Модуль №1

Занятие №4

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. Работа с таймером.
3. Перечисление окон.
4. Ресурсы приложения.
   4.1. Пиктограмма.
   4.2. Курсор.
5. Обработка ошибок.
6. Практическая часть.
7. Подведение итогов.
8. Домашнее задание.

1. Повторение пройденного материала

Данное занятие необходимо начать с краткого повторения материала предыдущего занятия. При общении со слушателями можно использовать следующие контрольные вопросы:

1) Какие бывают сообщения мыши? Какая дополнительная информация приходит с этими сообщениями?
2) Какие существуют клавиатурные сообщения? Какая дополнительная информация приходит с этими сообщениями?
3) В каком случае следует обрабатывать сообщение **WM_CHAR**, а в каком случае – **WM_KEYDOWN** и **WM_KEYUP**?
4) Какой функцией можно определить состояние указанной виртуальной клавиши?
5) Какой функцией можно вывести текст в заголовок окна?

6) Чем отличается функция **GetWindowRect** от функции **GetClientRect**?
7) Как преобразовать экранные координаты указанной точки в клиентские координаты относительно левого верхнего угла рабочей области заданного окна?
8) Как выполнить обратное преобразование?
9) Каким образом можно переместить окно, а также изменить его размеры?
10) Какая функция позволяет получить дескриптор окна верхнего уровня? Какую информацию необходимо передать в эту функцию?
11) Какая функция позволяет получить дескриптор дочернего окна? Какую информацию необходимо передать в эту функцию?
12) С помощью какого средства можно определить заголовок и класс окна?

2. Работа с таймером

Ознакомить слушателей с возможностью установки таймера в Windows-приложениях. Отметить, что использование таймера является хорошим способом время от времени «будить» программу. Это может быть полезным в том случае, если программа выполняется как фоновое приложение. Для установки таймера необходимо использовать функцию API **SetTimer**:

```
UINT SetTimer(
HWND hwnd, // дескриптор окна, которое собирается использовать таймер
UINT nID, // идентификатор устанавливаемого таймера
UINT wLength, // временной интервал для таймера в миллисекундах
TIMEPROC lpTFunc // указатель на функцию - обработчик прерываний таймера
);
```

Следует подчеркнуть, что функция, указатель на которую задается параметром **lpTFunc**, является процедурой, определенной в программе и вызываемой при обработке прерываний таймера. Эта функция должна быть определена как **VOID CALLBACK** и иметь такие же параметры, как и оконная функция окна. Однако, ес-

ли значение **lpTFunc** равно NULL, как это чаще всего и бывает, для обработки сообщений таймера будет вызываться оконная процедура главного окна приложения. В этом случае каждый раз по истечении заданного временного интервала в очередь сообщений программы будет помещаться сообщение **WM_TIMER,** а оконная процедура программы должна будет обрабатывать его так же, как и остальные сообщения. Функция **SetTimer** в случае успешного завершения возвращает значение идентификатора таймера, в противном случае возвращается 0.

Будучи установленным, таймер будет посылать сообщения до тех пор, пока программа не завершится или не вызовет функцию API **KillTimer:**

```
BOOL KillTimer(
HWND hwnd, // дескриптор окна, использующего таймер
UINT nID // идентификатор таймера
);
```

Рассмотреть со слушателями оба варианта обработки прерываний таймера:

1) обработку сообщения **WM_TIMER**;
2) обработку прерываний таймера с использованием специальной **CALLBACK**-функции.

В качестве примера обработки прерываний таймера с использованием сообщения **WM_TIMER** привести следующий код, который выводит текущие дату и время в заголовок окна. Информация обновляется с интервалом в 1 секунду:

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

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                     LPTSTR lpszCmdLine, int nCmdShow)
{
      HWND hWnd;
      MSG Msg;
      WNDCLASSEX wcl;
      wcl.cbSize = sizeof(wcl);
      wcl.style = CS HREDRAW | CS VREDRAW;
      wcl.lpfnWndProc = WindowProc;
      wcl.cbClsExtra = 0;
      wcl.cbWndExtra = 0;
      wcl.hInstance = hInst;
```

```
	wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
	wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
	wcl.lpszMenuName = NULL;
	wcl.lpszClassName = szClassWindow;
	wcl.hIconSm = NULL;

	if (!RegisterClassEx(&wcl))
		return 0;

	hWnd = CreateWindowEx(0, szClassWindow, TEXT("Работа с таймером"),
	WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
	CW_USEDEFAULT, NULL, NULL, hInst, NULL);

	ShowWindow(hWnd, nCmdShow);
	UpdateWindow(hWnd);

	while(GetMessage(&Msg, NULL, 0, 0))
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
	return Msg.wParam;
}

LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{
	static time_t t;
	static TCHAR str[100];
	switch(message)
	{
		case WM_DESTROY:
			PostQuitMessage(0);
			break;

		case WM_TIMER:
			// количество секунд, прошедших с 01.01.1970
			t = time(NULL);
			// формирование строки следующего формата:
			// день месяц число часы:минуты:секунды год
			lstrcpy(str, _tctime(&t));
			str[lstrlen(str) - 1] = '\0';
			// вывод даты и времени в заголовок окна
			SetWindowText(hWnd, str);
			break;
		case WM_KEYDOWN:
			// установка таймера по нажатию клавиши <ENTER>
			if(wParam == VK_RETURN)
				SetTimer(hWnd, 1, 1000, NULL);
			// уничтожение таймера по нажатию клавиши <ESC>
			else if(wParam == VK_ESCAPE)
				KillTimer(hWnd, 1);
			break;
		default:
			return DefWindowProc(hWnd, message, wParam, lParam);
	}
	return 0;
}
```

Следует отметить, что при поступлении сообщения **WM_TIMER** параметр **wParam** содержит идентификатор таймера (приложение может установить несколько таймеров), а **lParam** - адрес функции таймера (если он был задан при установке таймера).

В качестве примера обработки прерываний таймера с использованием функции обратного вызова привести следующий код:

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

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

TCHAR szClassWindow[] = TEXT("Каркасное приложение");


int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                  LPTSTR lpszCmdLine, int nCmdShow)
{
      HWND hWnd;
      MSG Msg;
      WNDCLASSEX wcl;
      wcl.cbSize = sizeof(wcl);
      wcl.style = CS_HREDRAW | CS_VREDRAW;
      wcl.lpfnWndProc = WindowProc;
      wcl.cbClsExtra = 0;
      wcl.cbWndExtra = 0;
      wcl.hInstance = hInst;
      wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
      wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      wcl.lpszMenuName = NULL;
      wcl.lpszClassName = szClassWindow;
      wcl.hIconSm = NULL;

      if (!RegisterClassEx(&wcl))
            return 0;

      hWnd = CreateWindowEx(0, szClassWindow, TEXT("Работа с таймером"),
      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, NULL, NULL, hInst, NULL);

      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);

      while(GetMessage(&Msg, NULL, 0, 0))
      {
            TranslateMessage(&Msg);
            DispatchMessage(&Msg);
      }
      return Msg.wParam;
}
```

```
VOID CALLBACK TimerProc(
      HWND hwnd, // дескриптор окна, которое собирается использовать таймер
      UINT uMsg, // идентификатор сообщения WM_TIMER
      UINT_PTR idEvent, // идентификатор устанавливаемого таймера
      DWORD dwTime  // временной интервал для таймера в миллисекундах
      )
{
      static time_t t;
      static TCHAR str[100];
      t = time(NULL);   // количество секунд, прошедших с 01.01.1970
      // формирование строки следующего формата:
      // день месяц число часы:минуты:секунды год
      lstrcpy(str, _tctime(&t));
      str[lstrlen(str) - 1] = '\0';
      SetWindowText(hwnd, str); // вывод даты и времени в заголовок окна
}

LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{
      switch(message)
      {
            case WM_DESTROY:
                  PostQuitMessage(0);
                  break;

            case WM_KEYDOWN:
                  // установка таймера по нажатию клавиши <ENTER>
                  if(wParam == VK_RETURN)
                        SetTimer(hWnd, 1, 1000, TimerProc);
                  // уничтожение таймера по нажатию клавиши <ESC>
                  else if(wParam == VK_ESCAPE)
                        KillTimer(hWnd, 1);
                  break;

            default:
                  return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

3. Перечисление окон

При рассмотрении данного вопроса напомнить слушателям о возможности поиска окон, как верхнего уровня, так и дочерних, на основании класса окна и заголовка окна. Подчеркнуть, что такую возможность предоставляют ранее рассмотренные функции **FindWindow** и **FindWindowEx**, возвращающие дескриптор окна верхнего уровня (т.е. окна, не имеющего «родителя») и дескриптор дочернего окна соответственно. Отметить, что существуют функции, которые расширяют воз-

можности вышеуказанных, и позволяют получить дескрипторы всех окон верхнего уровня, а также дескрипторы дочерних окон для указанного top-level окна.

Функция API **EnumWindows** предназначена для перечисления окон верхнего уровня:

```
BOOL EnumWindows(
      WNDENUMPROC lpEnumFunc, // указатель на функцию обратного вызова
      LPARAM lParam // аргумент, передаваемый в функцию обратного вызова
      );
```

Функция **EnumWindows** работает совместно с CALLBACK-функцией, указанной в первом параметре, и передаёт ей дескриптор каждого перечисленного окна верхнего уровня. **EnumWindows** продолжает свою работу до тех пор, пока не будут перечислены все окна верхнего уровня или пока CALLBACK-функция не вернёт нуль. Таким образом, с помощью функции **EnumWindows** можно определить, какие приложения, обладающие окном, выполняются в данное время.

Прототип функции обратного вызова имеет следующий вид:

```
BOOL CALLBACK EnumWindowsProc(
      HWND hwnd, // дескриптор очередного перечисленного окна верхнего уровня
      LPARAM lParam  // аргумент, переданный в CALLBACK-функцию
      );
```

Следует отметить, что для продолжения перечисления окон, CALLBACK–функция должна возвращать **TRUE**, в противном случае перечисление прекратится.

В качестве примера, демонстрирующего работу функции API **EnumWindows,** привести следующий код, в котором по нажатию клавиши <**CTRL**> начинается перечисление окон верхнего уровня:

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

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                     LPTSTR lpszCmdLine, int nCmdShow)
{
```

```
	HWND hWnd;
	MSG Msg;
	WNDCLASSEX wcl;
	cl.cbSize = sizeof(wcl);
	wcl.style = CS_HREDRAW | CS_VREDRAW;
	wcl.lpfnWndProc = WindowProc;
	wcl.cbClsExtra = 0;
	wcl.cbWndExtra = 0;
	wcl.hInstance = hInst;
	wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
	wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
	wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
	wcl.lpszMenuName = NULL;
	wcl.lpszClassName = szClassWindow;
	wcl.hIconSm = NULL;

	if (!RegisterClassEx(&wcl))
		return 0;

	hWnd = CreateWindowEx(0, szClassWindow,
		TEXT("Перечисление окон верхнего уровня"), WS_OVERLAPPEDWINDOW,
		CW USEDEFAULT, CW USEDEFAULT, CW USEDEFAULT, CW USEDEFAULT, NULL,
		NULL, hInst, NULL);

	ShowWindow(hWnd, nCmdShow);
	UpdateWindow(hWnd);


	while(GetMessage(&Msg, NULL, 0, 0))
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
	return Msg.wParam;
}


BOOL CALLBACK EnumWindowsProc(HWND hWnd, LPARAM lParam)
{
	HWND hWindow = (HWND) lParam; // дескриптор окна нашего приложения
	TCHAR caption[MAX_PATH] = {0}, classname[100] = {0}, text[500] = {0};
	// получаем текст заголовка текущего окна верхнего уровня
	GetWindowText(hWnd, caption, 100);
	// получаем имя класса текущего окна верхнего уровня
	GetClassName(hWnd, classname, 100);
	if(lstrlen(caption))
	{
		lstrcat(text, TEXT("Заголовок окна: "));
		lstrcat(text, caption);
		lstrcat(text, TEXT("\n"));
		lstrcat(text, TEXT("Класс окна: "));
		lstrcat(text, classname);
		MessageBox(hWindow, text,
		TEXT("Перечисление окон верхнего уровня"),
		MB_OK | MB_ICONINFORMATION);
	}
	return TRUE; // продолжаем перечисление окон верхнего уровня
}
```

```
LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{
      switch(message)
      {
            case WM_DESTROY:
                  PostQuitMessage(0);
                  break;
            case WM_KEYDOWN:
                  if(wParam == VK_CONTROL)
                        // начинаем перечисление окон верхнего уровня
                        EnumWindows(EnumWindowsProc, (LPARAM) hWnd);
                  break;
            default:
                  return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

Функция API **EnumChildWindows** предназначена для перечисления дочерних окон указанного top-level окна:

```
BOOL EnumChildWindows(
      HWND hWndParent, // дескриптор родительского окна
      WNDENUMPROC lpEnumFunc, // указатель на функцию обратного вызова
      LPARAM lParam // аргумент, передаваемый в функцию обратного вызова
      );
```

Функция **EnumChildWindows** работает совместно с CALLBACK-функцией, указанной во втором параметре, и передаёт ей дескриптор каждого перечисленного дочернего окна. **EnumChildWindows** продолжает свою работу до тех пор, пока не будут перечислены все дочерние окна или пока CALLBACK-функция не вернёт нуль. Таким образом, функция **EnumChildWindows** может применяться для последовательной обработки дочерних окон, если заранее неизвестно их количество.

Прототип функции обратного вызова имеет следующий вид:

```
BOOL CALLBACK EnumChildProc(
      HWND hwnd, // дескриптор очередного перечисленного окна
      LPARAM lParam  // аргумент, переданный в CALLBACK-функцию
      );
```

Следует отметить, что для продолжения перечисления окон, CALLBACK–функция должна возвращать **TRUE**, в противном случае перечисление прекратится.

В качестве примера, демонстрирующего работу функции API **EnumChildWindows,** привести следующий код, в котором по нажатию клавиши <**CTRL**> начинается перечисление дочерних окон главного окна приложения «Калькулятор».

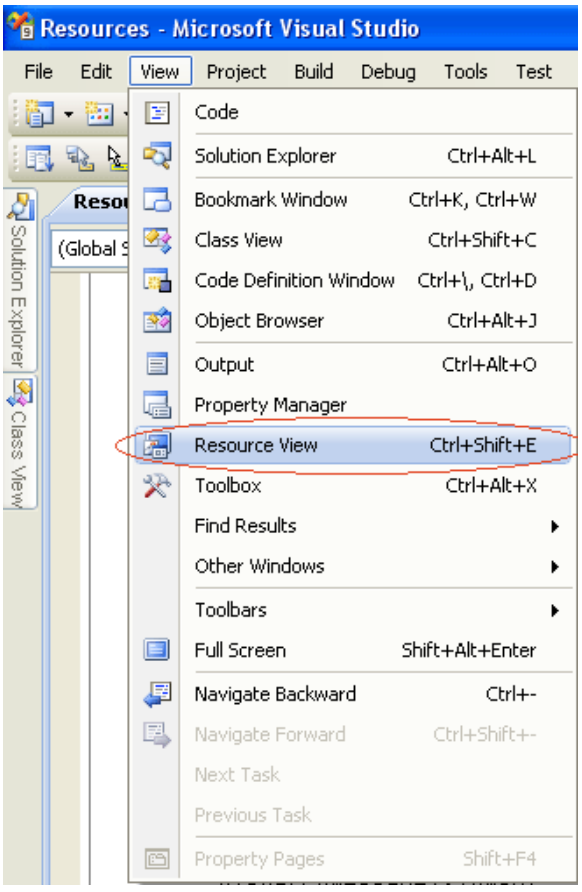
```
#include <windows.h>
#include <tchar.h>

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

TCHAR szClassWindow[] = TEXT("Каркасное приложение");

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                  LPTSTR lpszCmdLine, int nCmdShow)
{
      HWND hWnd;
      MSG Msg;

      WNDCLASSEX wcl;
      wcl.cbSize = sizeof(wcl);
      wcl.style = CS_HREDRAW | CS_VREDRAW;
      wcl.lpfnWndProc = WindowProc;
      wcl.cbClsExtra = 0;
      wcl.cbWndExtra = 0;
      wcl.hInstance = hInst;
      wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
      wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      wcl.lpszMenuName = NULL;
      wcl.lpszClassName = szClassWindow;
      wcl.hIconSm = NULL;

      if (!RegisterClassEx(&wcl))
            return 0;

      hWnd = CreateWindowEx(0, szClassWindow,
            TEXT("Перечисление дочерних окон"), WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
            NULL, NULL, hInst, NULL);

      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);

      MessageBox(hWnd,
      TEXT("Откройте, пожалуйста, \"Калькулятор\", и нажмите <CTRL>"),
      TEXT("Перечисление дочерних окон"),
      MB_OK | MB_ICONINFORMATION);

      while(GetMessage(&Msg, NULL, 0, 0))
      {
            TranslateMessage(&Msg);
```

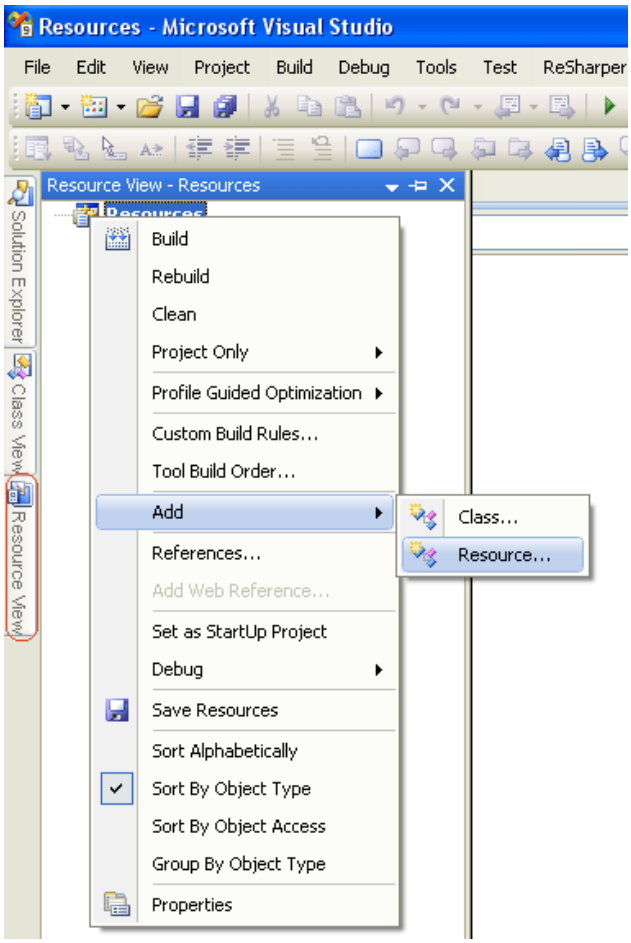
```
            DispatchMessage(&Msg);
      }
      return Msg.wParam;
}


BOOL CALLBACK EnumChildProc(HWND hWnd, LPARAM lParam)
{
      HWND hWindow = (HWND) lParam; // дескриптор окна нашего приложения
      TCHAR caption[MAX_PATH] = {0}, classname[100] = {0}, text[500] = {0};
      // получаем текст заголовка текущего дочернего окна
      GetWindowText(hWnd, caption, 100);
      // получаем имя класса текущего дочернего окна
      GetClassName(hWnd, classname, 100);
      if(lstrlen(caption))
      {
            lstrcat(text, TEXT("Заголовок окна: "));
            lstrcat(text, caption);
            lstrcat(text, TEXT("\n"));
      }
      lstrcat(text, TEXT("Класс окна: "));
      lstrcat(text, classname);
      MessageBox(hWindow, text, TEXT("Перечисление дочерних окон"),
                  MB_OK | MB_ICONINFORMATION);
      return TRUE; // продолжаем перечисление дочерних окон
}


LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{
      switch(message)
      {
            case WM_DESTROY:
                  PostQuitMessage(0);
                  break;
            case WM_KEYDOWN:
                  if(wParam == VK_CONTROL)
                  {
                  // получим дескриптор "Калькулятора"
                  HWND h = FindWindow(TEXT("SciCalc"), TEXT("Калькулятор"));
                  if(!h)
                        MessageBox(hWnd,
                        TEXT("Необходимо открыть \"Калькулятор\""),
                        TEXT("Ошибка!!!"), MB_OK | MB_ICONSTOP);
                  else
                  // начинаем перечисление дочерних окон "Калькулятора"
                  EnumChildWindows(h, EnumChildProc, (LPARAM) hWnd);
                  }
                  break;
            default:
                  return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

4. Ресурсы приложения

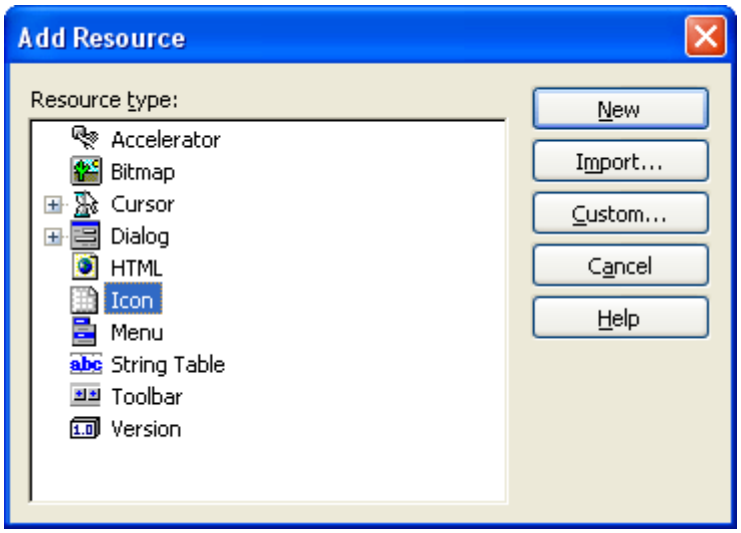
В рамках данного занятия следует начать ознакомление слушателей с ресурсами приложения. Отметить, что ресурсы являются составной частью приложений для Windows. В них определяются такие объекты, как пиктограммы (иконки), курсоры, растровые образы, таблицы строк, меню, диалоговые окна и многие другие. Для некоторых видов ресурсов система содержит предопределенные объекты. Например, стандартная иконка (**IDI_APPLICATION**) или стандартный курсор (**IDC_ARROW**).

Все нестандартные ресурсы должны быть определены в файле описания ресурсов (resource script), который является ASCII-файлом с расширением **.rc**. Обратить внимание слушателей, что подобный файл можно подготовить в обычном текстовом редакторе. Однако такая технология использовалась в прошлом, поскольку Microsoft Visual Studio содержит удобные редакторы ресурсов, максимально упрощающие и автоматизирующие этот процесс.

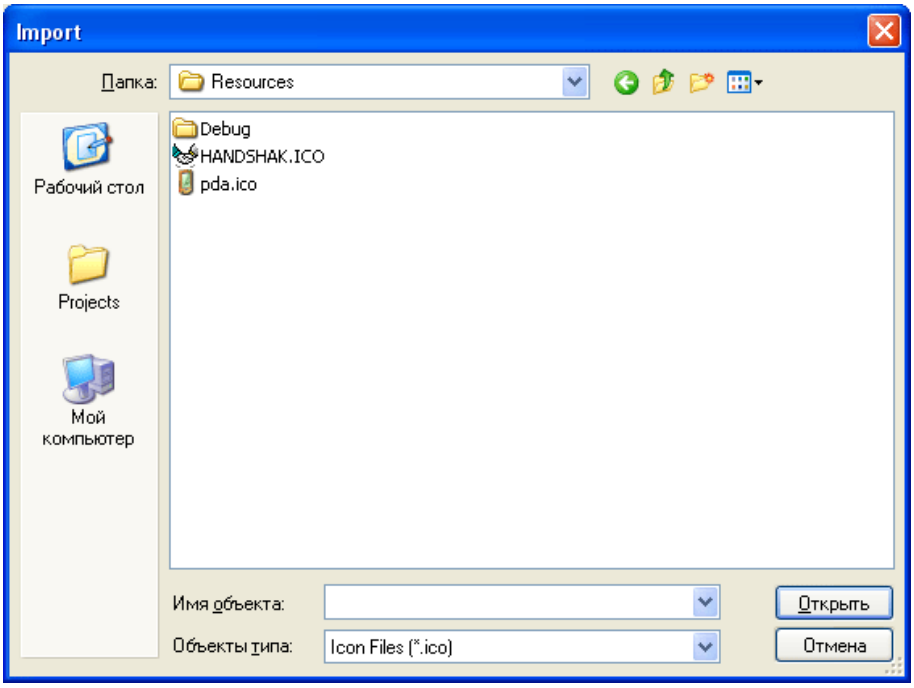




Необходимо отметить, что изначально в новом проекте ресурсы отсутствуют. Для добавления ресурса необходимо активизировать вкладку **Resource View**, в которой с помощью контекстного меню вызвать диалог добавления ресурса **Add -> Resource...**



В появившемся диалоговом окне необходимо выбрать один из стандартных типов ресурса, либо с помощью кнопки **Custom...** добавить нестандартный тип ресурса. При нажатии кнопки **New** вызывается редактор ресурса, в котором можно создать новый ресурс указанного типа. При нажатии кнопки **Import** … вызывается диалоговое окно для выбора существующего ресурса.

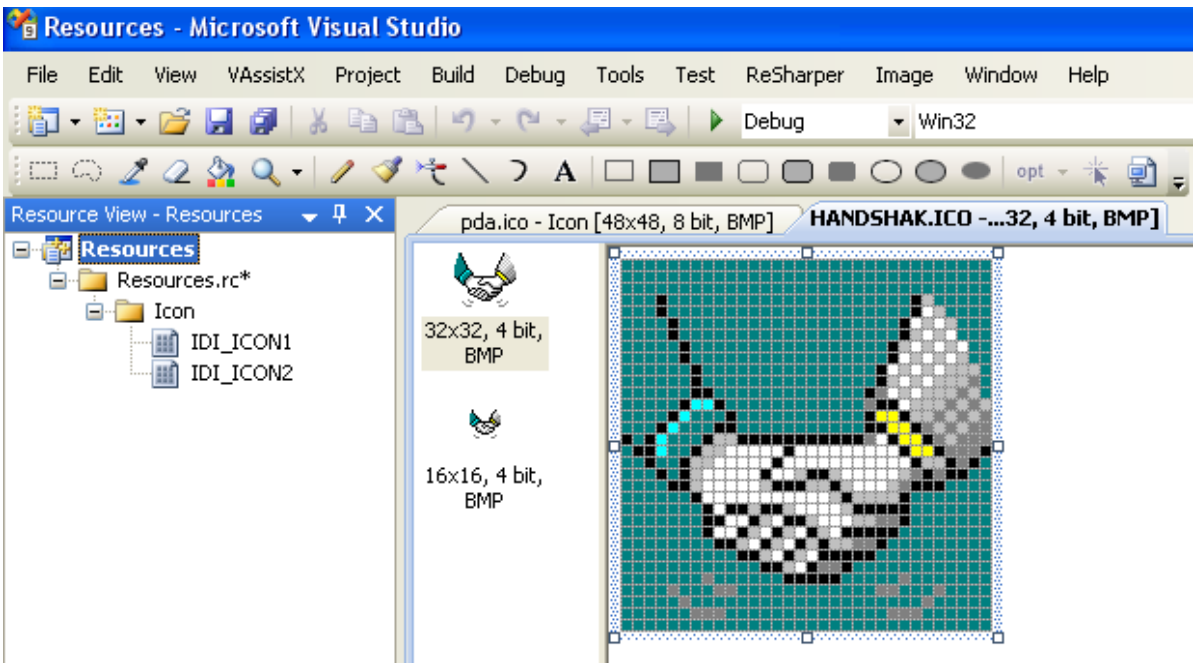


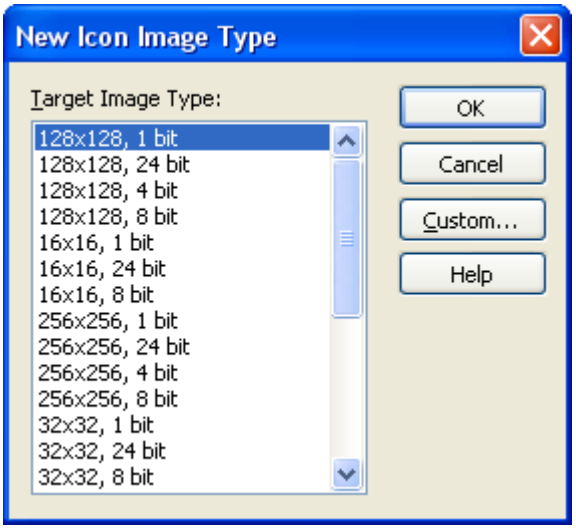
После добавления ресурса при компиляции проекта файл описания ресурсов транслируется компилятором ресурсов. В результате образуется бинарный файл с расширением **.res**. Затем компоновщик включает полученный файл в выполняемый файл программы вместе с кодом и данными программы из файлов с расширениями .**obj** и .**Lib**.

После вводного ознакомления слушателей с понятием ресурса приложения, следует перейти к рассмотрению таких ресурсов, как пиктограмма и курсор.

4.1. Пиктограмма

Пиктограммы (иконки) — это небольшие растровые изображения, применяемые Windows для визуального представления приложений, файлов и папок. Чаще всего для приложения создают иконки следующих типовых размеров: 16 x 16 пикселей для малых иконок и 32 x 32 пикселя для стандартных иконок. Однако существует возможность создания иконки иного размера. Для этого, вызвав контекстное меню в редакторе иконки, необходимо выбрать пункт **New Image Type...**, и в появившемся диалоговом окне выбрать нужный тип иконки.





Созданная иконка сохраняется в файл с расширением **.ico**, а описание иконки добавляется в файл описания ресурсов с расширением **.rc**. При этом иконке назначается идентификатор (например, **IDI_ICON1**), который впоследствии можно изменить. Рекомендуется присваивать ресурсам идентификаторы, отражающие их семантику.

Наряду с файлом описания ресурсов, редактор ресурсов создает еще и заголовочный файл **resource.h**, содержащий определения используемых именованных констант.

Для использования в программе иконки, находящейся в ресурсах приложения, иконку следует загрузить с помощью рассмотренной ранее функции API **LoadIcon**:

```
HICON LoadIcon (
    HINSTANCE hInst, //дескриптор экземпляра приложения, содержащего иконку
    LPCTSTR lpszName //строка, содержащая имя иконки
    );
```

Дескриптор экземпляра приложения приходит в первом параметре функции **WinMain**. Альтернативным способом получения дескриптора экземпляра приложения является вызов функции API **GetModuleHandle**:

```
HMODULE GetModuleHandle(
    LPCTSTR lpModuleName /* имя DLL модуля */
    );
```

При нулевом значении параметра функции она вернёт дескриптор экземпляра приложения.

Во втором параметре функции **LoadIcon** требуется строка с именем иконки. Поскольку имя иконки представляет собой целочисленный идентификатор (например, **IDI_ICON1**), то его можно преобразовать в строку с помощью макроса **MAKEINTRESOURCE** (make an integer into resource string):

```
#define MAKEINTRESOURCE(i)  (LPTSTR) ((DWORD) ((WORD) (i)))
```

В качестве примера получения дескриптора иконки привести следующий фрагмент кода:

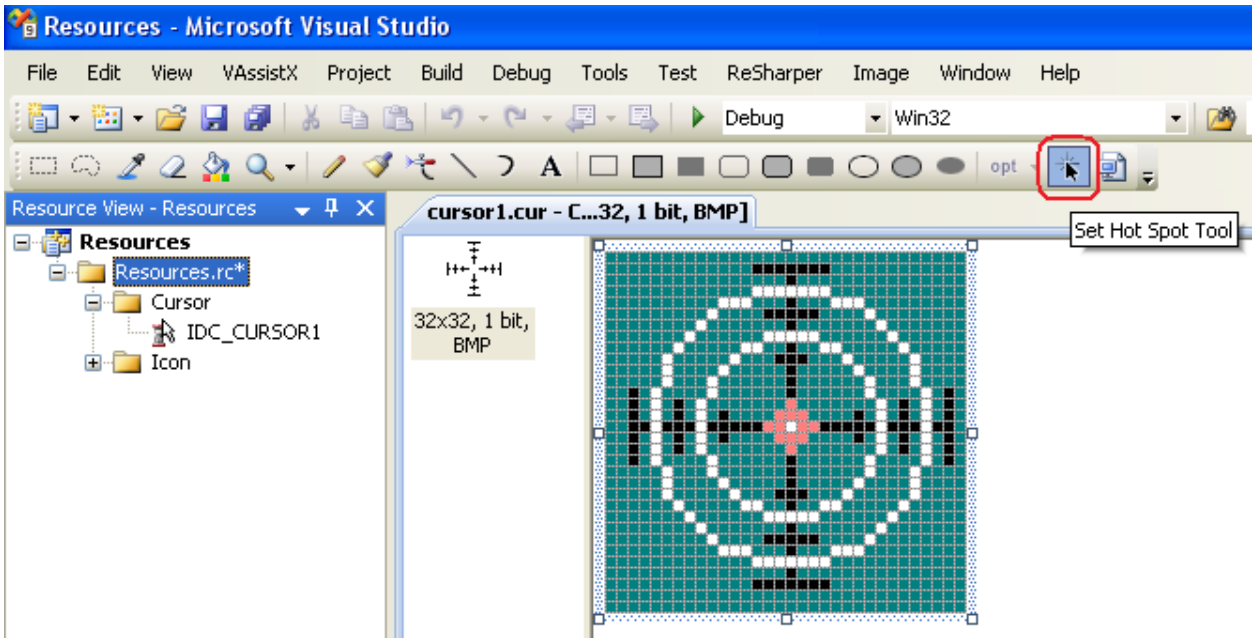
```
HINSTANCE hInstance = GetModuleHandle(0);
HICON hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON1));
```

4.2. Курсор

Курсор — это изображение размером 32 x 32 пикселя, которое отмечает положение указателя мыши. Курсор во многом похож на иконку. Главное отличие заключается в наличии активной точки (**hotspot**). Активной точкой называется пиксель, который принадлежит изображению курсора и отмечает его точное положение на экране в любой момент времени. В стандартном курсоре, имеющем вид стрелки, активная точка расположена в левом верхнем углу курсора.

Чтобы назначить активную точку, нужно выбрать инструмент **Set Hot Spot Tool**, а затем щелкнуть мышью на том пикселе изображения, который должен стать активной точкой.

Созданный курсор сохраняется в файл с расширением **.cur**, а описание курсора добавляется в файл описания ресурсов. При этом курсору назначается идентификатор (например, **IDC_CURSOR1**), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.



Для использования в программе курсора, находящегося в ресурсах приложения, курсор следует загрузить с помощью рассмотренной ранее функции API **LoadCursor**:

```
HCURSOR LoadCursor (
      HINSTANCE hInst, // дескриптор экземпляра приложения, содержащего курсор
      LPCTSTR lpszName // строка, содержащая имя курсора
      );
```

Во втором параметре функции **LoadCursor** требуется строка с именем курсора. Поскольку имя курсора представляет собой целочисленный идентификатор (например, **IDC_CURSOR1**), то его можно преобразовать в строку с помощью рассмотренного выше макроса **MAKEINTRESOURCE.**

В качестве примера получения дескриптора курсора привести следующий фрагмент кода:

```
      HINSTANCE hInstance = GetModuleHandle(0);
      HCURSOR hCursor = LoadCursor(hInstance, MAKEINTRESOURCE(IDC_CURSOR1));
```

Как отмечалось ранее, иконка и курсор приложения указываются на этапе определения класса окна при инициализации структуры **WNDCLASSEX**. Однако

существует возможность модифицировать оконный класс, в частности, определить для приложения другую иконку или курсор. Для этой цели служит функция API **SetClassLong**:

```
DWORD SetClassLong(
    HWND hWnd, //дескриптор окна с типом класса, который нужно модифицировать
    int nIndex, // данный параметр указывает, что необходимо изменить
    LONG dwNewLong // новое значение для замены
);
```

В качестве примера модификации оконного класса привести следующий фрагмент кода:

```
HICON hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON1));
SetClassLong(hWnd, GCL_HICON, LONG(hIcon));
HCURSOR hCursor1 = LoadCursor(hInstance, MAKEINTRESOURCE(IDC_CURSOR1));
SetClassLong(hWnd, GCL_HCURSOR, LONG(hCursor1));
```

Для динамического изменения формы курсора в зависимости от его местонахождения применяется функция API **SetCursor**:

```
HCURSOR SetCursor(
      HCURSOR hCursor // дескриптор курсора
      );
```

В качестве практического примера использования иконок и курсоров, определённых в ресурсах приложения, следует рассмотреть следующий код:

```
#include <windows.h>
#include "resource.h"
#include <tchar.h>

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
HICON hIcon;
HCURSOR hCursor1, hCursor2;

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                     LPTSTR lpszCmdLine, int nCmdShow)
{
```

```
	HWND hWnd;
	MSG Msg;
	WNDCLASSEX wcl;
	wcl.cbSize = sizeof(wcl);
	wcl.style = CS_HREDRAW | CS_VREDRAW;
	wcl.lpfnWndProc = WindowProc;
	wcl.cbClsExtra = 0;
	wcl.cbWndExtra = 0;
	wcl.hInstance = hInst;
	// иконка загружается из ресурсов приложения
	wcl.hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_ICON1));
	// курсор загружается из ресурсов приложения
	wcl.hCursor = LoadCursor(hInst, MAKEINTRESOURCE(IDC_CURSOR1));
	wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
	wcl.lpszMenuName = NULL;
	wcl.lpszClassName = szClassWindow;
	wcl.hIconSm = NULL;
	if (!RegisterClassEx(&wcl))
		return 0;
	hWnd = CreateWindowEx(0, szClassWindow, TEXT("Ресурсы"),
	WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
	CW_USEDEFAULT, NULL, NULL, hInst, NULL);
	ShowWindow(hWnd, nCmdShow);
	UpdateWindow(hWnd);
	while(GetMessage(&Msg, NULL, 0, 0))
	{
		TranslateMessage(&Msg);
		DispatchMessage(&Msg);
	}
	return Msg.wParam;
}

LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{
	switch(message)
	{
		case WM_DESTROY:
			PostQuitMessage(0);
			break;
		case WM_CREATE:
			{
				// получаем дескриптор приложения
				HINSTANCE hInstance = GetModuleHandle(0);
				// загружаем иконку из ресурсов приложения
				hIcon = LoadIcon(hInstance,
					MAKEINTRESOURCE(IDI_ICON2));
				// загружаем курсоры из ресурсов приложения
				hCursor1 = LoadCursor(hInstance,
					MAKEINTRESOURCE(IDC_CURSOR1));
				hCursor2 = LoadCursor(hInstance,
					MAKEINTRESOURCE(IDC_CURSOR2));
			}
			break;
		case WM_KEYDOWN:
			if(wParam == VK_RETURN)
			// устанавливаем иконку
			hIcon = (HICON) SetClassLong(hWnd, GCL_HICON, LONG(hIcon));
			break;
```

```
            case WM_MOUSEMOVE:
                  {
                        // устанавливаем тот или иной курсор в зависимости от
                        // местонахождения указателя мыши
                        RECT rect;
                        GetClientRect(hWnd, &rect);
                        int x = LOWORD(lParam);
                        int y = HIWORD(lParam);
                        if(x >= rect.right / 4 && x <= rect.right * 3 / 4 &&
                           y >= rect.bottom / 4 && y <= rect.bottom * 3 / 4)
                              SetCursor(hCursor1);
                        else
                              SetCursor(hCursor2);
                  }
                  break;
            default:
                  return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

5. Обработка ошибок

Акцентировать внимание слушателей на актуальности данного вопроса, так как важным критерием оценки любого приложения является надёжность его работы.

При использовании в коде программы той или иной функции API необходимо понимать, как в этих функциях устроена обработка ошибок. Обычно, при вызове функции Windows, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если передан недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, функция возвращает значение, свидетельствующее об ошибке. Например, некоторые функции имеют логический тип **BOOL** возвращаемого значения. В этом случае ложное возвращаемое значение обозначает ошибку. Другим примером является набор функций, которые возвращают дескриптор некоторого объекта **HANDLE**. Если вызов одной из таких функций заканчивается неудачно, то обычно возвращается **NULL.** При возникновении ошибки желательно разобраться, почему вызов данной функции оказался неудачен. Следует отметить, что за каждой ошибкой закреплен свой код — 32-битное число, которое можно получить, вызвав функцию API **GetLastError**. Данную функцию нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян. Если **GetLastError** возвращает

нулевое значение, это означает, что предшествующий вызов функции Windows завершился успешно.

В некоторых случаях было бы удобно получить описание ошибки на основе кода ошибки. Для этого цели предусмотрена функция API **FormatMessage**:

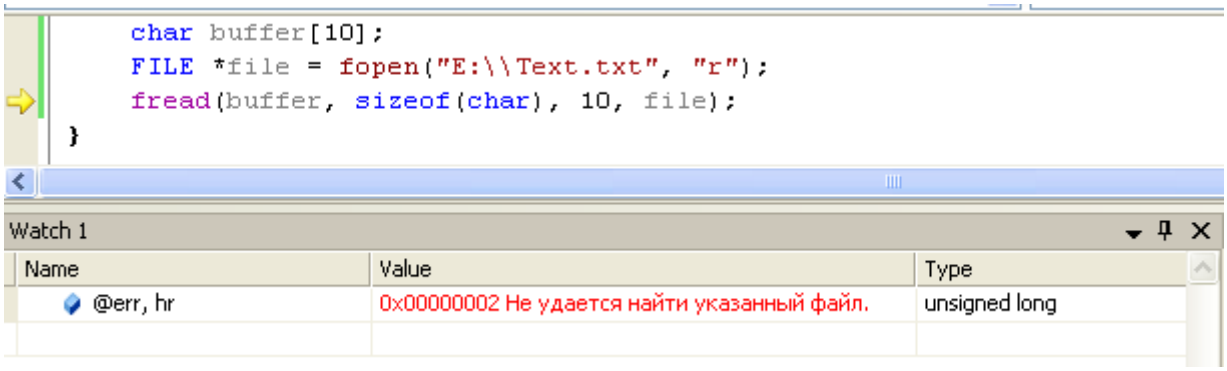
```
DWORD FormatMessage(
      DWORD dwFlags, // набор битовых флагов, которые определяют различные
      // аспекты процесса форматирования, а также способ интерпретации
      // 2-го параметра lpSource
      LPCVOID lpSource, // указатель на строку, содержащую сообщение об ошибке
      DWORD dwMessageId, // код ошибки
      DWORD dwLanguageId, // идентификатор языка, на котором выводится
      // описание ошибки
      LPTSTR lpBuffer, // выходной буфер, который выделяется системой, если
      // в 1-м параметре указан флаг FORMAT_MESSAGE_ALLOCATE_BUFFER
      DWORD nSize, // число символов, записываемых в выходной буфер,
      // либо минимальный размер выделяемого буфера, если
      // в 1-м параметре указан флаг FORMAT_MESSAGE_ALLOCATE_BUFFER
      va_list* Arguments // список аргументов форматирования
      );
```

В качестве примера использования вышеописанных функций можно привести следующий фрагмент кода:

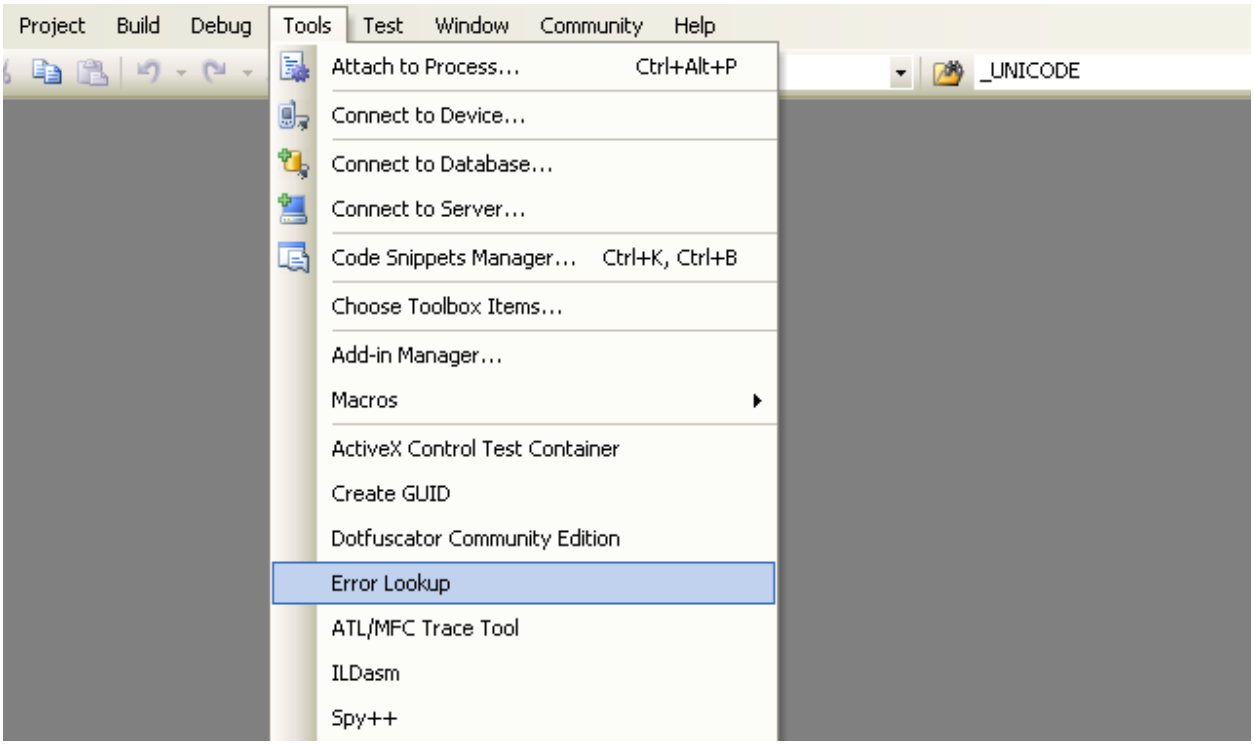
```
DWORD dwError = GetLastError(); // получим код последней ошибки
LPVOID lpMsgBuf = NULL;
TCHAR szBuf[300];
 //Функция FormatMessage форматирует строку сообщения
BOOL fOK = FormatMessage(
      FORMAT_MESSAGE_FROM_SYSTEM /* флаг сообщает функции, что нужна строка,
      соответствующая коду ошибки, определённому в системе */
      | FORMAT_MESSAGE_ALLOCATE_BUFFER /* необходимо выделить соответствующий
      блок памяти для хранения текста с описанием ошибки */,
      NULL /* текст с описанием ошибки будет находиться в буфере, выделенном
      системой. Адрес задается в 5-м параметре */,
      dwError /* код ошибки */,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT) /* идентификатор языка, на
      котором выводится описание ошибки */,
      (LPTSTR) &lpMsgBuf /* указатель на буфер, в который запишется текст с
      описанием ошибки */,
      0, // память выделяет система
      NULL // список аргументов форматирования
      );
      if(lpMsgBuf != NULL)
      {
            wsprintf(szBuf, TEXT("Ошибка %d: %s"), dwError, lpMsgBuf);
```

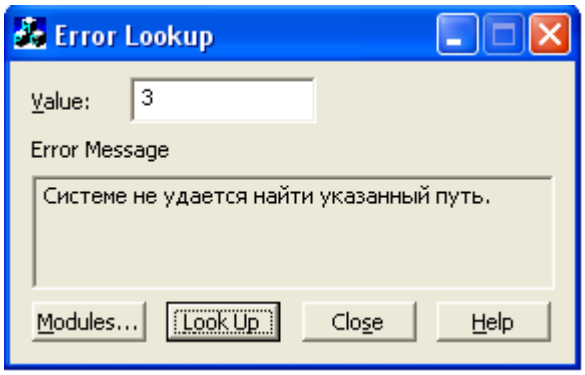
```
            MessageBox(hDialog, szBuf, TEXT("Сообщение об ошибке"),
                                MB_OK | MB_ICONSTOP);
            LocalFree(lpMsgBuf); // освободим память, выделенную системой
      }
```

Обратить внимание слушателей, что особенно полезно отслеживать код последней ошибки в процессе отладки. Отметить, что отладчик в Microsort Visual Studio позволяет настраивать окно **Watch** так, чтобы оно всегда показывало код и описание последней ошибки после выполнения текущей команды. Для этого необходимо в окне **Watch** ввести **«@err,hr»**.



Ознакомить слушателей с утилитой **Error Lookup**, которая позволяет получить описание ошибки по ее коду.





6. Практическая часть

Написать приложение, обладающее следующей функциональностью:

- после нажатия клавиши <**Enter**> через каждую секунду (или иной промежуток времени) «прячется» одна из кнопок «Калькулятора», выбранная случайным образом;
- после нажатия клавиши <**Esc**> данный процесс останавливается.

7. Подведение итогов

Подвести общие итоги занятия. Отметить возможность использования таймера в приложении, при этом выделив два основных подхода обработки прерываний таймера. Напомнить слушателям, что существует возможность определить, какие приложения, обладающие окном, выполняются в данное время и какие дочерние окна у них имеются. Ещё раз акцентировать внимание слушателей на необходимости контроля работы функций Windows и обработки ошибок в случае неудачного завершения работы функций.

8. Домашнее задание

1. Написать приложение, обладающее следующей функциональностью:

- при нажатии клавиши <**Enter**> главное окно позиционируется в левый верхний угол экрана с размерами (300x300) и начинает перемещаться по периметру экрана с определённой скоростью;
- при нажатии клавиши <**Esc**> перемещение окна прекращается.

2. Написать приложение, обладающее следующей функциональностью:
   - при последовательном нажатии клавиши <**Enter**> дочерние окна главного окна приложения «Калькулятор» минимизируются;
   - при последовательном нажатии клавиши <**Esc**> дочерние окна восстанавливаются в обратном порядке, т.е. дочернее окно, которое минимизировалось последним, первым будет восстановлено.