров блокнота. Это, конечно, не страшно, однако при создании следующего экземпляра блокнота мы не только теряем всю информацию о ранее созданном экземпляре, но и возможность уничтожения объекта ядра, что приведет к "замусориванию памяти".

Переменную `exitCode` — код возврата — получим при обращении к функции `GetExitCodeProcess()`:

```
BOOL WINAPI GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);
```

Эта переменная принимает значение `STILL_ACTIVE (0x103)` только в том случае, если процесс активен, поэтому мы благополучно создадим процесс, если он еще не создан.

> ***Примечание***
>
> Функция `GetExitCodeProcess()` может безопасно обратиться по дескриптору, равному 0 — именно это и будет происходить, пока процесс еще не создан. Если же процесс уничтожен "изнутри", то дескриптор этого процесса будет актуален в родительском приложении, поскольку объект ядра не будет уничтожен до тех пор, пока его дескриптор не закрыт функцией `CloseHandle()`, и мы сможем получить код завершения дочернего процесса.

При удалении блокнота в пункте меню **Закрыть блокнот** необходимо также проверить — работает ли дочерний процесс. Чтобы разобраться, как это сделано, необходимо представлять механизм создания процесса.

При создании нового процесса операционной системой создаются объекты ядра "процесс" и "поток" и выделяется виртуальное адресное пространство процесса. У каждого объекта ядра существует счетчик пользователей, и здесь счетчику присваивается значение 2. Одну единицу присваивает сам созданный процесс, вторая единица добавляется при передаче дескриптора родительскому процессу. По существующему соглашению операционная система автоматически удаляет объект ядра, как только счетчик обращений принимает нулевое значение. Однако когда приложение закрылось самостоятельно, счетчик пользователей уменьшается на 1 и объект ядра сохраняется, в нем же хранится код возврата.

Таким образом, операционная система сохраняет объект ядра дочернего процесса до тех пор, пока не будет разорвана его связь с родительским процессом.

> ***Примечание***
>
> Можно "разорвать" связь с родительским процессом сразу после его создания вызовом функции `CloseHandle()`, которая просто уменьшит счетчик обращений на 1.
>
> ```
> CloseHandle(pInfo. hThread);
> CloseHandle(pInfo.hProcess);
> ```
>
> Теперь при самостоятельном закрытии дочернего процесса объект ядра будет автоматически уничтожен операционной системой.

Имея дескриптор дочернего процесса, несложно проверить его существование, и, если дочерний процесс работает, мы его уничтожим.

```
GetExitCodeProcess(pInfo.hProcess, &exitCode);
if (exitCode == STILL_ACTIVE) TerminateProcess(pInfo.hProcess, 0);
```

Теперь можно завершать работу блокнота "естественным образом" изнутри или принудительно из родительского процесса, но в этом случае мы потеряем все записанные в блокноте данные.

Создание потока

Каждый выполняемый процесс имеет хотя бы один поток выполнения. Этот поток создается автоматически и условно называется главным. Программный код выполняется в этом потоке. Однако внутри процесса можно создать несколько потоков выполнения, при этом они будут выполняться "параллельно" и процесс может рассматриваться как контейнер для потоков. Все потоки выполняются в контексте некоторого процесса и *разделяют одно адресное пространство*, поэтому потоки могут выполнять один и тот же код и оперировать одними и теми же данными. Переключение между потоками требует меньших затрат ресурсов и происходит быстрее.

Потоку при создании выделяются следующие ресурсы:

- объект ядра;
- стек потока.

Обычно потоки создают для повышения производительности вычислительной системы в том случае, когда они используют разные ресурсы, которые могут использоваться одновременно. Например, в программе, читающей файл с диска, производящей вычисления и выводящей данные на печать, уместно создать три потока, однако здесь возникает проблема их синхронизации. Действительно, если данные с диска еще не прочитаны, то вычислять еще рано. Эту проблему мы обсудим позднее.

Для создания потока используется функция `CreateThread()`:

```
HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, //атрибуты доступа
    DWORD dwStackSize,                        //размер стека потока
    LPTHREAD_START_ROUTINE lpStartAddress,    //функция потока
    LPVOID lpParameter,                       //параметр функции
    DWORD dwCreationFlags,                    //состояние потока
    LPDWORD lpThreadId);                      //идентификатор потока
```

Функция возвращает дескриптор созданного потока либо `0` в случае ошибки.

Если размер стека указан `0`, по умолчанию создается такой же стек, как и у родительского потока.

> ***Примечание***
>
> По умолчанию стек устанавливается размером в 1 Мбайт.

Тип потоковой функции `PTHREAD_START_ROUTINE` описан в файле включений `winbase.h` следующим образом:

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(LPVOID lpThreadParameter);
```

Поток завершается самостоятельно при выходе из функции потока либо может быть завершен извне функцией `TerminateThread()`:

```
BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Функция принимает дескриптор созданного потока `hThread` и код завершения `dwExitCode`, возвращает `TRUE` при успешном завершении.

Для досрочного завершения работы поток может вызвать функцию `ExitThread()`:

```
VOID WINAPI ExitThread(DWORD dwExitCode);
```

В примере (листинг 6.2) создадим поток, который присвоит значение указателю на текстовую строку и завершит работу.

Листинг 6.2. Создание потока

```
TCHAR *pStr;
DWORD WINAPI MyThread(LPVOID param)
{
   pStr = (TCHAR*)param;
   return 0;
}
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   TCHAR *str = _T("Работал поток!!!");
   switch (message)
   {
      case WM_CREATE:
         CreateThread(NULL, 0, MyThread, str, 0, NULL);
         break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      TextOut(hdc, 0, 0, pStr, _tcslen(pStr));
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При создании окна в сообщении `WM_CREATE` создадим поток:

```
CreateThread(NULL, 0, MyThread, str, 0, NULL);
```

> ***Примечание***
>
> Идентификатор потока нам не нужен, и мы вместо него указали `NULL`.

Потоковая функция `MyThread()` получила указатель на строку и присвоила глобальной переменной значение:

```
pStr = (TCHAR*)param;
```

Здесь требуется явное преобразование типа.

В сообщении `WM_PAINT` выведем эту строку в окно:

```
TextOut(hdc, 0, 0, pStr, _tcslen(pStr));
```

Поскольку функция потока очень короткая, то значение указателя `pStr` присвоится до того, как он нам понадобится. На рис. 6.1 показан результат работы программы.

Рис. 6.1. Программа с двумя потоками

Однако чаще всего приходится прилагать определенные усилия, чтобы добиться правильной синхронизации работы потоков.

Функции C++ для создания и завершения потока

Стандартные API-функции операционной системы Windows `CreateThread()` и `ExitThread()`, которые мы использовали для создания и завершения потока, разрабатывались еще для Windows 3.1 и имеют некоторые проблемы, связанные с "утечкой" памяти, а также с обработкой статических переменных библиотечными функциями. Например, несколько потоков могут одновременно изменять внутреннюю статическую переменную функции `strtok()`, в которой хранится указатель на текущую строку.

Для решения подобных коллизий была разработана новая многопоточная библиотека. Прототипы новых функций, предназначенных для создания и завершения потока, размещены в файле включений `process.h`.

Функция `_beginthreadex()` используется для создания потока:

```
unsigned int __cdecl _beginthreadex(void *secAttr, unsigned stackSize,
       unsigned (__stdcall *threadFunc) (void *), void *param,
       unsigned flags, unsigned *ThreadId);
```

Все параметры функции имеют то же значение, что и у функции `CreateThread()`, но типы данных приведены в синтаксисе C++. Разработчики библиотеки решили сохранить верность стандартным типам C++. Но, хотя битовое представление параметров полностью совпадает, компилятор потребует явного преобразования типов. Так, возвращаемое значение функции имеет тип `unsigned long`, и его можно использовать в качестве дескриптора, тем не менее, требуется явное преобразование.

Потоковая функция здесь должна соответствовать прототипу:

```
unsigned __stdcall threadFunc(void* param);
```

Для завершения потока используется функция `_endthreadex()`, ее прототип:

```
void __cdecl _endthreadex(unsigned ExitCode);
```

Обе эти функции требуют многопоточную библиотеку времени выполнения, которая, начиная с Visual Studio 2005, используется по умолчанию.

Для иллюстрации работы функции создания потока `_beginthreadex()` добавим файл включений `process.h` к нашему первому многопоточному приложению (см. листинг 6.2). Произведем следующие замены:

`CreateThread(NULL,0, MyThread, str, 0, NULL);`	`_beginthreadex(NULL, 0, MyThread, str, 0, NULL);`

`DWORD WINAPI MyThread(LPVOID param)`
`{`
`. . .`
`}`
	`unsigned __stdcall MyThread(void* param)`
	`{`
	`. . .`
	`}`

После компиляции приложение будет работать как и ранее, но использует уже новую библиотеку функций.

Примечание

На самом деле функции `_beginthreadex()` и `_endthreadex()` являются "обертками" для традиционных функций `CreateThread()` и `ExitThread()`. Они предварительно выделяют память под статические переменные создаваемого потока и очищают ее при завершении этого потока.

Измерение времени работы потока

Измерение времени выполнения некоторого участка кода представляется не такой уж простой задачей, поскольку квант времени, выделенный потоку, может завершиться в любой момент. Поэтому, если мы определим два момента времени и найдем их разность, то не можем быть уверены, что это и есть чистое время выполнения кода потока. Между этими моментами поток мог неоднократно терять управление. К счастью, имеется функция `GetThreadTimes()`, которая позволяет определить, сколько времени поток затратил на выполнение операций. Ее прототип размещен в файле включений `process.h`.

```
BOOL WINAPI GetThreadTimes(HANDLE hThread, LPFILETIME lpCreationTime,
LPFILETIME lpExitTime, LPFILETIME lpKernelTime, LPFILETIME lpUserTime);
```

Функция принимает дескриптор потока и 4 указателя на переменные, возвращающие временные показатели потока в единицах по 100 нс (т. е. 10^{-7}с):

- `CreationTime` — время создания потока с 1.01.1601;
- `ExitTime` — время завершения потока с 1.01.1601;
- `KernelTime` — время выполнения кода операционной системы;
- `UserTime` — время выполнения кода приложения.

Поскольку для хранения времени в таком формате 32 бит не хватает, для этих целей выделяются две 32-битных переменных структуры `FILETIME`.

```
struct FILETIME { DWORD dwLowDateTime;
                  DWORD dwHighDateTime;};
```

Имеется еще одна функция `GetProcessTimes()`, которая позволит определить, сколько времени *все потоки* процесса (даже уже завершенные) затратили на выполнение задачи. В качестве первого параметра функции используется дескриптор процесса.

```
BOOL WINAPI GetProcessTimes(HANDLE hProcess, LPFILETIME lpCreationTime,
LPFILETIME lpExitTime, LPFILETIME lpKernelTime, LPFILETIME lpUserTime);
```

Для примера (листинг 6.3 и рис. 6.2) приведем оконную функцию программы, вычисляющей время работы потока.

Листинг 6.3. Измерение времени выполнения потока

```
#include <process.h>
unsigned __stdcall MyThread(void* param)
{
   for (int i = 0; i < 10000000; i++);
   return 0;
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   HANDLE hThread;
   LARGE_INTEGER Create, Exit, kernel, user;
   static __int64 kernelTime, userTime, totalTime;
   TCHAR str[256];
   RECT rt;
   switch (message)
   {
   case WM_COMMAND:
     switch (LOWORD(wParam))
     {
     case ID_THREAD:
        hThread = (HANDLE)_beginthreadex(NULL,0,MyThread,NULL,0,NULL);
        WaitForSingleObject(hThread, INFINITE);
        GetThreadTimes(hThread, (FILETIME *)&Create.u,
        (FILETIME *)&Exit.u, (FILETIME *)&kernel.u, (FILETIME *)&user.u);
        CloseHandle(hThread);
        kernelTime = kernel.QuadPart;
        userTime = user.QuadPart;
        totalTime = Exit.QuadPart - Create.QuadPart;
        InvalidateRect(hWnd, NULL, TRUE);
        break;
```

```
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
    case WM_PAINT:
      GetClientRect(hWnd, &rt);
      hdc = BeginPaint(hWnd, &ps);
      _stprintf(str, _T("kernelTime = %I64d\nuserTime  = %I64d\ntotalTime\
     = %I64d"), kernelTime, userTime, totalTime);
      DrawText(hdc, str, _tcslen(str), &rt, DT_LEFT);
      EndPaint(hWnd, &ps);
      break;
    case WM_DESTROY: PostQuitMessage(0); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

При обработке пункта меню с идентификатором `ID_THREAD` создаем поток функцией `_beginthreadex()` и функцией `WaitForSingleObject()` ждем его завершения. Здесь мы забегаем немного вперед, об этой функции будем говорить при обсуждении вопроса синхронизации процессов и потоков, пока лишь скажем, что функция приостановит выполнение текущего потока до завершения потока `hThread`.

После чего получим временные характеристики потока:

```
GetThreadTimes(hThread, (FILETIME *)&Create.u, (FILETIME *)&Exit.u,
               (FILETIME *)&kernel.u, (FILETIME *)&user.u);
```

Чтобы не заниматься преобразованием пары 32-битных чисел в одно 64-битовое значение, поступим проще — используем объединение `LARGE_INTEGER`, где совмещена в памяти структура `FILETIME` с одной 64-битной переменной типа `LONGLONG` (эквивалентно новому типу данных `__int64`). Теперь мы можем обращаться с переменной типа `LARGE_INTEGER`, как со структурой `FILETIME`, рассматривая поле `u`, или как с 64-разрядным числом `LONGLONG` в поле `QuadPart`.

> ***Примечание***
>
> В файле включений `winnt.h` определено объединение
>
> ```
> typedef union _LARGE_INTEGER
> {    struct {DWORD LowPart; LONG HighPart;};
>      struct {DWORD LowPart; LONG HighPart;} u;
>      LONGLONG QuadPart;
> } LARGE_INTEGER;
> ```

Функция `GetThreadTimes()` позволит получить три временные характеристики:

- `kernelTime = kernel.QuadPart` — время работы ядра протока;
- `userTime = user.QuadPart` — время выполнения собственного кода потока;
- `totalTime = Exit.QuadPart-Create.QuadPart` — полное время работы потока.

Результат получен в виде 64-разрядного числа типа `__int64`, для преобразования которого в текстовое представление проще всего использовать функцию `_stprintf()`, как мы это делали в листинге 1.13. Нужно только использовать формат 64-разрядного целого числа — `%I64d`.

Примечание

Можно было воспользоваться 64-разрядным аналогом функции `itoa()`:

```
char* _i64toa(__int64 Val, char *DstBuf, int Radix);
```

или

```
wchar_t* _i64tow(__int64 Val, wchar_t *DstBuf, int Radix);
```

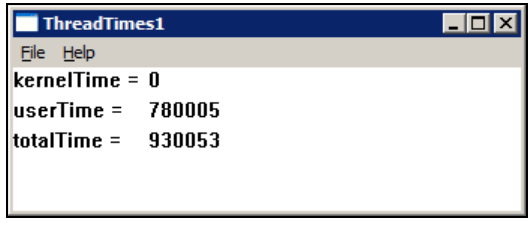


Рис. 6.2. Измерение времени работы потока

Высокоточное измерение времени

Однако функция `GetThreadTimes()` не позволит измерить малый промежуток внутри одного кванта времени (около 20 миллисекунд, 1 мс = 10^{-3}с). Для этих целей в Windows предусмотрены две функции.

Функция `QueryPerformanceFrequency()` позволит измерить частоту счетчика выполнения с высоким разрешением:

```
BOOL WINAPI QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency);
```

а функция `QueryPerformanceCounter()` возвращает его текущий отсчет.

```
BOOL WINAPI QueryPerformanceCounter(LARGE_INTEGER *lpPerformanceCount);
```

В качестве параметра используется указатель на объединение `LARGE_INTEGER`, описанное в файле включений `winnt.h`.

Обе функции возвращают `FALSE`, если не удалось найти счетчик выполнения.

В качестве примера (листинг 6.4) "измерим" время работы цикла, вид окна приложения приведен на рис. 6.3.

Листинг 6.4. Тест для высокоточного измерения временного интервала

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
```

```
   TCHAR str[60], tmp[20];
   int i, sum;
   LARGE_INTEGER frequency, Start, End;
   static __int64 totalTime;
   switch (message)
   {
   case WM_COMMAND:
     switch (LOWORD(wParam))
     {
     case ID_CYCLE:
        QueryPerformanceFrequency(&frequency);
        Sleep(0);
        QueryPerformanceCounter(&Start);
        // Измеряемый код
        for (i = sum = 0; i < 1000; i++) sum += i;
        ////////////////////////////////////////
        QueryPerformanceCounter(&End);
        totalTime = (End.QuadPart - Start.QuadPart)*1000000/
                    frequency.QuadPart;
        InvalidateRect(hWnd, NULL, TRUE);
        break;
     case IDM_EXIT: DestroyWindow(hWnd); break;
     default: return DefWindowProc(hWnd, message, wParam, lParam);
     }
   break;
   case WM_PAINT:
     hdc = BeginPaint(hWnd, &ps);
     _tcscpy(str, _T("Время работы цикла в мкс: "));
     _i64tot(totalTime, tmp, 10);
     _tcscat(str, tmp);
     TextOut(hdc, 0, 0, str, _tcslen(str));
     EndPaint(hWnd, &ps);
     break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Для хранения временных характеристик опишем переменные:

```
LARGE_INTEGER frequency, Start, End;
static __int64 totalTime;
```

Вычислительный алгоритм запустим на выполнение через меню главного окна, его идентификатор `ID_CYCLE`. Перед началом вычислений измерим частоту счетчика и снимем первый отсчет:

```
QueryPerformanceFrequency(&frequency);
QueryPerformanceCounter(&start);
```

> ***Примечание***
>
> Мы воспользовались функцией `Sleep(0)` перед первым отсчетом времени, чтобы вычисления гарантированно начались с началом выделенного потоку кванта времени. Функция `Sleep(msec)` обеспечивает задержку выполнения потока на `msec` миллисекунд, удаляя поток из очереди ожидания процессорного времени; если же аргумент равен 0, поток отдает остаток кванта времени и ждет получения следующего кванта.

По завершению цикла снимаем второй отсчет:

```
QueryPerformanceCounter(&end);
```

и вычисляем время между двумя отсчетами в микросекундах (1 мкс = 10^{-6}с):

```
totalTime = (end.QuadPart-Start.QuadPart)*1000000/frequency.QuadPart;
```

Здесь поле `QuadPart` имеет тот же тип `__int64`, что и переменная `totalTime`.

> ***Примечание***
>
> Мы использовали масштабный множитель `1000000`, чтобы результат вывести в микросекундах. Больший множитель использовать нет смысла, поскольку накладные расходы на вызов функции `QueryPerformanceCounter()` составляют около 1 мкс.

Сформируем строку для вывода в сообщении `WM_PAINT`. Для преобразования в строку 64-битного значения используем функцию `_i64tot()`.

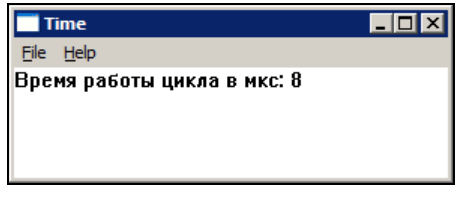


Рис. 6.3. Высокоточное измерение времени

Приоритеты потоков

Распределением процессорного времени между потоками, выполняемыми в операционной системе, занимается *планировщик потоков*, являющийся важнейшим компонентом Windows. Каждому потоку при запуске присваивается свой уровень *приоритета* (от 0 — самый низкий приоритет, до 31 — самый высокий приоритет). Причем 0 приоритет зарезервирован за специальным системным потоком "очистки свободных страниц памяти" и пользовательским потокам не присваивается.

Уровень приоритета потока складывается из двух компонентов: класса приоритета процесса и относительного приоритета потока. Рассмотрим их подробно.

В табл. 6.1 приведены идентификаторы, которыми можно определить класс приоритета процесса при его создании в параметре `dwCreationFlags` функции `CreateProcess()`. Этот класс будет применяться для всех потоков, созданных процессом.

Таблица 6.1. *Классы приоритета процесса*

Класс приоритета	Идентификатор	Значение
Real–time (реального времени)	`REALTIME_PRIORITY_CLASS`	`0x00000100`
High (высокий)	`HIGH_PRIORITY_CLASS`	`0x00000080`
Above normal (выше обычного)	`ABOVE_NORMAL_PRIORITY_CLASS`	`0x00008000`
Normal (обычный)	`NORMAL_PRIORITY_CLASS`	`0x00000020`
Below normal (ниже обычного)	`BELOW_NORMAL_PRIORITY_CLASS`	`0x00004000`
Idle (простаивающий)	`IDLE_PRIORITY_CLASS`	`0x00000040`

По умолчанию (параметр `dwCreationFlags = 0`) задается класс приоритета `Normal` и большинство процессов создается именно таким образом. Не следует задавать процессу слишком высокий класс — это может сказаться на производительности системы в целом. Необходимо помнить, что класс Real-time используется лишь в исключительных случаях.

Функция `GetPriorityClass()`:

```
DWORD WINAPI GetPriorityClass(HANDLE hProcess);
```

возвращает информацию о классе приоритета процесса, а другая функция `SetPriorityClass()`:

```
BOOL WINAPI SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
```

позволяет изменить класс приоритета работающего приложения, который будет действовать лишь на вновь созданные потоки.

Определить класс приоритета текущего процесса можно следующим образом:

```
DWORD Priority = GetPriorityClass(GetCurrentProcess());
```

где функция `GetCurrentProcess()` вернет дескриптор текущего процесса.

Изменить класс приоритета процесса после его создания можно так:

```
SetPriorityClass(pInfo.hProcess, HIGH_PRIORITY_CLASS);
```

Однако класс приоритета процесса является лишь базовым уровнем, относительно которого определяется приоритет потока. При создании потока функцией `CreateThread()` он всегда создается с приоритетом Normal, который задает базовый уровень для текущего класса.

Для изменения приоритета созданного потока используется функция `SetThreadPriority()`:

```
BOOL WINAPI SetThreadPriority(HANDLE hThread, int nPriority);
```

где `nPriority` — относительный приоритет потока.

Приведем для справки в табл. 6.2 приоритеты потоков из книги Дж. Рихтера.

Таблица 6.2. *Приоритеты потоков*

Относительный приоритет потока	Класс приоритета потока					
	Idle	**Below Normal**	**Normal**	**Above Normal**	**High**	**Real-time**
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	14	25
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Можно задать 7 различных приоритетов потока, а абсолютное значение складывается из базового приоритета класса и относительного приоритета потока. Из таблицы видно, что с относительным приоритетом Time-critical для всех классов, кроме Real-time, абсолютный приоритет — 15, а с приоритетом Idle — 1. Это сделано, чтобы ограничить область *"динамического приоритета"*, в которой должны находиться пользовательские потоки.

Идентификаторы, необходимые для установки относительного приоритета потока, определены в файле включений `winbase.h`, их значения приведены в табл. 6.3.

Таблица 6.3. *Идентификаторы относительных приоритетов потока*

Относительный приоритет потока	Идентификатор	Значение
Time-critical	`THREAD_PRIORITY_TIME_CRITICAL`	15
Highest	`THREAD_PRIORITY_HIGHEST`	2
Above normal	`THREAD_PRIORITY_ABOVE_NORMAL`	1
Normal	`THREAD_PRIORITY_NORMAL`	0
Below normal	`THREAD_PRIORITY_BELOW_NORMAL`	-1
Lowest	`THREAD_PRIORITY_LOWEST`	-2
Idle	`THREAD_PRIORITY_IDLE`	-15

Чтобы создать поток с относительным приоритетом, например, Highest, обычно создают "приостановленный поток", изменяют его приоритет и запускают на выполнение функцией `ResumeThread()`:

```
DWORD WINAPI ResumeThread(HANDLE hThread);
```

Например,

```
HANDLE hThread;
hThread = CreateThread(NULL, 0, MyThread, NULL, CREATE_SUSPENDED, NULL);
SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST);
ResumeThread(hThread);
```

Текущее значение относительного приоритета потока можно узнать из возвращаемого значения функции `GetThreadPriority()`:

```
int WINAPI GetThreadPriority(HANDLE hThread);
```

В нашем случае, если бы мы добавили в предыдущий пример строку:

```
int priority = GetThreadPriority(hThread);
```

то получили бы значение равное `2`. Однако истинное значение может быть больше, и это связано с тем, что планировщик потоков может самостоятельно повышать приоритет потоков, но только в области приоритетов 1—15, поэтому эту область и называют областью динамического приоритета. Единственное, в чем мы можем быть уверены, так это в том, что приоритет потока не ниже установленного.

> ***Примечание***
>
> Можно изменить приоритет и у работающего потока.

Планировщик потоков просматривает все потоки, претендующие на процессорное время (потоки, находящиеся в режиме ожидания, исключаются из распределения). Просматриваются потоки, начиная со старшего приоритета — 31, и, как только будет найден поток, ожидающий очереди, ему будет отдан квант времени (около 20 мс). Очевидно, чем меньше приоритет потока, тем меньше у него шансов на получение кванта времени, и "сборщик мусора" с приоритетом 1 получит управление только тогда, когда системе совсем уже нечем будет заняться. С другой стороны, если мы зададим слишком высокий приоритет, то можем заблокировать некоторые системные утилиты, что может привести к "тупиковой" ситуации.

Синхронизация потоков в пользовательском режиме

Операционная система поддерживает несколько механизмов, регулирующих доступ процессов и потоков к разделяемым ресурсам. Самым простым и быстрым способом разделения ресурсов является использование *блокирующих* (*Interlocked*) *функций*, а также основанного на них механизма критических секций.

Interlocked-функции

Нередко возникает простая задача синхронизации, когда нескольким потокам требуется изменить значение одной глобальной переменной. В этом случае можно решить задачу, используя Interlocked-функции. Эти функции называют *атомарными*, имея в виду *неделимость* операции: они выполняются очень быстро, отнимая примерно 50 тактов процессорного времени.

Представим ситуацию:

```
long variable = 0;
DWORD WINAPI Thread1(PVOID param)
{
  variable++;
  return 0;
}
DWORD WINAPI Thread2(PVOID param)
{
  variable++;
  return 0;
}
```

Два потока пытаются увеличить значение переменной `variable`. Допустим, первый поток считывает значение переменной `variable`, равное `0`, и в этот момент произойдет передача управления второму потоку, который также считывает значение переменной `variable`, равное `0`, увеличивает значение на `1` и записывает результат обратно в глобальную переменную. Теперь первый поток снова получает управление, увеличивает значение на `1` и записывает результат обратно в переменную.

После завершения работы потоков переменная `variable` равна `1`, а не `2`, как нам бы хотелось.

Для того чтобы подобная операция выполнилась корректно, необходимо на время операции чтения/записи заблокировать обращение к переменной со стороны другого потока. Это достигается в Interlocked-функциях, которые устанавливают на время операции специальный флаг, указывающий, что данный адрес памяти заблокирован:

- `LONG InterlockedExchangeAdd(PLONG plAddend, LONG lIncrement);`

  функция позволяет увеличить значение переменной `*plAddend` на величину `lIncrement` (или уменьшить, если параметр имеет отрицательное значение);

- `LONG InterlockedExchange(PLONG plTarget, LONG lVal);`

  функция заменяет значение переменной `*plTarget` на `lVal`;

- `PVOID InterlockedExchangePointer(PVOID* ppvTarget, PVOID pvVal);`

  функция заменяет значение указателя `*ppvTarget` на `pvVal`.

Возвращаемым значением функций является новое значение переменной, на которую указывает первый параметр.

> ***Примечание***
>
> Имеется дополнительный набор атомарных функций, реализованных в операционной системе Vista и Windows 7, в том числе их 64-битные аналоги. Подробнее в MSDN.

Если вернуться к нашему примеру, то нужно было бы сделать так:

```
DWORD WINAPI Thread1(PVOID param)
{
  InterlockedExchangeAdd(&variable, 1);
  return 0;
```

```
}
DWORD WINAPI Thread2(PVOID param)
{
  InterlockedExchangeAdd(&variable, 1);
  return 0;
}
```

Для двух других функций постройте пример самостоятельно.

Критические секции (critical section)

Критические секции реализованы по принципу "эстафетной палочки", которой обмениваются потоки, проверяя доступность ресурса, и являются наиболее экономичным способом синхронизации потоков.

Вначале необходимо объявить переменную типа структуры `CRITICAL_SECTION` на глобальном уровне:

```
CRITICAL_SECTION CriticalSection;
```

Перед использованием переменная должна быть инициализирована функцией `InitializeCriticalSection()`:

```
VOID WINAPI InitializeCriticalSection(LPCRITICAL_SECTION lpCrSection);
```

Для работы с критическими секциями используются две функции, которые принимают в качестве параметра указатель на структуру `CRITICAL_SECTION`:

```
VOID WINAPI EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
VOID WINAPI LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Перед началом "критического" участка кода необходимо вызывать функцию `EnterCriticalSection()`, которая проверяет доступность ресурса. Если ресурс доступен, он блокируется, а код выполняется, иначе — работа потока приостанавливается до освобождения ресурса.

В конце "критического" участка кода необходимо объявить критическую секцию доступной при помощи функции `LeaveCriticalSection()`.

Когда надобность в критической секции отпадет, можно удалить ее функцией `DeleteCriticalSection()`:

```
VOID WINAPI DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

Рассмотрим простой пример: отдельный поток создается для чтения текстового файла, вывод в окно осуществляется в сообщении `WM_PAINT` (листинг 6.5). Код примера в основном повторяет код, использованный в задаче чтения текстового файла (см. листинг 2.1—2.3).

Листинг 6.5. Критическая секция для двух потоков

```
#include <process.h>
#include <fstream>
#include <vector>
#include <string>
```

```
std::vector<std::string> v;
CRITICAL_SECTION fs;
unsigned __stdcall Thread(void* param)
{
   std::ifstream in;
   std::string st;
   EnterCriticalSection(&fs);
   in.open(_T("readme.txt"));
   while (getline(in, st)) v.push_back(st);
   in.close();
   LeaveCriticalSection(&fs);
   InvalidateRect((HWND)param, NULL, TRUE);
   return 0;
}
////////////////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   HDC hdc;
   PAINTSTRUCT ps;
   std::vector<std::string>::iterator it;
   int y;
   static HANDLE hThread;
   switch (message)
   {
   case WM_CREATE:
      InitializeCriticalSection(&fs);
      hThread = (HANDLE)_beginthreadex(NULL, 0, Thread, hWnd, 0, NULL);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      EnterCriticalSection(&fs);
      for (y = 0, it = v.begin(); it < v.end(); ++it, y += 16)
         TabbedTextOutA(hdc, 0, y, it->data(), it->length(), 0, NULL, 0);
      LeaveCriticalSection(&fs);
      EndPaint(hWnd, &ps);
```

```
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Переменную `fs` типа `CRITICAL_SECTION` описываем на глобальном уровне, здесь же опишем контейнер `vector` для типа `string`, где будем хранить строки. Инициализацию критической секции проводим в сообщении `WM_CREATE` перед созданием потока.

А в функцию потока и обработчик сообщения `WM_PAINT` вставим в качестве обложек пару функций:

```
EnterCriticalSection(&fs);
. . .
LeaveCriticalSection(&fs);
```

что и обеспечит нам последовательность выполнения критического участка кода. Действительно, если файл еще не прочитан, выводить его в окно рано, в то же время при перерисовке окна лучше воздержаться от продолжения ввода.

Поскольку в функции потока нам понадобится дескриптор окна `hWnd`, передадим его параметром потоковой функции, однако потребуется явное преобразование типа:

```
InvalidateRect((HWND)param, NULL, TRUE);
```

Синхронизация с использованием объектов ядра

Interlocked-функции и критические секции, несмотря на их простоту и скорость работы, не обеспечивают достаточной гибкости для синхронизации потоков. Они неприменимы для синхронизации процессов. В этом случае приходится использовать объекты ядра, которые могут находиться в свободном и занятом состоянии.

Перечислим объекты ядра: *процессы, потоки, задания, файлы, события, ожидаемые таймеры, семафоры, мьютексы*.

При работе с объектами ядра нужно иметь в виду, что они принадлежат не создавшему их процессу, а операционной системе. Именно это обстоятельство и позволяет получить доступ к ним из разных процессов.

Функции для работы с объектами ядра требуют переключения в режим ядра, что отнимает примерно 1000 тактов процессорного времени, поэтому являются "медленными" операциями.

Набор Wait-функций позволяет приостанавливать выполнение потока до освобождения объекта ядра. Чаще всего используется функция:

```
DWORD WINAPI WaitForSingleObject(
    HANDLE hHandle,              //дескриптор объекта ядра
    DWORD dwMilliseconds);       //время ожидания
```

Функция ожидает освобождения объекта ядра `dwMilliseconds` миллисекунд (значение параметра `INFINITE` используется для неограниченного ожидания) и, если по истечении этого времени объект не был освобожден, возвращает значение `WAIT_TIMEOUT`. При успешном завершении функция возвращает нулевое значение `WAIT_OBJECT_0`.

Если объект ядра `hHandle` занят, функция `WaitForSingleObject()` переводит поток в режим ожидания (т. е. поток "засыпает" и не участвует более в распределении процессорного времени), если же объект свободен, он переводится в занятое состояние, и выполняются следующие инструкции программного кода.

Имеется еще одна функция, которая позволяет ожидать освобождения сразу нескольких объектов:

```
DWORD WINAPI WaitForMultipleObjects(
    DWORD dwCount,                //количество объектов ядра
    CONST HANDLE* phObjects,      //массив дескрипторов
    BOOL fWaitAll,                //режим работы
    DWORD dwMilliseconds);        //время ожидания
```

Если `fWaitAll = TRUE`, функция будет ждать освобождения ***всех*** объектов, иначе — поток будет возобновлен при освобождении ***любого*** объекта из массива `phObjects`, а возвращаемое значение определит индекс объекта в массиве `phObjects`.

Семафоры

Семафоры реализованы как счетчики, которые увеличивают свое значение, когда ресурс освобождается, и уменьшают это значение, когда какой-либо поток потребует ресурс. Создаются семафоры функцией `CreateSemaphore()`:

```
HANDLE WINAPI CreateSemaphoreW(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,//атрибуты доступа
    LONG lInitialCount,                         //счетчик
    LONG lMaximumCount,                         //количество задач
    LPCWSTR lpName);                            //имя семафора
```

Семафоры являются глобальными системными объектами и, если два различных процесса открывают семафор с одним именем, используют один и тот же объект. Если вместо имени указан `NULL`, создается локальный для данного процесса семафор.

Параметр `lMaximumCount` задает максимальное число потоков, которые могут работать с объектом. При значении `lMaximumCount = 1` семафор называют *исключающим*.

`lInitialCount` — начальное значение счетчика обращений. Если счетчик равен `0`, все функции, ожидающие семафор, будут блокированы, пока значение счетчика не увеличится. Счетчик изменяется в пределах `[0; lMaximumCount]`.

Функция возвращает дескриптор созданного семафора или `0` в случае ошибки.

Открыть существующий семафор можно функцией `OpenSemaphore()`:

```
HANDLE WINAPI OpenSemaphoreW(
    DWORD dwDesiredAccess, //права доступа, обычно SEMAPHORE_ALL_ACCESS
```

```
    BOOL bInheritHandle,    //если TRUE, дескриптор может быть унаследован
    LPCWSTR lpName);        //имя семафора
```

Если семафор найден, функция возвращает его дескриптор, иначе — `0`.

Следующая функция освобождает семафор, позволяя использовать его другому процессу или потоку (функция просто увеличивает счетчик семафора на `lReleaseCount`).

```
BOOL WINAPI ReleaseSemaphore(
    HANDLE hSemaphore,        //дескриптор семафора
    LONG lReleaseCount,       //значение будет добавлено к счетчику
    LPLONG lpPreviousCount); //сохраняет предыдущее значение счетчика
```

Построим простой пример (листинг 6.6), где в диалоговом окне разместим два элемента управления Progress Bar (индикатора выполнения) и создадим два потока, каждый из которых управляет своим индикатором. Для наглядности отобразим стандартный светофор.

Работать с семафором можно при помощи функций `WaitForSingleObject()` или `WaitForMultipleObjects()`.

> ***Примечание***
>
> Потоки будем теперь создавать функцией `_beginthreadex()`. Нужно добавить файл включений `process.h`.

Листинг 6.6. Демонстрация работы семафора

```
#include <process.h>
#include <commctrl.h>
HANDLE hSemaphore;
INT_PTR CALLBACK Dialog1(HWND, UINT, WPARAM,LPARAM);
////////////////////////////////////////////
HWND SetElement(HWND hDlg, HDC& mem, WORD IdBitmap, WORD IdProgress)
{
  HBITMAP hBitmap  = LoadBitmap(hInst, MAKEINTRESOURCE(IdBitmap));
  HDC hdc = GetDC(hDlg);
  mem = CreateCompatibleDC(hdc);
  ReleaseDC(hDlg, hdc);
  SelectObject(mem, hBitmap);
  HWND handle = GetDlgItem(hDlg, IdProgress);
  SendMessage(handle, PBM_SETRANGE, 0, 30<<16);
  SendMessage(handle, PBM_SETSTEP, 1, 0);
  SendMessage(handle, PBM_SETPOS, 0, 0);
  return handle;
}
////////////////////////////////////////////
unsigned __stdcall MyThread1(void* param)
```

```
{
  HWND hWnd = (HWND)param;
  HDC hdc, mem;
  int t = 0;
  HWND hProgress = SetElement(hWnd, mem, IDB_BITMAP1, IDC_PROGRESS1);
  while(1)
  {
    WaitForSingleObject(hSemaphore, INFINITE);
    Sleep(500);
    hdc = GetDC(hWnd);
    BitBlt(hdc, 320, 25, 25, 50, mem, 0, 0, SRCCOPY);
    ReleaseDC(hWnd, hdc);
    if (++t > 30) t = 0;
    SendMessage(hProgress, PBM_SETPOS, t, 0);
    ReleaseSemaphore(hSemaphore, 1, NULL);
  }
  return 0;
}
////////////////////////////////////////////
unsigned __stdcall MyThread2(void* param)
{
  HWND hWnd = (HWND)param;
  HDC hdc, mem;
  int t = 0;
  HWND hProgress = SetElement(hWnd, mem, IDB_BITMAP2, IDC_PROGRESS2);
  while(1)
  {
    WaitForSingleObject(hSemaphore, INFINITE);
    Sleep(500);
    hdc = GetDC(hWnd);
    BitBlt(hdc, 320, 25, 25, 50, mem, 0, 0, SRCCOPY);
    ReleaseDC(hWnd, hdc);
    if (++t > 30) t = 0;
    SendMessage(hProgress, PBM_SETPOS, t, 0);
    ReleaseSemaphore(hSemaphore, 1, NULL);
  }
  return 0;
}
////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
  switch (message)
```

```
  {
  case WM_COMMAND:
    switch (LOWORD(wParam))
    {
    case ID_SEMAPHORE:
      DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, Dialog1);
      break;
    case IDM_EXIT: DestroyWindow(hWnd); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
  case WM_DESTROY: PostQuitMessage(0); break;
  default: return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
//////////////////////////////////////////////
INT_PTR CALLBACK Dialog1(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
  static HANDLE hThread1, hThread2;
  switch (message)
  {
    case WM_INITDIALOG:
      InitCommonControls();
      hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);
      hThread1=(HANDLE)_beginthreadex(NULL, 0, MyThread1, hDlg, 0, NULL);
      hThread2=(HANDLE)_beginthreadex(NULL, 0, MyThread2, hDlg, 0, NULL);
      return TRUE;
    case WM_COMMAND:
      if (LOWORD(wParam) == IDCANCEL)
      {
        TerminateThread(hThread1, 0);
        TerminateThread(hThread2, 0);
        CloseHandle(hThread1);
        CloseHandle(hThread2);
        EndDialog(hDlg, 0);
        return TRUE;
      }
      break;
  }
return FALSE;
}
```

Для использования элемента управления Progress Bar (индикатор выполнения) необходимо добавить файл включений `commctrl.h` и библиотеку `comctl32.lib`, а также инициировать общие элементы управления обращением к функции `InitCommonControls()`.

Для отображения индикатора выполнения подготовим в редакторе ресурсов два рисунка с изображением светофора в разных состояниях размером 25×50, их идентификаторы — `IDB_BITMAP1` и `IDB_BITMAP2`.

Дескриптор семафора опишем на глобальном уровне:

```
HANDLE hSemaphore;
```

В оконной функции главного окна создадим пункт меню **START**, при обработке которого вызываем диалоговое окно:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, Dialog1);
```

В диалоговом окне разместим два индикатора выполнения с идентификаторами `IDC_PROGRESS1` и `IDC_PROGRESS2`.

При открытии диалогового окна инициируем библиотеку общих элементов управления:

```
InitCommonControls();
```

Определим локальный для процесса семафор:

```
hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);
```

Семафор определяем как *исключающий*, `lMaximumCount = 1`. Начальное значение счетчика равно `1`, т. е. семафор изначально свободен.

А теперь можно создать два потока:

```
hThread1 = (HANDLE)_beginthreadex(NULL, 0, MyThread1, hDlg, 0, NULL);
hThread2 = (HANDLE)_beginthreadex(NULL, 0, MyThread2, hDlg, 0, NULL);
```

передавая им в качестве параметра дескриптор диалогового окна.

Потоки сразу начинают выполняться, поскольку параметр `dwCreationFlags = 0`.

Функции потоков построены симметрично, поэтому рассмотрим только одну из них, например, `MyThread1()`.

Настройку элемента управления "индикатор выполнения", а также загрузку растрового изображения выполним в отдельной функции `SetElement()`, которая вернет дескриптор соответствующего элемента Progress Bar.

Установим диапазон индикатора от 0 до 30 (верхняя граница в старшем слове `lParam`, нижняя — в младшем). Сформируем значение логическим сдвигом — `30<<16`. Шаг индикатора установим в `1`, а начальное значение в `0`.

Потоковая функция построена как бесконечный цикл `while(1) …` и может быть остановлена лишь извне. Внутри цикла ждем освобождения семафора:

```
WaitForSingleObject(hSemaphore, INFINITE);
```

После открытия семафора установим задержку в 0,5 сек, чтобы можно было визуально наблюдать процесс его переключения. Получим контекст устройства вывода и скопируем картинку из контекста памяти на поверхность диалогового окна:

```
BitBlt(hdc, 320, 25, 25, 50, mem1, 0, 0, SRCCOPY);
```

Примечание

Координаты для вывода изображения светофора `(320,25)` зависят от размера и позиции элементов управления, а также установленного разрешения дисплея, что может привести к смещению изображения. В этом случае придется координаты скорректировать.

Следующей конструкцией:

```
if (++t > 30) t = 0;
```

обеспечим циклическое повторение переменной `t` значений из диапазона `[0;30]`.

Увеличиваем значение индикатора:

```
SendMessage(hProgress1, PBM_SETPOS, t, 0);
```

и открываем семафор:

```
ReleaseSemaphore(hSemaphore, 1, NULL);
```

Теперь второй поток может получить управление.

Так происходит переключение между двумя потоками, и мы можем наблюдать этот процесс в диалоговом окне (рис. 6.4).

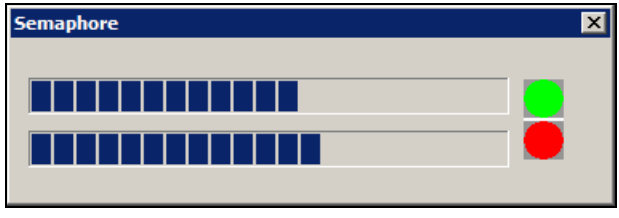


Рис. 6.4. Работа двух потоков выполнения

При нажатии на кнопку системного меню ☒ мы уничтожаем потоки, закрываем их дескрипторы `hThread1`, `hThread2`, после чего закрываем диалоговое окно функцией `EndDialog()`.

События

Поток может создать объект "Событие", перевести его в занятое состояние и выполнить какие-то операции. Любой поток, даже принадлежащий другому процессу, может проверить занятость этого события и перейти в режим ожидания до перехода события в свободное состояние.

Событие создается функцией `CreateEvent()`:

```
HANDLE WINAPI CreateEventW(
    LPSECURITY_ATTRIBUTES lpEventAttributes, //атрибуты доступа
    BOOL bManualReset,                       //режим сброса события
    BOOL bInitialState,                      //начальное состояние
    LPCWSTR lpName);                         //имя события
```

Первый и последний параметры имеют тот же смысл, что и для семафора: атрибуты доступа и глобальное имя события (или `NULL` для локального события).

Второй параметр `lpEventAttributes` определяет режим сброса события: `TRUE` — вручную, `FALSE` — автоматически.

Третий параметр `bInitialState` определяет начальное состояние события: `TRUE` — свободно, `FALSE` — занято.

Возвращаемым значением функции является дескриптор события.

Если поток другого процесса создает событие с тем же именем, то нового объекта не образуется, а реализуется доступ к уже существующему в системе объекту.

Доступ к существующему событию можно получить и функцией `OpenEvent()`:

```
HANDLE WINAPI OpenEventW(
    DWORD dwDesiredAccess, //права доступа, обычно EVENT_ALL_ACCESS
    BOOL bInheritHandle,   //если TRUE, дескриптор может быть унаследован
    LPCWSTR lpName);       //имя события
```

Функция возвращает дескриптор события или `NULL`, когда событие не найдено.

После создания события его можно перевести в свободное состояние функцией:

```
BOOL WINAPI SetEvent(HANDLE hEvent);
```

а функция `ResetEvent()` переводит событие в занятое состояние:

```
BOOL WINAPI ResetEvent(HANDLE hEvent);
```

Объект "Событие" можно удалить вызовом `CloseHandle()`.

Для иллюстрации применения этого объекта приведем фрагмент программы (листинг 6.7), которая создает два потока: один для выделения блока памяти, а другой для ее "очистки". Разумеется, эти процессы необходимо синхронизовать, поскольку "очистить" память можно лишь после ее выделения.

Листинг 6.7. Синхронизация потоков объектом Event

```
#include <process.h>
struct STR { int k; char *p;};
HANDLE hEvent;
//////////////////////////////////////
unsigned __stdcall MyThread1(void* param)
{
   ((STR*)param)->p = new char[((STR*)param)->k];
   SetEvent(hEvent);
   return 0;
}
unsigned __stdcall MyThread2(void* param)
{
   WaitForSingleObject(hEvent, INFINITE);
```

```
    for (int i = 0; i < ((STR*)param)->k; i++) ((STR*)param)->p[i] = '\0';
    SetEvent(hEvent);
    return 0;
}
////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR str[100];
    static HANDLE hThread1, hThread2;
    static STR mem = {2500000, NULL};
    switch (message)
    {
    case WM_CREATE:
         hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
         hThread1 = (HANDLE)_beginthreadex(NULL,0,MyThread1,&mem,0,NULL);
         hThread2 = (HANDLE)_beginthreadex(NULL,0,MyThread2,&mem,0,NULL);
         break;
    case WM_COMMAND:
         switch (LOWORD(wParam))
         {
         case IDM_EXIT: DestroyWindow(hWnd); break;
         default: return DefWindowProc(hWnd, message, wParam, lParam);
         }
         break;
    case WM_PAINT:
         hdc = BeginPaint(hWnd, &ps);
         _stprintf(str, _T("Память выделена по адресу: %p"), mem.p);
         TextOut(hdc, 0, 0, str, _tcslen(str));
         EndPaint(hWnd, &ps);
         break;
    case WM_DESTROY: PostQuitMessage(0); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Для передачи в функцию потока указателя на блок выделяемой памяти и ее размера опишем структуру `STR`. Тогда, создав переменную типа `STR`, можем передать ее адрес при создании потока:

```
hThread1 = (HANDLE)_beginthreadex(NULL, 0, MyThread1, &mem, 0, NULL);
```

Создавать поток мы теперь будем функцией `_beginthreadex()`, как в предыдущей задаче. Событие с автоматическим сбросом в занятом состоянии создаем при создании окна:

```
hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

а его дескриптор объявим на глобальном уровне, чтобы он был доступен в потоковых функциях.

Первый поток `MyThread1` выделяет память:

```
((STR*)param)->p = new char[((STR*)param)->k];
```

и переводит событие в свободное состояние:

```
SetEvent(hEvent);
```

Второй поток `MyThread2` ждет завершение первого и "чистит" память:

```
WaitForSingleObject(hEvent, INFINITE);
for (int i = 0; i < ((STR*)param)->k; i++) ((STR*)param)->p[i] = '\0';
```

После этого событие вновь переводится в свободное состояние:

```
SetEvent(hEvent);
```

> ***Примечание***
>
> Поскольку мы создали событие с автоматическим сбросом, то функция `WaitForSingleObject()`, закончив ожидание, вновь переведет его в занятое состояние. Таким образом, мы можем использовать это событие как признак того, что какой-то поток работает с данной областью памяти.

Мьютексы

Мьютекс (от англ. *mutex — mutual exclusion*) реализован как счетчик: он похож на исключающий семафор и обеспечивает потокам взаимоисключающий доступ к ресурсам. Мьютекс хранит идентификатор потока, счетчик числа пользователей и счетчик рекурсии. Он отличается от семафора тем, что Wait-функция проверяет значение его идентификатора, и, если он совпадает с идентификатором текущего потока, то этот поток, несмотря на то, что мьютекс занят, получит доступ к ресурсам, а счетчик рекурсии увеличится на 1. Таким образом, один поток может пользоваться ресурсами мьютекса многократно.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий — сколько раз.

Создается мьютекс функцией `CreateMutex()`, прототип которой размещен в файле включений `winbase.h`:

```
HANDLE WINAPI CreateMutexW(
  LPSECURITY_ATTRIBUTES lpMutexAttributes, //атрибуты доступа
  BOOL bInitialOwner,                      //начальное состояние
  LPCWSTR lpName);                         //имя мьютекса
```

Существующий мьютекс может быть открыт функцией `OpenMutex()`:

```
HANDLE WINAPI OpenMutexW(
  DWORD dwDesiredAccess,     //права доступа
  BOOL bInheritHandle,       //начальное состояние
  LPCWSTR lpName);           //имя мьютекса
```

Параметр `bInitialOwner` определяет начальное состояние мьютекса. Если он равен `FALSE` — мьютекс создается в свободном состоянии, его идентификатор равен `0`, также равен `0` счетчик рекурсий. Если же параметр равен `TRUE`, мьютекс занят текущим потоком, при этом его идентификатору присваивается значение идентификатора текущего потока, счетчику рекурсий присваивается значение `1`.

Функция `ReleaseMutex()`:

```
BOOL WINAPI ReleaseMutex(HANDLE hMutex);
```

освобождает существующий мьютекс. Поскольку поток может захватить мьютекс несколько раз, то и освободить его он должен столько же раз. Когда поток вновь захватывает мьютекс, его счетчик рекурсий увеличивается на 1, а при освобождении мьютекса уменьшается на 1. Мьютекс считается свободным, когда его счетчик рекурсий равен 0, тогда и идентификатор мьютекса автоматически принимает нулевое значение.

Для примера заменим в листинге 6.7 семафор на мьютекс и посмотрим, что из этого получится. Нам потребуется изменить лишь несколько строк в программе. Опишем дескриптор:

```
HANDLE hMutex;
```

и вместо семафора создадим мьютекс в свободном состоянии:

```
hMutex = CreateMutex(NULL, FALSE, NULL);
```

В функции `WaitForSingleObject()` используем дескриптор мьютекса:

```
WaitForSingleObject(hMutex, INFINITE);
```

и освободим мьютекс в конце бесконечного цикла потоковой функции:

```
ReleaseMutex(hMutex);
```

После всех этих изменений приложение будет работать так же, как и ранее.

Ожидаемые таймеры

Ожидаемые таймеры — это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Для создания ожидаемого таймера используется функция `CreateWaitableTimer()`:

```
HANDLE WINAPI CreateWaitableTimerW(
    LPSECURITY_ATTRIBUTES lpTimerAttributes,     //атрибуты доступа
    BOOL bManualReset,                           //тип таймера
    LPCWSTR lpTimerName);                        //имя таймера
```

Подобно другим объектам ядра, существующий таймер может быть открыт функцией `OpenWaitableTimer()`:

```
HANDLE WINAPI OpenWaitableTimerW(
    DWORD dwDesiredAccess, //права доступа
```

```
    BOOL bInheritHandle,  //если TRUE, дескриптор может быть унаследован
    LPCWSTR lpTimerName); //имя таймера
```

Параметры этих функций имеют тот же смысл, что и для прочих объектов ядра. `bManualReset` определяет тип таймера: со сбросом вручную — `TRUE`, с автосбросом — `FALSE`. Когда в свободное состояние приходит таймер с ручным сбросом, *возобновляются все потоки*, ожидающие этот таймер; для таймера с автосбросом может возобновиться лишь один поток.

Для задания времени срабатывания таймера имеется функция `SetWaitableTimer()`:

```
BOOL WINAPI SetWaitableTimer(
    HANDLE hTimer,                          //дескриптор таймера
    const LARGE_INTEGER *lpDueTime,         //время первого срабатывания
    LONG lPeriod,                           //период срабатывания
    PTIMERAPCROUTINE pfnCompletionRoutine, //функция тревожного ожидания
    LPVOID lpArgToCompletionRoutine,        //параметр функции
    BOOL fResume);                          //вывод из "спящего режима"
```

Потоки, ожидающие таймер, получают процессорное время в момент времени, задаваемый переменной `DueTime`, и в следующие, кратные `lPeriod`, моменты. Таймер может вызывать так называемую функцию "тревожного ожидания", указатель на которую передается параметром `pfnCompletionRoutine`. Эта функция получает управление при переходе таймера в свободное состояние, тогда поток, ожидающий таймер, может получить управление лишь после завершения ее работы.

Следующий аргумент `lpArgToCompletionRoutine` передается в качестве параметра функции "тревожного ожидания", которая должна быть объявлена как:

```
VOID APIENTRY pfnCompletionRoutine(LPVOID lpArgToCompletionRoutine,
DWORD dwTimerLowValue, DWORD dwTimerHighValue);
```

Два параметра функции `dwTimerLowValue` и `dwTimerHighValue` задают время срабатывания таймера и используются как младшие и старшие значения структуры `FILETIME`.

Если последний параметр `fResume = TRUE`, то таймер "разбудит" даже "спящий" компьютер, если же `fResume = FALSE` (как чаще всего и бывает), поток не получит процессорного времени, пока компьютер не выйдет из спящего режима.

> ***Примечание***
>
> Можно задать время срабатывания таймера относительно текущего момента. Время задается в переменной `DueTime` в единицах по 100 нс, но значение должно быть отрицательным. Например, так можно задать момент времени через 10 с после создания таймера: `DueTime.QuadPart = -100000000;`

Функция `CancelWaitableTimer()`:

```
BOOL WINAPI CancelWaitableTimer(HANDLE hTimer);
```

позволяет отменить таймер `hTimer`.

Для демонстрации работы ожидаемого таймера рассмотрим фрагмент программы (листинг 6.8), где мы создаем таймер и задаем время его срабатывания. Далее соз-

даем поток, который ждет таймер и, после его срабатывания, выводит диалоговое окно с текущей датой и временем.

Примечание

Функции, которые мы используем в следующем листинге, реализованы лишь в Windows NT и выше, поэтому необходимо до включения файла `windows.h` определить параметр версии Windows. Например, для Windows NT должно присутствовать определение: `#define _WIN32_WINNT 0x0400`. Иначе в файлах включений не будут найдены прототипы этих функций, и мы получим сообщение об ошибке компиляции. При работе в Visual Studio 2005 и выше об этом можно не беспокоиться, необходимый параметр установится автоматически, так, в Visual Studio 2010 и операционной системе Windows 7 его значение — `0x0601`.

Листинг 6.8. Создание ожидаемого таймера

```
#include <process.h>
HANDLE hTimer;
unsigned __stdcall MyThread(void* param)
{
   TCHAR str[30];
   SYSTEMTIME tm;
   FILETIME localTime;
   WaitForSingleObject(hTimer, INFINITE);
   GetSystemTime(&tm);
   SystemTimeToFileTime(&tm, &localTime);
   FileTimeToLocalFileTime(&localTime, &localTime);
   FileTimeToSystemTime(&localTime, &tm);
   _stprintf(str, _T("Date %2u.%2u.%4u Time %2u:%2u:%2u"), tm.wDay,
               tm.wMonth, tm.wYear, tm.wHour, tm.wMinute, tm.wSecond);
   MessageBox((HWND)param, str, _T("Включился ожидаемый таймер"), MB_OK |
               MB_ICONHAND);
   return 0;
}
//////////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   static HANDLE hThread;
   SYSTEMTIME sysTime = {2010, 5, 2, 5, 9, 04, 0, 0};
   FILETIME localTime, utcTime;
   LARGE_INTEGER *lTime;
   switch (message)
   {
   case WM_CREATE:
      hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
```

```
      SystemTimeToFileTime(&sysTime, &localTime);
      LocalFileTimeToFileTime(&localTime, &utcTime);
      lTime = (LARGE_INTEGER*)&utcTime;
      SetWaitableTimer(hTimer, lTime, 0, NULL, NULL, FALSE);
      hThread = (HANDLE)_beginthreadex(NULL, 0, MyThread, hWnd, 0, NULL);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
         case IDM_EXIT: DestroyWindow(hWnd); break;
         default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY:
      CloseHandle(hThread);
      CancelWaitableTimer(hTimer);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Дескриптор таймера опишем на глобальном уровне, чтобы иметь доступ к нему в функции потока, а таймер создадим при создании окна приложения с атрибутами доступа по умолчанию и автосбросом.

```
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
```

Теперь нужно установить время срабатывания таймера, но функция `SetWaitableTimer()` принимает время типа `LARGE_INTEGER`. Однако задание момента времени в формате 64-разрядного целого числа является занятием крайне неблагодарным, поэтому мы определим переменную типа структуры `SYSTEMTIME`:

```
struct SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;};
```

и зададим начальный момент времени так:

```
SYSTEMTIME sysTime = {2010, 5, 2, 5, 9, 04, 0, 0};
```

Теперь нужно преобразовать формат времени к типу FILETIME.

```
SystemTimeToFileTime(&sysTime, &localTime);
```

Почему именно к этому типу? Дело в том, что имеется еще одна проблема — функция SetWaitableTimer() работает с форматом времени, отсчитываемым по 100 нс (1 нс = 10^{-9} с) от 1 января 1601 г., причем еще приведенному к нулевому меридиану *(Universal Coordinated Time — UTC)*, а мы обычно задаем местное (локальное) время. Нужно это время преобразовать в UTC. Функция, которая это делает, принимает параметры типа FILETIME.

```
LocalFileTimeToFileTime(&localTime, &utcTime);
```

Однако нам все еще нужно время типа LARGE_INTEGER. Привести к нему можно явным преобразованием типа:

```
lTime = (LARGE_INTEGER*)&utcTime;
```

Вот теперь можно установить время срабатывания таймера:

```
SetWaitableTimer(hTimer, lTime, 0, NULL, NULL, FALSE);
```

и создать поток:

```
hThread = (HANDLE)_beginthreadex(NULL, 0, MyThread, hWnd, 0, NULL);
```

Посмотрим, как будет работать вновь созданный поток. Получив управление, он будет ждать освобождения таймера:

```
WaitForSingleObject(hTimer, INFINITE);
```

Как только таймер сработал, поток получает управление и определяет текущее время:

```
GetSystemTime(&tm);
```

и опять же в переменной типа SYSTEMTIME. Теперь нужно провести обратное преобразование к локальному времени:

```
SystemTimeToFileTime(&tm, &localTime);
FileTimeToLocalFileTime(&localTime, &localTime);
```

Ну и, наконец, возвращаемся к формату SYSTEMTIME:

```
FileTimeToSystemTime(&localTime, &tm);
```

> ***Примечание***
>
> Можно, конечно, сразу получить время в формате SYSTEMTIME, воспользовавшись функцией GetLocalTime(), однако хотелось показать весь набор функций преобразования времени.

Для вывода диалогового окна нужно сформировать строку с датой и временем. Проще всего это сделать в функции sprintf():

```
_stprintf(str, _T("Date %2u.%2u.%4u Time %2u:%2u:%2u"), tm.wDay,
              tm.wMonth, tm.wYear, tm.wHour, tm.wMinute, tm.wSecond);
```

где мы для вывода беззнаковых чисел целого типа используем формат "%u",

а сам диалог организуем стандартным образом (рис. 6.5):

```
MessageBox((HWND)param, str, _T("Включился ожидаемый таймер"), MB_OK |
              MB_ICONHAND);
```

Здесь в первом параметре необходимо указать дескриптор окна приложения, который мы передали потоку при его создании. Пришлось, однако, использовать явное преобразование типа `(HWND)param`.

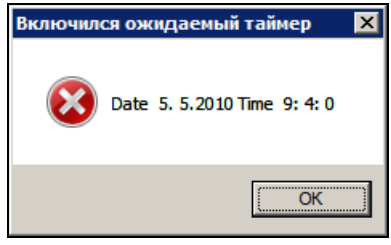


Рис. 6.5. Сработал "Ожидаемый таймер"

Обмен данными между процессами

Операционная система Windows "изолирует" процессы, преследуя благую цель — повышение устойчивости системы. Однако различным процессам, тем не менее, часто требуется обмениваться данными. Осуществлять же обмен путем записи в файл, а затем чтением из этого файла, не очень эффективно. Разумеется, были разработаны различные механизмы обмена данными. Мы рассмотрим несколько технологий обмена данными между процессами, которые часто используются при построении сложных информационных систем.

Разделяемая память для нескольких экземпляров exe-файла

Начнем с задачи определения количества загруженных копий приложения. Проблема здесь заключается в том, что каждый экземпляр приложения представляет собой отдельный процесс со своим адресным пространством, и эти пространства совершенно независимы. Для того чтобы подойти к решению этой задачи, рассмотрим, не вдаваясь в детали, как организовано это адресное пространство.

В Windows NT и более поздних версиях каждый процесс получает 2 Гбайт виртуального адресного пространства, где расположен программный код, статические данные, стек и куча. Разумеется, используется не все адресное пространство, а только та его часть, которая реально необходима процессу. Это виртуальное адресное пространство проецируется на физическую память, организованную постранично, а поскольку физической памяти всегда не хватает, то часть ее может помещаться в файл подкачки. Файл подкачки организован так, чтобы в один кластер на диске помещалась страница памяти, кстати, это и является причиной различных размеров страниц памяти (4, 8, 16K). Причем программный код exe-файла, а также DLL-библиотеки не записываются в файл подкачки: при необходимости они могут быть восстановлены из исходных файлов.

Примечание

Именно это обстоятельство приводит к запрету модификации программного кода исполняемого модуля, поскольку в любой момент времени код программы может быть восстановлен с диска в исходном состоянии.

Глобальные же и статические переменные загружаются в область памяти, которая может быть "сброшена" в файл подкачки.

Однако при загрузке приложения можно создать *разделяемую* область памяти, которая будет совместно использоваться всеми его экземплярами. Для этого нужно объявить именованный раздел памяти директивой:

```
#pragma data_seg("имя_раздела")
```

после чего объявить *инициализированные* переменные (если не присвоить начальное значение, переменная не будет размещена в именованном разделе), например, объявим:

```
volatile long shared_var = 0;
```

и закроем раздел той же директивой, но без параметров.

```
#pragma data_seg()
```

Примечание

Модификатор `volatile` здесь не обязателен, но уместен, он говорит компилятору о том, что значение переменной может быть изменено извне. В этом случае компилятор должен воздержаться от оптимизации участков кода, содержащих эту переменную.

Но это еще не все. Нужно дать указание компоновщику для создания раздела памяти с соответствующими правами доступа. Проще всего это сделать, включив в текст директиву препроцессора:

```
#pragma comment(linker, "/Section:имя_раздела,RWS")
```

что эквивалентно использованию соответствующего ключа в командной строке компоновщика.

Здесь ключи доступа `RWS (Read, Write, Shared)` означают, что раздел памяти доступен для чтения, записи и является разделяемым.

Итак, сводя все сказанное вместе, приведем фрагмент программы (листинг 6.9), которая показывает в заголовке окна номер загруженного экземпляра, используя переменную из общей области памяти.

Листинг 6.9. Индикация номера экземпляра приложения

```
#pragma data_seg("Shared")
volatile int shared_var = 0;
#pragma data_seg()
#pragma comment(linker, "/Section:Shared,RWS")
////////////////////////////////////////////
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   TCHAR str[256], name[256];
```

```
    switch (message)
    {
      case WM_CREATE:
         shared_var++;
         SendMessage(hWnd, WM_GETTEXT, 256, (LPARAM)name);
         _stprintf(str, _T("%s - Экземпляр: %d"), name, shared_var);
         SendMessage(hWnd, WM_SETTEXT, 0, (LPARAM)str);
         break;
    case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
    case WM_DESTROY: PostQuitMessage(0); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

При создании главного окна приложения увеличим значение переменной `shared_var` на 1. Хотя нам известен заголовок окна, покажем, как можно его прочитать, передавая окну сообщение `WM_GETTEXT`. Указатель на строку заголовка получим в `lParam` функции `SendMessage()`:

```
SendMessage(hWnd, WM_GETTEXT, 256, (LPARAM)name);
```

где в `wParam` передаем максимальный размер строки.

Сформируем новый заголовок из исходного, добавляя к нему строку `" Экземпляр:"` и номер экземпляра из переменной `shared_var`:

```
_stprintf(str, _T("%s - Экземпляр: %d"), name, shared_var);
```

Новый заголовок получим, передавая окну сообщение `WM_SETTEXT`:

```
SendMessage(hWnd, WM_SETTEXT, 0, (LPARAM)str);
```

Файлы, проецируемые в память

Прежде чем приступать к рассмотрению техники обмена данными между процессами, рассмотрим проецирование дискового файла в память. Идея заключается в том, что для дискового файла выделяется регион виртуального адресного пространства и сопоставляется начало этого региона начальному байту файла. После чего, открывая файл, мы работаем в оперативной памяти в адресном пространстве процесса, а операционная система использует область файла на диске как файл подкачки, автоматически внося в него все производимые изменения.

Обычно к технике проецирования файла в память прибегают для экономии памяти и ускорения доступа. Создание проекции файла в память выполняется в 3 этапа.

1. Создание или открытие файла

Для открытия проецируемого в память дискового файла используется только функция `CreateFile()`:

```
HANDLE WINAPI CreateFileW(
    LPCWSTR lpFileName,                          //имя файла
    DWORD dwDesiredAccess,                       //способ доступа
    DWORD dwShareMode,                           //совместный доступ
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, //атрибуты доступа
    DWORD dwCreationDisposition,                 //проверка существования
    DWORD dwFlagsAndAttributes,                  //и атрибутов файла
    HANDLE hTemplateFile                         //временный файл
);
```

Первый параметр `FileName` — имя файла, `dwDesiredAccess` указывает способ доступа: `GENERIC_READ` или `GENERIC_WRITE`, параметры говорят сами за себя и могут комбинироваться операцией логического сложения. Параметр `dwShareMode` определяет тип совместного доступа к файлу: `0` — нет совместного доступа, `FILE_SHARE_READ` — позволит постороннему процессу чтение файла, а `FILE_SHARE_WRITE` — запись. Эти флаги могут комбинироваться операцией логического сложения для полного доступа.

Параметр `lpSecurityAttributes` определяет атрибуты доступа к файлу, а `dwCreationDisposition` — действия при попытке открытия несуществующего файла. Может принимать следующие значения:

- `OPEN_EXISTING` — открывает файл, ошибка, если файл не существует;
- `CREATE_NEW` — создает файл, ошибка, если файл существует;
- `CREATE_ALWAYS` — создает файл, существующий файл будет уничтожен;
- `OPEN_ ALWAYS` — открывает файл, если файл не существует, создается новый файл;
- `TRUNCATE_EXISTING` — открывает файл и усекает до нулевого размера, ошибка, если файл не существует.

Параметр `dwFlagsAndAttributes` определяет атрибуты файла для проверки или установки. Приведем наиболее распространенные значения:

- `FILE_ATTRIBUTE_NORMAL` — атрибуты не установлены;
- `FILE_ATTRIBUTE_READONLY` — файл для чтения;
- `FILE_ATTRIBUTE_HIDDEN` — скрытый файл;
- `FILE_ATTRIBUTE_ARCHIVE` — архивный файл;
- `FILE_ATTRIBUTE_SYSTEM` — системный файл;
- `FILE_ATTRIBUTE_TEMPORARY` — временный файл.

Существует еще масса дополнительных флагов для уточнения режима открытия файла, которые добавляются операцией логического сложения, но мы не будем на них останавливаться, отсылая читателя к MSDN.

Последний параметр функции `hTemplateFile` — дескриптор временного файла — для существующих файлов не используется, и мы будем задавать значение по умолчанию.

Возвращаемое значение функции `CreateFile()` — дескриптор открытого файла, а в случае неудачи — `INVALID_HANDLE_VALUE` (в отличие от других функций, где в случае неудачи возвращается `-1`).

2. Создание объекта ядра "*проекция файла*"

После открытия файла функцией `CreateFile()` необходимо указать объем физической памяти, необходимой для проекции файла. Для этого вызывается функция `CreateFileMapping()`:

```
HANDLE WINAPI CreateFileMappingW(
  HANDLE hFile,                                  //дескриптор файла
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,//атрибуты доступа
  DWORD flProtect,                               //атрибут защиты
  DWORD dwMaximumSizeHigh,                       //размер файла в виде
  DWORD dwMaximumSizeLow,                        //64-разрядного числа
  LPCWSTR lpName);                               //имя объекта
```

Если первые два параметра функции не вызывают вопросов, то следующие параметры рассмотрим подробнее.

`flProtect` — атрибут защиты страниц физической памяти:

- `PAGE_READONLY` — данные можно считывать, файл должен открываться с флагом `GENERIC_READ`;
- `PAGE_READWRITE` — полный доступ к файлу, файл должен открываться с флагом `GENERIC_READ | GENERIC_WRITE`;
- `PAGE_WRITECOPY` — данные можно считывать, но при записи в файл создаются копии страниц памяти. Открываться файл должен либо для чтения, либо с полным доступом.

Два параметра `dwMaximumSizeLow` и `dwMaximumSizeHigh` служат для записи длины файла в виде двух частей 64-разрядного числа как младшего `Low` и старшего `High` значения. Для того чтобы проекция файла отражала истинные размеры файла, нужно указать нулевые значения обеих параметров.

Последний параметр `lpName` — имя объекта ядра *"проекция файла"* — может использоваться для доступа к объекту из других процессов. Если значение параметра `NULL`, то создается локальный для данного процесса объект.

Функция `CreateFileMapping()` возвращает дескриптор *"проекции файла"*, а в случае неудачи — `NULL`.

3. Проецирование файла на адресное пространство процесса

После создания объекта "проекция файла" необходимо, чтобы операционная система зарезервировала регион адресного пространства под файл и выделила физическую память. Для этого предназначена функция `MapViewOfFile()`:

```
LPVOID WINAPI MapViewOfFile(
  HANDLE hFileMappingObject,   //дескриптор проекции файла
```

```
  DWORD dwDesiredAccess,       //доступ к данным
  DWORD dwFileOffsetHigh,      //смещение в файле для отображения
  DWORD dwFileOffsetLow,       //как 64-битное значение
  DWORD dwNumberOfBytesToMap); //размер представления
```

Параметр `dwDesiredAccess` может принимать значения:

- `FILE_MAP_READ` — файл доступен для чтения;
- `FILE_MAP_WRITE` — полный доступ для чтения и записи;
- `FILE_MAP_COPY` — данные можно читать, но при записи создаются копии страниц памяти.

32-битные параметры `dwFileOffsetHigh` и `dwFileOffsetLow` определяют старшую и младшую часть 64-битного смещения в файле. С этого байта и происходит проецирование файла.

Понятно, что для эффективного использования проекции файла необходимо, чтобы физической памяти было достаточно, однако, как правило, это не так. В этом случае мы имеем возможность проецировать файл "кадрами", а размер кадра определять, исходя из имеющихся ресурсов.

> ***Примечание***
>
> Смещение в файле должно быть кратно гранулярности выделения памяти в данной системе. В Windows она составляет 64K, но уточнить значение можно в поле `dwAllocationGranularity` структуры `SYSTEM_INFO` после обращения к функции `GetSystemInfo()`. Понятие гранулы памяти в 64K является рудиментным остатком от 16-разрядной операционной системы.

Последний параметр `dwNumberOfBytesToMap` определяет размер представления, т. е. определяет размер региона в адресном пространстве, необходимый для отображения файла. Если указать `0`, система попытается выделить память до конца файла.

Возвращаемым значением функции является указатель на начало области памяти "проекции файла" или `NULL`, в случае ошибки. Теперь, по этому указателю, с файлом можно работать, как с обычным массивом в памяти, а отображение производимых изменений на дисковый файл производится операционной системой автоматически.

Рассмотрим фрагмент программы (листинг 6.10) проецирования текстового файла в память и вывод в его окно.

Листинг 6.10. Проекция в память текстового файла

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static HANDLE hFile, hFileMap;
   static DWORD fSize;
   static char *p;
```

```
   RECT rt;
   switch (message)
   {
   case WM_CREATE:
      hFile = CreateFile(_T("readme.txt"), GENERIC_READ, 0, NULL,
               OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
      if (hFile != INVALID_HANDLE_VALUE)
      {
      fSize = GetFileSize(hFile, NULL);
      hFileMap=CreateFileMapping(hFile,NULL,PAGE_READONLY,0,fSize,NULL);
      if (hFileMap != NULL)
         if(p = (char*)MapViewOfFile(hFileMap,FILE_MAP_READ,0,0,0)) break;
      }
      DestroyWindow(hWnd);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      GetClientRect(hWnd, &rt);
      hdc = BeginPaint(hWnd, &ps);
      DrawTextA(hdc, p, fSize, &rt, DT_LEFT);
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY:
      UnmapViewOfFile(p);
      CloseHandle(hFileMap);
      CloseHandle(hFile);
      PostQuitMessage(0);
      break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

При создании окна откроем текстовый файл. Чтобы долго не искать подходящий файл небольшого размера, используем файл `readme.txt`, автоматически создаваемый при компиляции приложения в текущей папке.

```
hFile = CreateFile(_T("readme.txt"), GENERIC_READ, 0, NULL,
             OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

Зададим режим открытия файла "только для чтения" — GENERIC_READ; следующий параметр 0 — общего доступа к файлу нет; атрибуты доступа к файлу "по умолчанию" — NULL; открываем существующий файл — OPEN_EXISTING; атрибуты файла "по умолчанию" — FILE_ATTRIBUTE_NORMAL; временный файл не используется — NULL.

Теперь проверяем, открылся ли файл:

```
if (hFile != INVALID_HANDLE_VALUE)
```

Если файл открылся, то его дескриптор отличен от INVALID_HANDLE_VALUE, и можно определить его размер:

```
fSize = GetFileSize(hFile, NULL);
```

Функция возвращает младшее 32-разрядное значение размера файла, а старшие 32 бита возвращаются по адресу второго параметра, но поскольку мы уверены, что размер файла не превышает 4 Гбайт, игнорируем старшее значение, указав в качестве параметра NULL.

Теперь можно создать объект "проекция файла":

```
hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, fSize, NULL);
```

Первым параметром указываем дескриптор открытого файла hFile, атрибуты доступа используем "по умолчанию" — NULL, а атрибуты защиты памяти "только для чтения" — PAGE_READONLY, поскольку мы не собираемся ничего в эту область памяти писать. Размер файла можно было бы задать двумя нулями, тогда он определится автоматически, но поскольку мы его уже нашли, укажем в младшем параметре размер, а старший зададим нулем — 0, fSize. И, наконец, последний параметр — NULL, поскольку мы не собираемся обеспечивать доступ к файлу извне и объект создаем локально для данного процесса.

Проверим, создался ли объект:

```
if (hFileMap != NULL)
```

Если объект создан, можно выделить физическую память, совмещая эту операцию с оператором if:

```
p = (char*)MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 0);
```

Передаем в параметрах: hFileMap — дескриптор "проекции файла"; FILE_MAP_READ — вид доступа к памяти; 0,0 — смещение от начала файла, читаем файл с начала; в последнем параметре укажем 0, система сама определит размер файла, хотя могли бы указать fSize.

При неудачной работе любой из трех функций обращаемся к DestroyWindow(hWnd) для прекращения работы.

Если же все прошло удачно, выведем прочитанный файл в окно в сообщении WM_PAINT. Для простоты не будем организовывать скроллинг (этот вопрос мы подробно обсудили в *главе* 2). Сейчас мы можем рассматривать указатель p как адрес одномерного массива типа char, где помещено двоичное представление текстового файла размера fSize.

Выводить будем в клиентскую область окна функцией DrawTextA().

При завершении работы необходимо в сообщении WM_DESTROY освободить память, занятую проекцией файла:

```
UnmapViewOfFile(p);
```

а также закрыть объект "проекция файла" и исходный файл:

```
CloseHandle(hFileMap);
CloseHandle(hFile);
```

Этот простой пример показывает, что после осуществления проецирования файла в память мы можем работать с его содержимым так же, как с любым одномерным массивом.

Освобождение ресурсов

Функция MapViewOfFile() выделяет регион оперативной памяти. Когда же необходимость в этой памяти отпадает, ее можно освободить вызовом функции UnmapViewOfFile():

```
BOOL WINAPI UnmapViewOfFile(LPCVOID lpBaseAddress);
```

ее параметр lpBaseAddress — указатель на выделенный регион памяти.

Эту функцию необходимо применять, если мы вновь обращаемся к функции MapViewOfFile() для выделения нового региона памяти. В этом случае "старый регион" автоматически не уничтожается, и у нас появляются "бесхозные" области памяти, которые можно освободить лишь при закрытии приложения.

Однако поскольку система производит буферизацию страниц в файле, то для того, чтобы изменения в оперативной памяти отобразились в файле данных, нужно предварительно "сбросить" буфер в файл вызовом функции FlushViewOfFile():

```
BOOL WINAPI FlushViewOfFile(LPCVOID lpBaseAddress,
                            DWORD dwNumberOfBytesToFlush);
```

где первый параметр lpBaseAddress — адрес памяти в пределах адресного пространства проецируемого файла (он округляется до начала страницы), а второй параметр dwNumberOfBytesToFlush — размер записываемой области.

Объекты ядра "файл" и "проекция файла" удаляются вызовом функции CloseHandle(). Причем, как и все объекты ядра, они ведут счетчик пользователей, поэтому, если необходимости в их дескрипторах более не возникает, их можно удалить сразу после использования, например, так:

```
HANDLE hFile = CreateFile(. . .);
HANDLE hFileMap = CreateFileMapping(hFile, . . .);
CloseHandle(hFile);
PVOID p = MapViewOfFile(hFileMap, . . .);
CloseHandle(hFileMap);
. . . . .
```

Здесь закрытие дескриптора файла не приведет к закрытию объекта ядра "файл", поскольку после создания "проекции файла" у объекта "файл" появляется еще один пользователь. Поэтому функция CloseHandle(hFile) освободит память под описатели (дескрипторы) и уменьшит число пользователей этого объекта. То же имеет место и для объекта "проекция файла".

Совместный доступ к данным нескольких процессов

Существует несколько механизмов совместного доступа к данным несколькими процессами, которые в конечном итоге основаны на механизме проецирования файла в память. Идея использования проекции файла в память заключается в том, что мы рассматриваем файл данных как последовательность страниц файла подкачки, которые можно отобразить на адресное пространство процессов. В таком случае, работая с этими страницами в любом из процессов в их собственном адресном пространстве, мы фактически работаем с одной и той же областью физической памяти.

Технологически это осуществить даже проще, чем проецировать дисковый файл. Открывать файл здесь ни к чему, нужно сразу создать объект "проекция файла" функцией `CreateFileMapping()`, указывая вместо дескриптора файла значение `INVALID_HANDLE_VALUE`, которое служит указанием, что проецируется не дисковый файл, а свободные страницы файла подкачки. В этом случае, однако, нельзя задавать нулевой размер проецируемого файла, более того, размер резервируемой области памяти должен быть кратным размеру страницы памяти (как правило, 4K), а имя "проекции файла" является объектом ядра и доступно из другого процесса.

Для открытия существующего объекта "проекция файла" можно использовать функцию `CreateFileMapping()` или `OpenFileMapping()`:

```
HANDLE WINAPI OpenFileMappingW(DWORD dwDesiredAccess,BOOL bInheritHandle,
                               LPCWSTR lpName);
```

Здесь `dwDesiredAccess` — способ доступа к памяти, если `bInheritHandle = TRUE`, дескриптор может наследоваться, `lpName` — имя "проекции файла".

Далее, как и для файла, получаем указатель региона области памяти функцией `MapViewOfFile()`.

Для примера рассмотрим задачу, которая будет состоять из двух проектов: первый (листинг 6.11) создает страничный файл для хранения массива данных, а второй проект (листинг 6.12) прочитает этот страничный файл и выведет данные в окно.

Листинг 6.11. Запись данных в проецируемую область памяти

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   static HANDLE hFileMemory;
   static int *p;
   int i, j, *q;
   switch (message)
   {
   case WM_CREATE:
      hFileMemory = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
                    PAGE_READWRITE, 0, 4096, _T("Shared"));
      if (hFileMemory == NULL)
```

```
        {
           DestroyWindow(hWnd);
           break;
        }
      p=(int*)MapViewOfFile(hFileMemory, FILE_MAP_READ|FILE_MAP_WRITE, 0,
                            0, 0);
      for (i = 1, q = p; i <= 10; i++)
         for (j = 1; j <= 10; j++, q++) *q = i*j;
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Создаем проецируемый файл при открытии окна:

```
hFileMemory = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
                                PAGE_READWRITE, 0, 4096, _T("Shared"));
```

Вместо дескриптора файла укажем `INVALID_HANDLE_VALUE` — это означает, что файл будет создаваться на страницах файла подкачки; второй параметр `NULL` — права доступа по умолчанию; `PAGE_READWRITE` — проецируемый файл с полным доступом (чтение/запись); `0,4096` — размер файла 4K; `"Shared"` — имя объекта "проекция файла".

Если дескриптор `hFileMemory` равен `NULL`, заканчиваем работу, иначе — выделяем физическую память, возвращая указатель типа `int*`:

```
p = (int*)MapViewOfFile(hFileMemory,FILE_MAP_READ|FILE_MAP_WRITE,0,0,0);
```

Указываем первым параметром полученный дескриптор "проекции файла" `hFileMemory`, а вторым параметром — права доступа к памяти — `FILE_MAP_READ | FILE_MAP_WRITE`. Три последних параметра — смещение в файле и размер области — зададим нулевыми значениями, тогда указатель `p` установлен на начало страничного файла, а 4 Кбайт физической памяти выделено согласно размеру проецируемого файла.

В качестве блока данных у нас сейчас имеется одномерный массив с указателем `p`. Не задумываясь глубоко, заполним этот массив таблицей умножения, организовав

двойной цикл с дополнительным указателем q, поскольку указатель p изменять крайне нежелательно. Памяти нам хватит, действительно: 10 × 10 × 4 = 400 байт.

```
for (i = 1, q = p; i <= 10; i++)
  for (j = 1; j <= 10; j++, q++) *q = i*j;
```

Выводить в окно таблицу мы не планируем, поэтому удалим обработчик сообщения WM_PAINT.

Теперь рассмотрим проект приложения (листинг 6.12), которое будет читать данные из разделяемой области памяти и выводить данные в окно (рис. 6.6). Разумеется, первый проект должен быть активен.

Листинг 6.12. Чтение данных из проецируемой области памяти

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
   HDC hdc;
   static HANDLE hFileMemory;
   static int *p;
   int x, y, *q;
   TCHAR str[10];
   switch (message)
   {
   case WM_CREATE:
      hFileMemory = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
                           FALSE, _T("Shared"));
      if (hFileMemory == NULL) DestroyWindow(hWnd);
      else p = (int*)MapViewOfFile(hFileMemory, FILE_MAP_READ|
                           FILE_MAP_WRITE, 0, 0, 0);
      break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      for (y = 0, q = p; y < 300; y += 30)
         for (x = 0; x < 300; x += 30, q++)
         {
```

```
            _itot(*q, str, 10);
            TextOut(hdc, x, y, str, _tcslen(str));
          }
        EndPaint(hWnd, &ps);
        break;
    case WM_DESTROY: PostQuitMessage(0); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

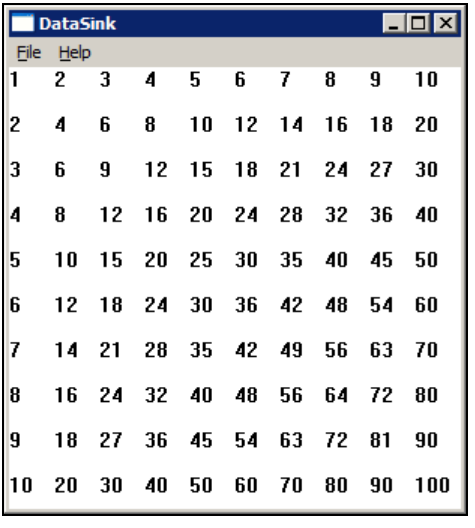


Рис. 6.6. Обмен данными между процессами

Существующий объект "проекция файла" откроем функцией `OpenFileMapping()`:

```
hFileMemory = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
                              FALSE, _T("Shared"));
```

Здесь указываем полный доступ — `FILE_MAP_READ|FILE_MAP_WRITE`, наследовать дескриптор файла нам ни к чему, второй параметр — `FALSE`; имя "проекции файла" — `"Shared"`.

Если объект ядра `"Shared"` найден, выделяем память в адресном пространстве нового процесса:

```
p = (int*)MapViewOfFile(hFileMemory,FILE_MAP_READ|FILE_MAP_WRITE,0,0,0);
```

с теми же параметрами, что и в первом проекте.

Конечно, адреса этого блока памяти будут разными в адресных пространствах двух процессов, но, тем не менее, будут указывать на одну и ту же область физической памяти. Чтобы убедиться в этом, выведем таблицу значений элементов массива в сообщении `WM_PAINT`.

Здесь организуем двойной цикл по *y*- и *x*-координате. Для простоты "шагаем" по обеим осям на 30 единиц, а для преобразования целого числа в текстовый вид используем функцию `_itot()`:

```
for (y = 0, q = p; y < 300; y += 30)
  for (x = 0; x < 300; x += 30, q++)
       {
            _itot(*q, str, 10);
            TextOut(hdc, x, y, str, _tcslen(str));
       }
```

Для демонстрации примера необходимо запустить на выполнение задачу "Запись данных в проецируемую область памяти". После чего запустим задачу "Чтение данных из проецируемый области памяти". Только в этом случае мы сможем прочитать подготовленные в проецируемом файле данные и вывести их в окне второго процесса.

Передача данных через сообщение

Некоторые оконные сообщения передают в качестве параметров указатели на строку данных, например, сообщение `WM_SETTEXT` использует `lParam` для передачи указателя строки. Хотя это и противоречит концепции разделения адресного пространства, в Windows сохраняется возможность использования таких сообщений для передачи данных окну другого процесса.

Если нам известен дескриптор окна загруженного в память процесса, мы можем послать ему сообщение функцией `SendMessage()`, например, послав сообщение:

```
SendMessage(FindWindow(NULL, _T("Калькулятор")), WM_SETTEXT, 0,
                        (LPARAM)_T("New Calculator"));
```

мы изменим заголовок окна с именем `"Калькулятор"`.

В данном примере мы нашли дескриптор окна калькулятора функцией `FindWindow()`:

```
HWND WINAPI FindWindowW(LPCWSTR lpClassName, LPCWSTR lpWindowName);
```

При помощи этой функции можно найти все окна по имени класса `ClassName` или же по заголовку `WindowName`. Обычно один из параметров равен `NULL`, как в нашем случае:

```
FindWindow(NULL, _T("Калькулятор")).
```

Возвращаемое значение функции — дескриптор активного окна или `NULL` в случае неудачи.

Предполагается, что в момент посылки сообщения в системе имеется работающее приложение, окно которого носит имя `"Калькулятор"`, тогда можно видеть, как изменится заголовок этого окна.

Если проанализировать ситуацию, то мы должны прийти к выводу, что работать это никак не может. Действительно, мы послали другому приложению адрес строки, расположенной в адресном пространстве процесса-источника. Однако читать

эту строку будет приложение-приемник в своем адресном пространстве, где полученный адрес не имеет никакого смысла.

Тем не менее, все работает. Положение спасает операционная система, которая неявно обеспечит передачу строки в адресное пространство процесса-приемника, используя механизм проецирования файла в память.

В операционной системе Windows для обмена данными предусмотрено и специальное сообщение `WM_COPYDATA`, например:

```
SendMessage(Receiver, WM_COPYDATA, (WPARAM)Sender, (LPARAM)&cds);
```

Последним параметром передается указатель на структуру `COPYDATASTRUCT`:

```
struct COPYDATASTRUCT {
  DWORD dwData;       //дополнительный параметр
  DWORD cbData;       //размер передаваемого массива в байтах
  PVOID lpData;       //указатель на массив данных
};
```

Получив сообщение `WM_COPYDATA`, приложение-приемник получает и копию структуры `COPYDATASTRUCT`, где поле `lpData` содержит указатель проекции массива данных уже в адресном пространстве приложения-приемника.

> ***Примечание***
>
> Адрес `lpData` будет актуален лишь в обработчике сообщения `WM_COPYDATA`. Если сохранить этот адрес и попытаться прочитать данные по этому адресу при обработке другого сообщения, то результат будет непредсказуем.

В качестве демонстрационной задачи использования сообщения `WM_COPYDATA` для обмена данными изменим код примера, представленного в листингах 6.11—6.12. В приложении-источнике для передачи данных создадим пункт меню **Send**, а в приложении-приемнике обработаем сообщение `WM_COPYDATA` и выведем массив данных в окно (листинги 6.13—6.14).

Листинг 6.13. Передача данных в сообщении `WM_COPYDATA`

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
   static int *p;
   int i, j, *q;
   COPYDATASTRUCT cds;
   switch (message)
   {
   case WM_CREATE:
        p = new int[100];
        for (i = 1, q = p; i <= 10; i++)
           for (j = 1; j <= 10; j++, q++) *q = i*j;
        break;
```

```
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case ID_SEND:
         cds.cbData = 100*sizeof(int);
         cds.lpData = p;
         SendMessage(FindWindow(NULL, _T("Acceptor")), WM_COPYDATA, 0,
                              (LPARAM)&cds);
         break;
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Блок данных создадим аналогично тому, как мы это делали в предыдущем примере (листинг 6.12). Передавать же данные в другое приложение с заголовком `"Acceptor"` будем, посылая сообщение при обработке пункта меню **Send**.

Здесь мы должны определить два поля структуры `COPYDATASTRUCT`:

```
cds.cbData = 100*sizeof(int);
cds.lpData = p;
```

где указывается размер массива данных в байтах и адрес начала массива.

После чего можно передать сообщение `WM_COPYDATA` окну с именем `Acceptor`, дескриптор которого мы найдем обращением к функции `FindWindow()`:

```
SendMessage(FindWindow(NULL,_T("Acceptor")),WM_COPYDATA,0,(LPARAM)&cds);
```

> ***Примечание***
>
> Сообщение `WM_COPYDATA` должно передаваться только как синхронное, поскольку объект "проекция файла", создаваемый системой для обмена данными, автоматически разрушается по завершении обработки этого сообщения. Поэтому использование в этом случае функции `PostMessage()` приведет к непредсказуемым последствиям.

Теперь в листинге 6.14 рассмотрим, как процесс, созданный в проекте **Acceptor**, примет переданные данные. Разумеется, на момент посылки сообщения второй процесс должен быть загружен в память.

Листинг 6.14. Прием данных посредством сообщения `WM_COPYDATA`

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   PAINTSTRUCT ps;
```

```
   HDC hdc;
   static int *p;
   int x, y, *q;
   TCHAR str[10];
   switch (message)
   {
   case WM_COPYDATA:
         p = new int[((COPYDATASTRUCT *)lParam)->cbData/sizeof(int)];
         q = (int*)(((COPYDATASTRUCT *)lParam)->lpData);
         for(int i = 0; i < 100; i++) p[i] = *q++;
         InvalidateRect(hWnd, NULL, TRUE);
         break;
   case WM_COMMAND:
      switch (LOWORD(wParam))
      {
      case IDM_EXIT: DestroyWindow(hWnd); break;
      default: return DefWindowProc(hWnd, message, wParam, lParam);
      }
      break;
   case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      if (p)
      {
         for (y = 0, q = p; y < 300; y += 30)
            for (x = 0; x < 300; x += 30, q++)
            {
               _itot(*q, str, 10);
               TextOut(hdc, x, y, str, _tcslen(str));
            }
      }
      EndPaint(hWnd, &ps);
      break;
   case WM_DESTROY: PostQuitMessage(0); break;
   default: return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

Оконная функция должна обрабатывать сообщение `WM_COPYDATA`. При этом `lParam` должен интерпретироваться как указатель на структуру `COPYDATASTRUCT`. Однако при извлечении размера массива для указателя приходится применять явные преобразования типа — `(COPYDATASTRUCT *)lParam`.

Поскольку по завершении обработки сообщения WM_COPYDATA проекция массива данных на адресное пространство второго процесса будет уничтожена, необходимо скопировать данные в адресное пространство текущего процесса. Для этого выделим необходимую область памяти:

```
p = new int[((COPYDATASTRUCT *)lParam)->cbData/sizeof(int)];
```

Получим адрес начала массива, так же используя явное преобразование типа:

```
q = (int*)(((COPYDATASTRUCT *)lParam)->lpData);
```

и скопируем массив данных:

```
for(i = 0; i< 100; i++) p[i] = *q++;
```

После этого можно перерисовать окно, объявив его недействительным:

```
InvalidateRect(hWnd, NULL, TRUE);
```

Сообщение WM_PAINT мы обрабатываем практически так же, как в листинге 6.13. Единственное отличие заключается в том, что мы цикл вывода данных заключили в оператор:

```
if (p) {. . .}
```

Это сделано, чтобы отображалось "пустое" окно, пока данные еще не получены. Действительно, в этом случае указатель p == NULL, поскольку определен как static, и выражение в скобках не выполняется. Когда же данные получены, указатель p отличен от нуля и таблица с данными будет выведена в окно.

Итак, мы рассмотрели два способа обмена данными между процессами. Какой из них выбрать в реальной задаче? Все зависит от поставленной задачи и от пристрастий программиста.

В *Приложении* для поиска окна другого процесса рассмотрена еще одна методика, которая может использоваться, когда нам известен идентификатор процесса.

Вопросы к главе

1. Какими параметрами характеризуется процесс и поток?
2. Функции, используемые для создания процесса и потока.
3. В чем принципиальное различие процесса и потока?
4. Как узнать код завершения процесса?
5. Функции C++ для создания и завершения потока.
6. Как измерить время работы потока, процесса?
7. Как измерить малый (менее одного кванта) интервал времени?
8. Классы приоритетов процессов и относительные приоритеты потоков.
9. Разграничение доступа к памяти при помощи функций атомарного доступа.
10. Синхронизация потоков критическими секциями.
11. Принцип работы Wait-функций.
12. Синхронизация потоков и процессов при помощи объектов ядра.

13. Чем отличается мьютекс от исключающего семафора?
14. Как создать ожидаемый таймер на заданный момент времени?
15. Создание общей области памяти для всех экземпляров процесса.
16. Три этапа создания файла, проецируемого в память.
17. Как освободить физическую память, занятую проекцией файла?
18. Как организовать совместный доступ к памяти нескольким процессам?
19. Какую минимальную и максимальную область памяти можно выделить для совместного использования двум процессам?
20. Какие сообщения можно использовать для обмена данными между процессами?

Задания для самостоятельной работы

1. В задаче "Создание процесса" (листинг 6.1) создать процесс в "свернутом виде". Почему не удается "свернуть" при запуске стандартный калькулятор?
2. "Измерить" время работы созданного в предыдущей задаче процесса. Вывести результат в главном окне "родительского" процесса.
3. Увеличить приоритет главного потока процесса, показать текущее значение приоритета в заголовке окна.
4. Написать программу, которая построчно читает текстовый файл с диска и выводит в окно. При чтении файла подсчитывать количество печатных знаков (за вычетом пробелов), а также количество слов. Операции выполнить в виде двух отдельных потоков. Последовательность действий синхронизовать при помощи одного из рассмотренных механизмов.
5. Напишите программу, в которой, с периодичностью в 1 секунду, один поток выделяет блок памяти для набора из 100 целых чисел, второй поток заполняет его случайными числами, третий поток выводит этот набор в окно таблицей 10×10. Синхронизовать потоки при помощи критических секций.
6. Создать два процесса. В первом процессе обеспечить ввод целочисленного значения, во втором процессе получить это число и вывести в окне.
7. Написать программу, которая каждый день в 09-00 "будит" компьютер и выводит заставку с приветствием, а 13 числа в пятницу предупреждает, что нужно проверить на вирусы.
8. Написать программу, которая запускается при включении компьютера (предполагается, что она включена в папку Автозагрузка) и через каждые 3 часа предлагает сделать небольшой перерыв.
9. Написать программу, которая, проецируя текстовый файл в память, собирает статистику: количество символов, слов, строк. Если размер файла превышает размер гранулярности, проецирование осуществлять стандартными гранулами.
10. Создать два приложения, которые обмениваются текстовыми строками, имитируя переписку.

Приложение

Поиск окна

Материал этого приложения посвящен вопросу управления окном другого приложения, что вряд ли приветствуется разработчиками операционной системы Windows. Действительно, каждый процесс в системе изолирован и, казалось бы, не должен иметь возможность взаимодействовать с другими процессами в системе, однако создатели библиотеки Win32 API включили в нее несколько наборов функций, которые обходят эти ограничения.

Все что нам нужно для управления окном, это его дескриптор. Существует несколько возможностей для отыскания дескриптора окна. Самый простой из них заключается в использовании функции `FindWindow()`, которая найдет дескриптор окна либо по имени класса окна, либо по его заголовку. Здесь же мы рассмотрим возможность поиска окна, перебирая все окна в их Z-порядке, т. е. том порядке, в котором они отображаются на мониторе.

Поиск всех окон, зарегистрированных в системе

Для начала рассмотрим тестовую задачу (листинг П.1), которая позволит нам просмотреть информацию обо всех зарегистрированных в системе окнах.

Листинг П.1. Поиск всех окон системы

```
#include <string>
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
  static std::string st("hProcess   hParent      idProcess   hWindows\
Title\r\n");
  HWND hFind;
  static HWND hEdit;
  TCHAR str[80], title[256];
  LONG hProcess, hParentEdit;
```

```
  DWORD id;
  switch (message)
  {
  case WM_CREATE:
    hEdit = CreateWindow("Edit", NULL, WS_CHILD|WS_VISIBLE|WS_HSCROLL|
      WS_VSCROLL|ES_LEFT|ES_MULTILINE|ES_AUTOHSCROLL|ES_AUTOVSCROLL,
      0, 0, 0, 0, hWnd, (HMENU) 1, hInst, NULL);
    break;
  case WM_SIZE:
    MoveWindow(hEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
    break;
  case WM_COMMAND:
    switch (LOWORD(wParam))
    {
    case ID_FIND:
      hFind = GetTopWindow(NULL);
      do
      {
        hProcess = GetWindowLong(hFind,GWL_HINSTANCE);
        hParentEdit = GetWindowLong(hFind,GWL_HWNDPARENT);
        SendMessage(hFind, WM_GETTEXT, (WPARAM)256, (LPARAM)title);
        GetWindowThreadProcessId(hFind, &id);
        sprintf(str, "%.8x   %.8x   %.8d   %.8x      %s\r\n", hProcess,
             hParentEdit, id, hFind, title);
        st.append(str);
        hFind = GetWindow(hFind, GW_HWNDNEXT);
      }
      while (hFind);
      SetWindowText(hEdit, st.c_str());
      break;
    case IDM_EXIT: DestroyWindow(hWnd); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
  case WM_DESTROY: PostQuitMessage(0); break;
  default: return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

Для сбора текстовой информации мы воспользуемся классом `string`, а вывод будем осуществлять в элемент управления `Edit Control`, который создадим в сообщении `WM_CREATE`, и наложим поверх клиентской области главного окна.

Создадим пункт меню с идентификатором `ID_FIND`, где при помощи функции `GetTopWindow()` найдем дескриптор окна, имеющего максимальную *z*-координату (самое "верхнее" окно). Теперь находим при помощи функции `GetWindowLong()` дескриптор приложения `hProcess` и родительского окна `hParentEdit`.

> ***Примечание***
>
> Для главного окна приложения родителем, как правило, является "рабочий стол" с дескриптором, равным 0.

Заголовок окна получаем, посылая ему сообщение `WM_GETTEXT`.

Идентификатор процесса, породившего данное окно, получим при помощи функции `GetWindowThreadProcessId()`.

После того как будет сформирована строка с информацией о данном окне, переходим к следующему в *Z*-порядке окну функцией `GetWindow()`. Цикл будет работать до самого "нижнего" по *z*-координате окна, после чего функция `GetWindow()` вернет 0 и цикл завершится.

Нам осталось поместить текстовую строку в окно редактирования, результат мы увидим в окне (рис. П.1).

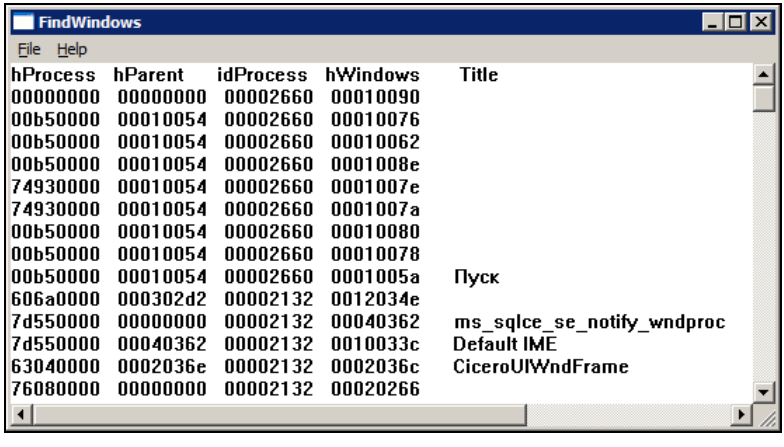


Рис. П.1. Информация о найденных окнах

Поиск главного окна созданного процесса

Рассмотрим теперь такую ситуацию — мы создали новый процесс и теперь хотели бы иметь возможность управлять главным окном процесса из процесса-родителя. Хотя при создании процесса мы получили его дескриптор, пользы от него не очень много. Можно с его помощью завершить работу процесса, но найти окно нам не удастся, к тому же этот дескриптор имеет смысл только в контексте родительского процесса, поскольку каждый процесс имеет собственный набор дескрипторов.

Гораздо более информативен оказывается идентификатор процесса, который имеет одинаковое значение для всех процессов.

Искать окно дочернего процесса мы будем по методике, рассмотренной в предыдущей задаче — ищем окно, идентификатор процесса которого совпадает с требуемым, а признак главного окна — нулевой дескриптор родительского окна.

Решим следующую тестовую задачу — откроем в качестве отдельного процесса блокнот и поменяем у него заголовок окна (листинг П.2).

Листинг П.2. Поиск главного окна дочернего процесса

```
TCHAR title[256] = _T("Новое окно блокнота");
TCHAR CommandLine[256] = _T("notepad ReadMe.txt");
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
   static STARTUPINFO tin;
   static PROCESS_INFORMATION pInfo;
   HWND hFind;
   DWORD id, hParentEdit;
   switch (message)
   {
   case WM_CREATE:
     tin.cb = sizeof(STARTUPINFO);
     tin.dwFlags = STARTF_USESHOWWINDOW;
     tin.wShowWindow = SW_SHOWNORMAL;
     CreateProcess(NULL, CommandLine, NULL, NULL, FALSE, 0, NULL, NULL,
               &tin, &pInfo);
     CloseHandle(pInfo.hThread);
     CloseHandle(pInfo.hProcess);
     break;
   case WM_COMMAND:
     switch (LOWORD(wParam))
     {
     case ID_FINDWINDOWS:
        hFind = GetTopWindow(NULL);
        do
        {
          if (hFind == 0)
          {
            MessageBox(hWnd, _T("Окно не найдено"),
                       _T("Ошибка поиска"), MB_OK | MB_ICONQUESTION);
            DestroyWindow(hWnd);
```

```
            return 1;
          }
          GetWindowThreadProcessId(hFind, &id);
          hParentEdit = GetWindowLong(hFind, GWL_HWNDPARENT);
          if (hParentEdit == 0 && id == pInfo.dwProcessId)
          {
            SendMessage(hFind, WM_SETTEXT, 0, (LPARAM)title);
            return 0;
          }
          hFind = GetWindow(hFind, GW_HWNDNEXT);
        }
        while (true);
        break;
    case IDM_EXIT: DestroyWindow(hWnd); break;
    default: return DefWindowProc(hWnd, message, wParam, lParam);
    }
    break;
  case WM_DESTROY: PostQuitMessage(0); break;
  default: return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

При создании окна приложения создадим отдельный процесс и откроем в нем блокнот. Для наглядности загрузим в него файл `ReadMe.txt`. При обработке пункта меню с идентификатором `ID_FINDWINDOWS` организуем перебор окон, начиная с "верхнего" в *Z*-порядке, и ищем окно, идентификатор процесса для которого совпадает с `pInfo.dwProcessId`, а родительское окно имеет нулевой дескриптор. Если такое окно найдено, посылаем ему сообщение `WM_SETTEXT` для смены заголовка, если же окно не было найдено и `hFind == 0`, выводим сообщение об ошибке и прекращаем работу.

Литература

1. Педзольд Ч. Программирование для Windows 95. В 2 т. Т. 1 — СПб.: BHV, 1996.
2. Шилд Г. Программирование на C и C++ для Windows 95. — Киев: Торгово-издательское бюро BHV, 1996.
3. Мюррей У., Паппас К. Создание переносимых приложений для Windows. — СПб.: BHV, 1997.
4. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows: Пер. с англ. — 4-е изд. — СПб.: Питер; М.: Издательско-торговый дом "Русская редакция", 2003.
5. Шупак Ю. А. Win32 API. Разработка приложений для Windows. — СПб.: Питер, 2008.
6. Верма Р. Д. Справочник по функциям Win32 API. — 2-е изд., перераб. и доп. — М.: Горячая линия — Телеком, 2005.

Дополнительная литература

1. Давыдов В. Г. Visual C++. Разработка Windows-приложений с помощью MFC и API-функций. — СПб.: БХВ-Петербург, 2008.
2. Саймон Р. Microsoft Windows 2000 API. Энциклопедия программиста. — Киев: Диасофт, 2001.
3. Вильямс М. Программирование в Windows 2000. Энциклопедия пользователя. — Киев: Диасофт, 2000.
4. Юань Фень. Программирование графики для Windows. — СПб.: Питер, 2002.
5. Румянцев П. В. Азбука программирования в Win32 API. — М.: Горячая линия — Телеком, 2001.

Предметный указатель

(окончание рубрики см. на стр. 280)