Модуль №1

Занятие №2

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. Утилита Spy++.
3. «Новые» типы данных.
4. Венгерская нотация.
5. Этапы создания проекта «Win32 Project».
6. Минимальное WinAPI-приложение.
   - 6.1. Определение класса окна.
   - 6.2. Регистрация класса окна.
   - 6.3. Создание окна. Стили окна.
   - 6.4. Отображение окна.
   - 6.5. Цикл обработки сообщений.
   - 6.6. Оконная процедура.
7. Окна сообщений.
8. Подведение итогов.
9. Домашнее задание.

1. Повторение пройденного материала

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

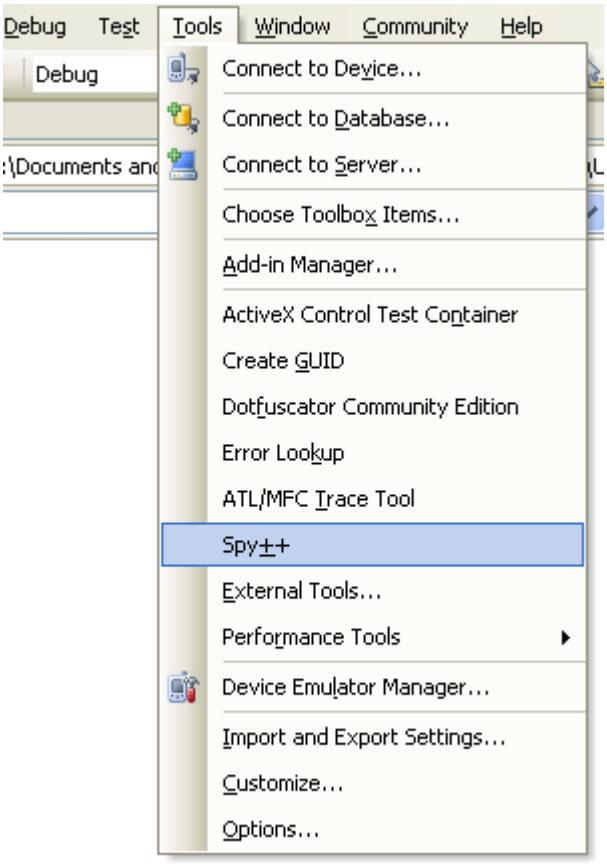
1) Что такое кодировка?
2) Чем отличается кодировка ANSI от Unicode-кодировки?
3) В чем состоит актуальность использования Unicode-кодировки?

4) Какой тип данных используется для работы с Unicode?
5) Какие средства для работы с Unicode предоставляет библиотека C++?
6) Что необходимо использовать для создания универсального кода, способного задействовать как ANSI-кодировку, так и Unicode-кодировку?
7) Какие основные преимущества операционной системы Windows перед операционной системой MS-DOS?
8) Что такое многозадачность?
9) Что такое многопоточность?
10) В чём заключается независимость приложения от аппаратных средств?
11) Какова структура и способ исполнения консольного приложения?
12) Что такое событие?
13) Что такое сообщение?
14) Какова архитектура приложений, построенных на событиях? Чем подобная архитектура отличается от архитектуры консольных приложений?
15) Что такое системная очередь сообщений?
16) Что такое очередь сообщений приложения?
17) Что такое окно?
18) Что такое дескриптор окна?
19) Что такое оконная процедура?
20) Что такое оконный класс?

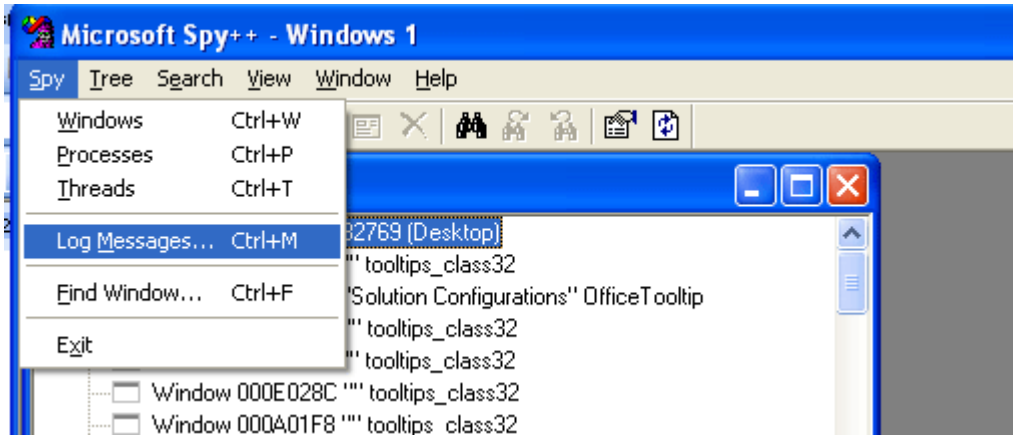
2. Утилита Spy++

Повторив материал предыдущего занятия, необходимо ознакомить слушателей с работой утилиты **Spy++.** Отметить, что в процессе отладки приложений иногда бывает полезным увидеть, какие сообщения вырабатывает

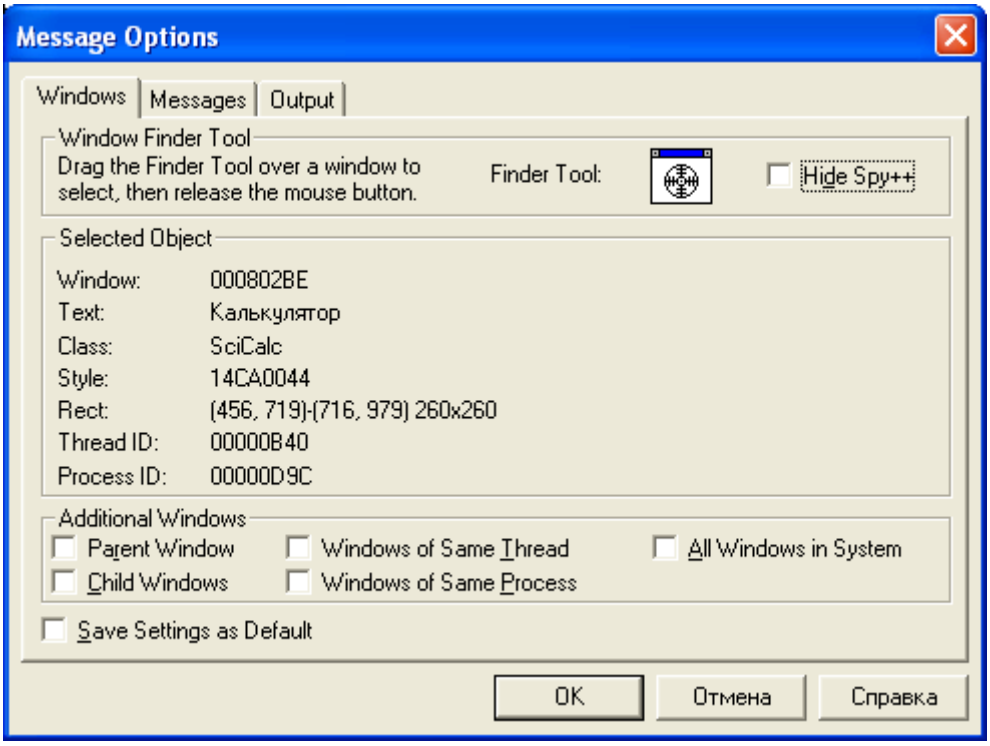
Windows в ответ на те или иные действия пользователя. В составе интегрированной среды Microsoft Visual Studio есть удобное инструментальное средство для решения данной проблемы — утилита Spy++. Эта программа «шпионит» за окном другого приложения, чтобы разработчик мог иметь представление о том, какие сообщения проходят через это окно.



Продемонстрировать работу данной утилиты на примере шпионского наблюдения за приложением «Калькулятор». Технология шпионского наблюдения за интересующим приложением довольно проста. Запустив на выполнение приложение «Калькулятор», необходимо разместить на экране окно утилиты Spy++ и окно приложения так, чтобы они были видны одновременно и не перекрывали друг друга.



Далее выполнить команду меню **Spy -> Log Messages**. В результате будет отображено диалоговое окно **Message Options**.

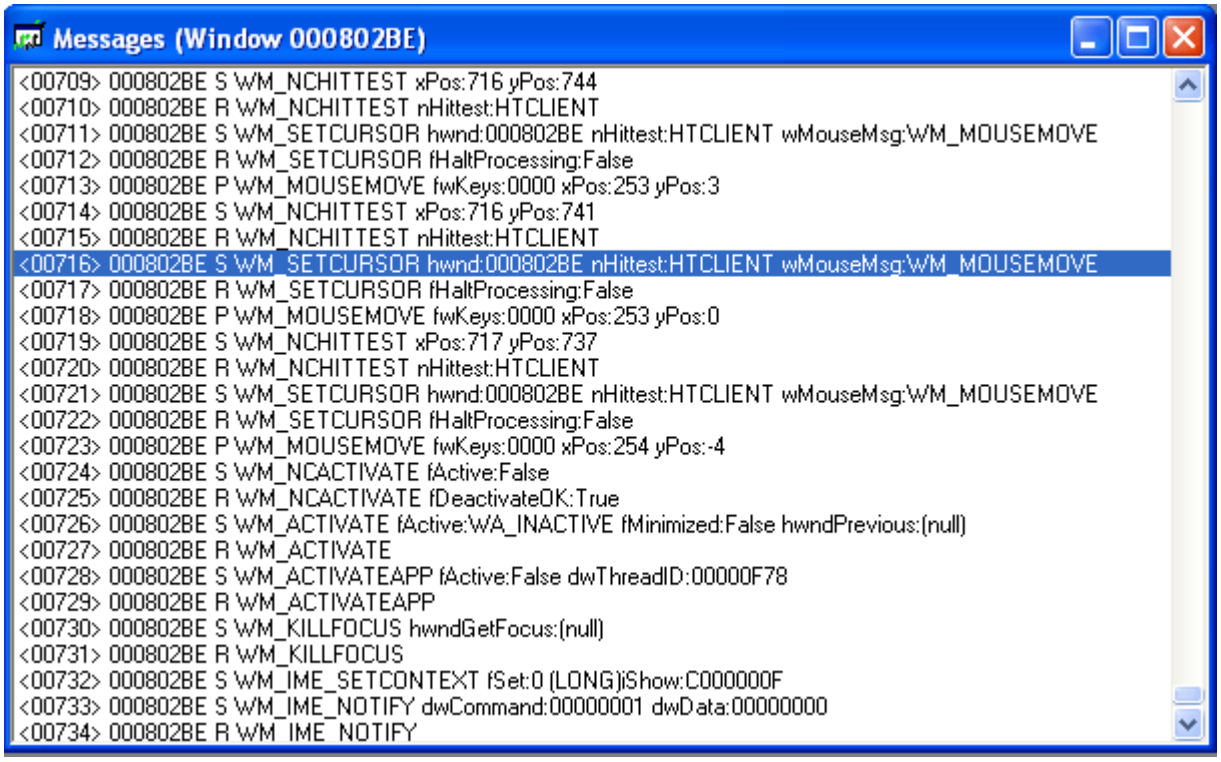


На вкладке **Windows** необходимо «схватить» пиктограмму **Finder Tool** и перетащить ее в окно приложения «Калькулятор». В группе **Selected Object** будет отображена следующая информация:

- дескриптор окна (Window);
- заголовок окна (Text);
- имя класса (Class);

- стиль (Style);
- координаты окна (Rect);
- дескриптор потока (Thread ID);
- дескриптор процесса (Process ID).

После нажатия кнопки OK будет отображено окно **Messages (Window...),** в котором утилита Spy++ будет фиксировать все сообщения, поступающие в «помеченное» окно.



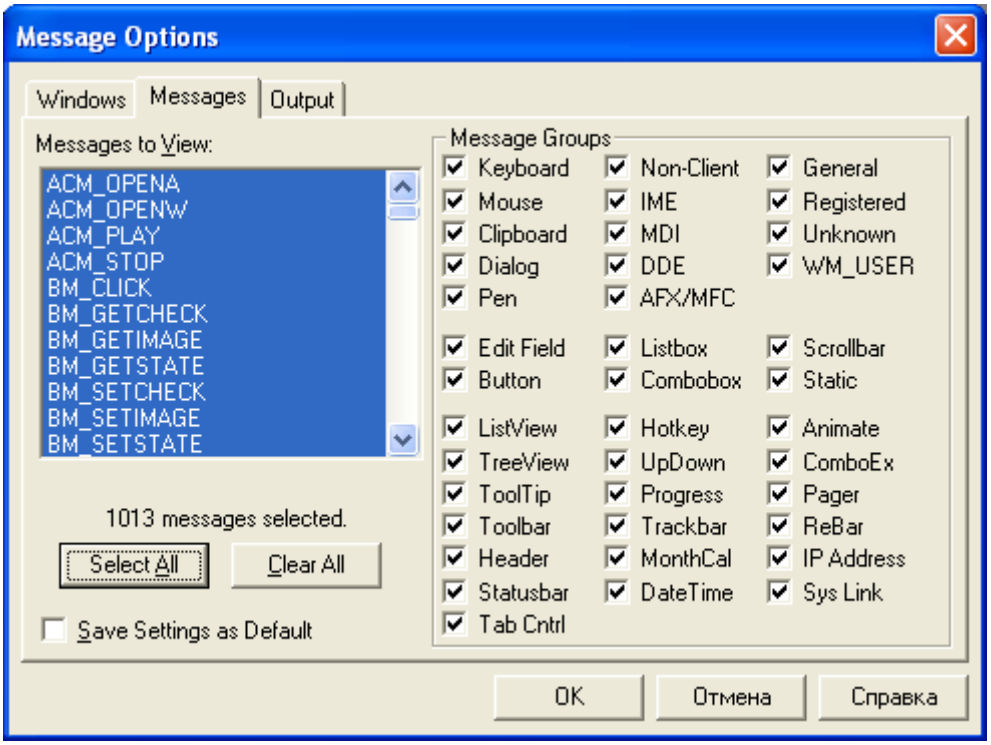
При этом каждое сообщение в окне Messages отображается в следующем формате:

1) порядковый номер сообщения;
2) дескриптор окна;
3) код сообщения;
4) идентификатор сообщения;
5) дополнительные параметры.

Чтобы упростить анализ работы приложения, можно фильтровать поток сообщений, поступающих в окно **Messages**. Для этого необходимо вернуться в диалоговое окно **Message Options** и перейти на вкладку Messages. В группе

Message Groups можно включить или исключить группу сообщений, относящихся либо к некоторому типу элемента управления, либо к некоторому внешнему источнику сообщений. Например, сообщения, поступающие от мыши. В списке **Messages to View** содержится полный перечень сообщений, и каждое из них можно включить или исключить. По умолчанию все сообщения включены.

Таким образом, использование утилиты Spy++ помогает иногда понять, почему приложение работает не так, как ожидается.



3. «Новые» типы данных

При ознакомлении слушателей с типами данных Windows, акцентировать внимание на том, что каждый тип данных является синонимом уже существующего типа языка C++. Например:

```
typedef int  BOOL
```

Привести следующую таблицу типов данных Windows:

Тип данных	Описание
BOOL	Булевский тип данных. Может принимать одно из двух значений TRUE или FALSE. Занимает 4 байта.
BYTE	1-байтное целое без знака.
COLORREF	Тип данных, используемый для работы с цветом. Занимает 4 байта.
DWORD	4-х байтное целое или адрес.
HANDLE	4-х байтное целое, используемое в качестве дескриптора.
HBITMAP	Дескриптор растрового изображения.
HBRUSH	Дескриптор кисти.
HCURSOR	Дескриптор курсора.
HDC	Дескриптор устройства.
HFONT	Дескриптор шрифта.
HICON	Дескриптор иконки.
HINSTANCE	Дескриптор экземпляра приложения.
HMENU	Дескриптор меню.
HWND	Дескриптор окна.
INT	4-х байтное целое со знаком.
LONG	4-х байтное целое со знаком.
LPARAM	Переменные этого типа передаются в качестве дополнительного параметра в функцию - обработчик какого-либо сообщения. В них обычно содержатся информация специфическая для данного события. Занимает 4 байта.
LPCSTR	4-х байтный указатель на константную строку символов. Указатели с приставкой LP обычно называют длинными указателями.
LPCWSTR	4-х байтный указатель на константную Unicode-строку.
LPSTR	4-х байтный указатель строку символов.
LPWSTR	4-х байтный указатель на Unicode-строку.
LRESULT	Значение типа LONG, возвращаемое оконной процедурой
UINT	4-х байтное целое без знака.
WORD	2-х байтное целое без знака.
WPARAM	Переменные этого типа передаются в качестве дополнительного параметра в функцию - обработчик какого-либо сообщения. В них обычно содержатся информация специфическая для данного события. Занимает 4 байта.

4. Венгерская нотация

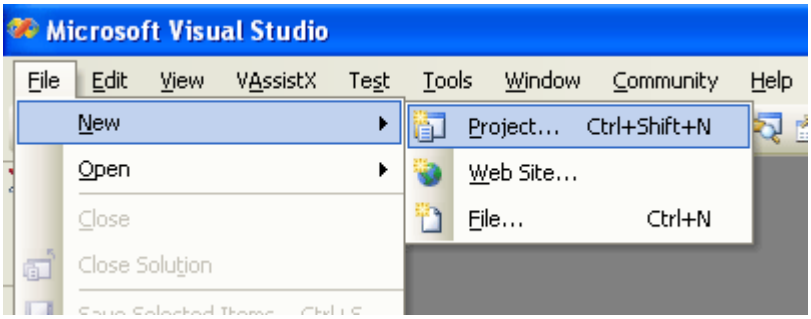
При рассмотрении данного вопроса подвести слушателей к мысли, что было бы удобно по имени переменной определить её назначение в программе, а также тип данных. Для решения этой проблемы программисты Microsoft предложили так называемую **венгерскую нотацию.** Она названа так потому, что ее в Microsoft популяризировал венгерский программист Чарльз Шимоньи (Charles Simonyi). В венгерской нотации переменным даются описательные имена, начинающиеся с заглавных букв. Например, Counter, Flag, BookTitle, AuthorName. Если имя состоит из нескольких слов, каждое слово начинается с заглавной буквы. Затем перед описательным именем добавляются буквы, чтобы указать тип переменной. Например, uCounter для типа unsigned int и bFlag для типа bool, szBookTitle для символьного массива (sz – string zero).

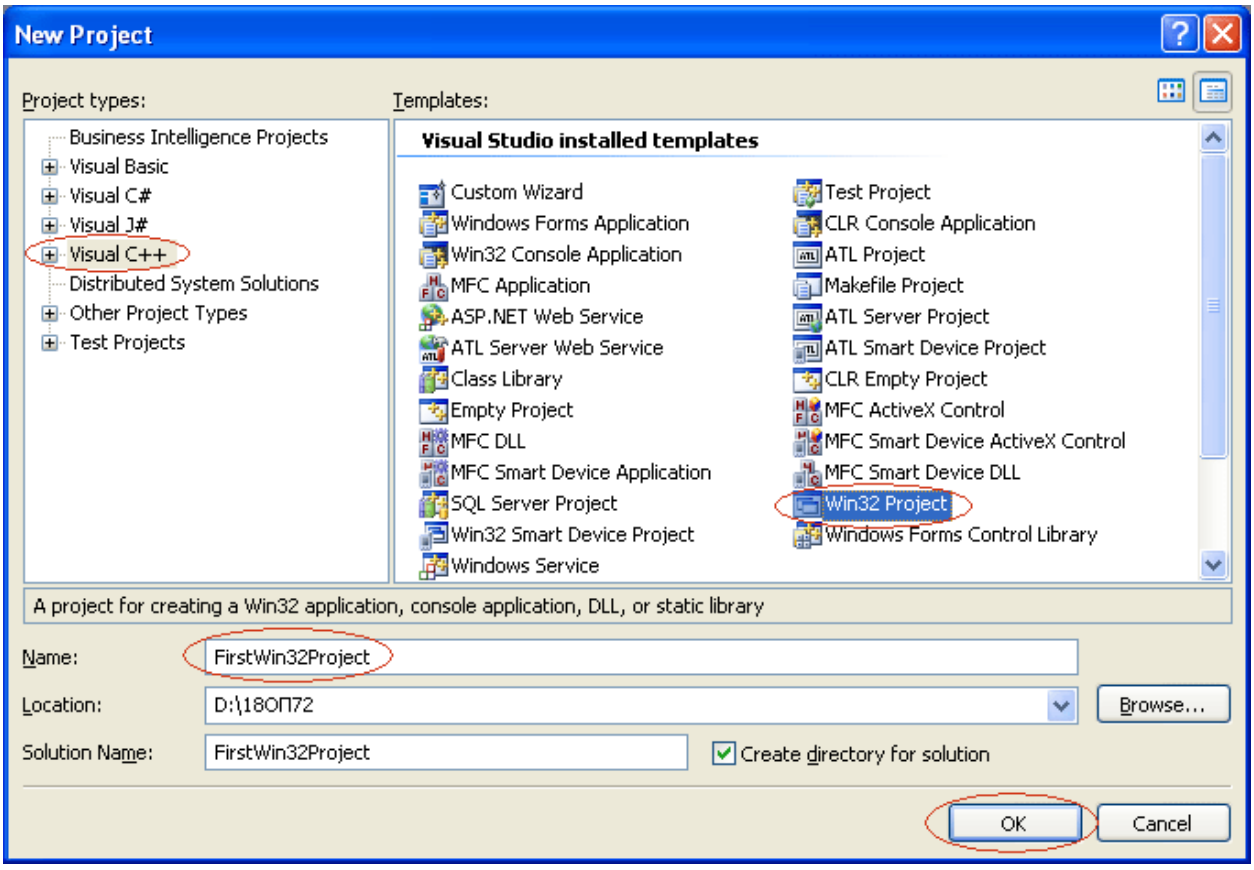
Обратить внимание слушателей, что большинство функций WinAPI используют венгерскую нотацию, поэтому, по меньшей мере, знать о ней необходимо. Подчеркнуть, что использование венгерской нотации желательно для написания понятного и читабельного кода. Отметить, что в венгерской нотации предлагаются следующие префиксы, приведя следующую таблицу:

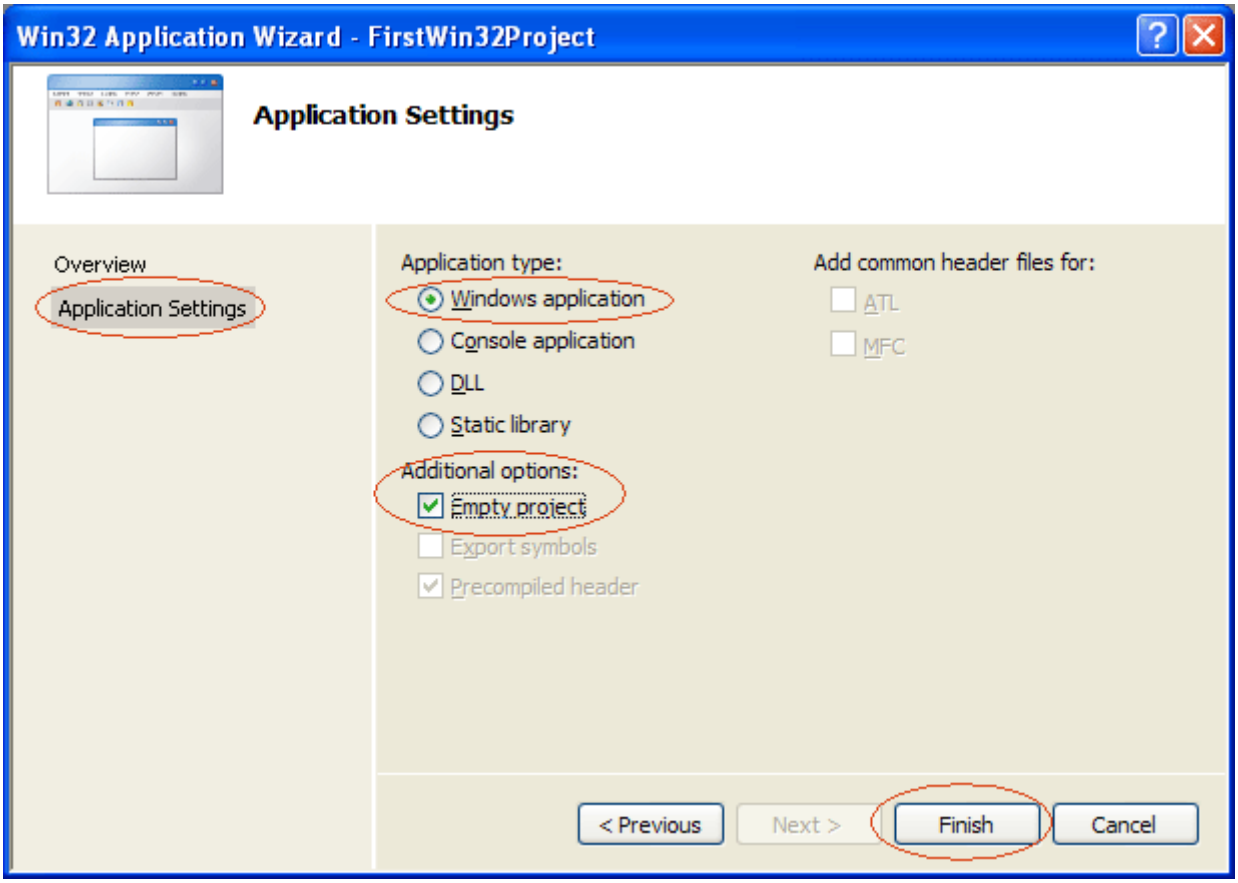
Префикс	Тип переменной
b	Логический тип (bool или BOOL)
i	Целое число (индекс)
n	Целое число (количество чего-либо)
u	Целое число без знака
d	Число с двойной точностью
sz	Строковая переменная, ограниченная нулем
p	Указатель
lp	Длинный указатель
a	Массив
lpfn	Длинный указатель на функцию
h	Дескриптор
cb	Счетчик байтов
C	Класс

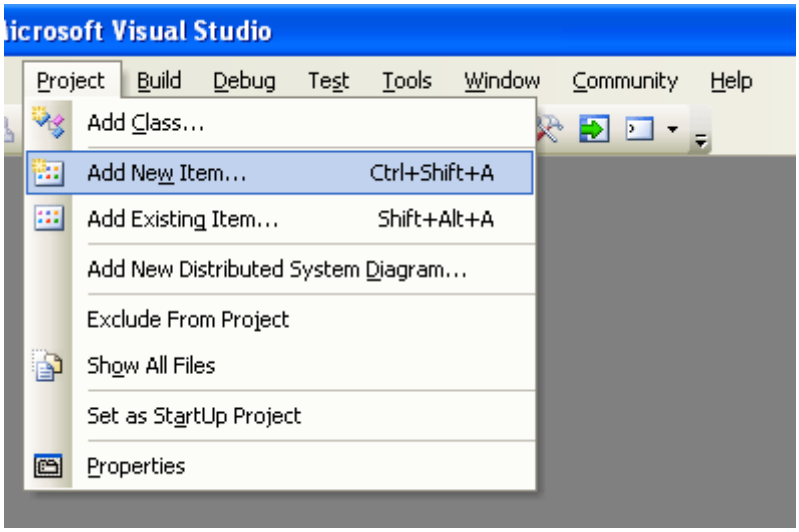
5. Этапы создания проекта «Win32 Project»

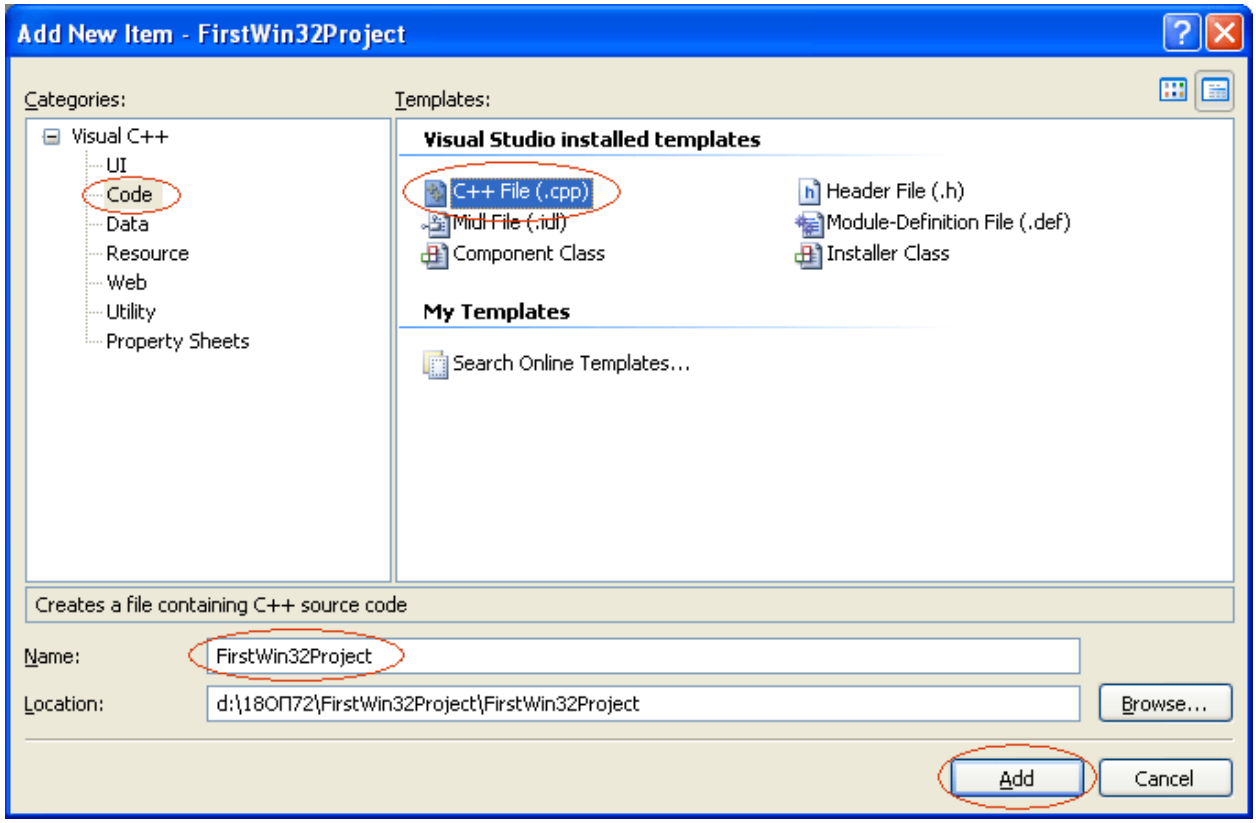
Обратить особое внимание слушателей на шаги формирования проекта Win32 Project.






Win32 Application Wizard - FirstWin32Project
Application Settings
Overview
Application Settings
Application type:
Windows application
Console application
DLL
Static library
Additional options:
Empty project
Export symbols
Precompiled header
Add common header files for:
ATL
MFC
< Previous
Next >
Finish
Cancel


Microsoft Visual Studio
Project
Build
Debug
Test
Tools
Window
Community
Help
Add Class...
Add New Item... Ctrl+Shift+A
Add Existing Item... Shift+Alt+A
Add New Distributed System Diagram...
Exclude From Project
Show All Files
Set as StartUp Project
Properties



6. Минимальное WinAPI-приложение

Минимальное WinAPI-приложение должно содержать как минимум две функции:

- **WinMain** — главную функцию, в которой создается основное окно программы и запускается цикл обработки сообщений;
- **WndProc** — оконную процедуру, обеспечивающую обработку сообщений для основного окна программы.

WinMain является точкой входа в программу и выполняет следующие действия:

- определение класса окна;
- регистрация класса окна;
- создание окна;
- отображение окна;
- запуск цикла обработки сообщений.

```
// Файл WINDOWS.H содержит определения, макросы, и структуры
// которые используются при написании приложений под Windows.
#include <windows.h>
#include <tchar.h>

//прототип оконной процедуры
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

TCHAR szClassWindow[] = TEXT("Каркасное приложение"); /* Имя класса окна */

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

      /* 1. Определение класса окна  */

      wcl.cbSize = sizeof (wcl);    // размер структуры WNDCLASSEX
      // Перерисовать всё окно, если изменён размер по горизонтали
      // или по вертикали
      wcl.style = CS_HREDRAW | CS_VREDRAW; // CS(Class Style) - стиль класса
      // окна
      wcl.lpfnWndProc = WindowProc; // адрес оконной процедуры
      wcl.cbClsExtra = 0;     // используется Windows
      wcl.cbWndExtra  = 0;    // используется Windows
      wcl.hInstance = hInst;  // дескриптор данного приложения

      // загрузка стандартной иконки
      wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);

      // загрузка стандартного курсора
      wcl.hCursor = LoadCursor(NULL, IDC_ARROW);

       // заполнение окна белым цветом
      wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      wcl.lpszMenuName = NULL;      //приложение не содержит меню
      wcl.lpszClassName = szClassWindow; //имя класса окна
      wcl.hIconSm = NULL; // отсутствие маленькой иконки

      /*  2. Регистрация класса окна  */

      if (!RegisterClassEx(&wcl))
            return 0; // при неудачной регистрации - выход

      /*  3. Создание окна  */

      // создается окно и  переменной hWnd присваивается дескриптор окна
      hWnd=CreateWindowEx(
            0,          // расширенный стиль окна
            szClassWindow,    // имя класса окна
            TEXT("Каркас  Windows приложения"), // заголовок окна
            WS_OVERLAPPEDWINDOW,    // стиль окна
            /* Заголовок, рамка, позволяющая менять размеры, системное меню,
                        кнопки развёртывания и свёртывания окна  */
            CW_USEDEFAULT,    // x-координата левого верхнего угла окна
            CW_USEDEFAULT,    // y-координата левого верхнего угла окна
            CW_USEDEFAULT,    // ширина окна
```

```
            CW_USEDEFAULT,    // высота окна
            NULL,             // дескриптор родительского окна
            NULL,             // дескриптор меню окна
            hInst,            // идентификатор приложения, создавшего окно
            NULL);            // указатель на область данных приложения

      /* 4. Отображение окна */

      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);     // перерисовка окна

      /* 5. Запуск цикла обработки сообщений  */

      // получение очередного сообщения из очереди сообщений
      while(GetMessage(&Msg, NULL, 0, 0))
      {
            TranslateMessage(&Msg); // трансляция сообщения
            DispatchMessage(&Msg);  // диспетчеризация сообщений
      }
      return Msg.wParam;
}

LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMessage, WPARAM wParam,
                            LPARAM lParam)
{
      switch(uMessage)
      {
            case WM_DESTROY:  // сообщение о завершении программы
                  PostQuitMessage(0);     // посылка сообщения WM_QUIT
                  break;
            default:
            // все сообщения, которые не обрабатываются в данной оконной
            // функции направляются обратно Windows на обработку по умолчанию
                  return DefWindowProc(hWnd, message, wParam, lParam);
      }
      return 0;
}
```

Необходимо дать подробные комментарии к каждой строке вышеприведенной программы. Отметить, что заголовочный файл **Windows.h** содержит определения, макросы, и структуры, которые используются при написании приложений под Windows. Таким образом, при программировании Windows-приложений с использованием WinAPI данный файл следует подключать всегда.

Главная функция приложения (точка входа в приложение) **_tWinMain** - это макроопределение, которое в зависимости от настроек проекта определяется по-разному. Если установлен **UNICODE**, **_tWinMain** определяется так:

```
#define _tWinMain   wWinMain
```

В ином случае **_tWinMain** определяется следующим образом:

```
#define _tWinMain   WinMain
```

Функция **_tWinMain** имеет следующий прототип:

```
INT WINAPI _tWinMain(
      HINSTANCE hInst, // дескриптор экземпляра приложения
      HINSTANCE hPrevInst, //равен 0 и необходим для совместимости
      LPTSTR  lpszCmdLine,  //  указатель  на  строку,  в  которую  копируются
      //аргументы приложения, если оно запущено в режиме командной строки
      int nCmdShow // способ визуализации окна при запуске программы
);
```

Спецификатор **WINAPI** определяет соглашение о вызове функции. Наиболее распространены два протокола вызова функции:

1. __ **cdecl** – по данному протоколу вызывающая функция сама очищает стек после вызываемой функции. При этом передача параметров функции в стек происходит в порядке справа налево. По данному протоколу происходит вызов функции только в языке C/C++. Это связано с тем, что в языке C имеется семейство функций с произвольным количеством параметров (**printf**).
2. __ **stdcall (WINAPI, CALLBACK)** – согласно этому протоколу вызываемая функция сама за собой очищает стек. При этом передача параметров функции в стек происходит в порядке справа налево. Такой вызов используется в других языках программирования (например, Паскаль, Фортран).

6.1. Определение класса окна

Для определения класса окна в функции **WinMain** заполняются поля структуры **WNDCLASSEX:**

```
typedef struct tagWNDCLASSEX {
UINT cbSize; // размер данной структуры в байтах
UINT style; // стиль класса окна
WNDPROC lpfnWndProc; // указатель на функцию окна (оконную процедуру)
int cbClsExtra; // число дополнительных байтов, которые должны
//быть распределены в конце структуры класса
int cbWndExtra; // число дополнительных байтов, которые должны
//быть распределены вслед за экземпляром окна
HINSTANCE hInstance; // дескриптор экземпляра приложения, в котором
//находится оконная процедура для этого класса
HICON hIcon; // дескриптор иконки
HCURSOR hCursor; // дескриптор курсора
HBRUSH hbrBackground; //дескриптор кисти, используемой для закраски фона окна
LPCTSTR lpszMenuName; // указатель на строку, содержащую имя меню,
//применяемого по умолчанию для этого класса
LPCTSTR lpszClassName; // указатель на строку, содержащую имя класса окна
HICON hIconSm; // дескриптор малой иконки
} WNDCLASSEX;
```

В рассматриваемом приложении в поле **style** структуры **WNDCLASSEX** указана комбинация стилей **CS_HREDRAW | CS_VREDRAW**. Это означает, что окно будет перерисовано, если изменён размер по горизонтали или по вертикали.

В поле **hIcon** устанавливается дескриптор иконки, возвращаемый функцией API **LoadIcon***:*

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

Для того чтобы использовать встроенные типы иконок Windows, первый параметр должен быть равен NULL, а в качестве второго параметра должен использоваться один из следующих макросов:

Макрос	Форма иконки
IDI_APPLICATION	Стандартная иконка для приложения
IDI_ASTERISK	Иконка "информация"
IDI_EXCLAMATION	Иконка "восклицательный знак"

Макрос	Форма иконки
IDI_HAND	Иконка "знак Стоп"
IDI_QUESTION	Иконка "вопросительный знак"

В поле **hCursor** устанавливается дескриптор курсора, возвращаемый функцией API **LoadCursor**:

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

Для того чтобы использовать встроенный тип курсора Windows, первый параметр должен быть равен NULL, а в качестве второго параметра должен использоваться один из следующих макросов:

Макрос	Форма иконки
IDC_ARROW	Стандартный курсор - стрелка
IDC_CROSS	Перекрестье
IDC_IBEAM	Текстовый двутавр
IDC_WAIT	"Песочные часы"
IDC_HELP	Стрелка и вопросительный знак
IDC_SIZEALL	Четырехконечная стрелка

Поле **hbrBackground** инициализируется дескриптором кисти, используемым для закраски фона окна. Кисть (brush) — это графический объект, который представляет собой шаблон пикселов различных цветов, используемый для закрашивания области. В Windows имеется несколько стандартных или предопределенных кистей. Вызов функции API **GetStockObject** с аргументом **WHITE_BRUSH** возвращает дескриптор белой кисти. Так как возвращаемое значение имеет тип **HGDIOBJ**, то его необходимо преобразовать к типу **HBRUSH**.

```
HGDIOBJ GetStockObject(
int object  //предопределённый объект GDI
);
```

6.2. Регистрация класса окна

Когда класс окна полностью определен, он должен быть зарегистрирован в системе. Для этого используется функция API **RegisterClassEx,** возвращающая значение, идентифицирующее зарегистрированный класс окна:

```
ATOM RegisterClassEx(
    CONST WNDCLASS * lpWClass // адрес структуры WNDCLASSEX
);
```

6.3. Создание окна. Стили окна

После того как класс окна определен и зарегистрирован, можно создавать окна этого класса, используя функцию API **CreateWindowEx:**

```
HWND CreateWindowEx(
DWORD dwExStyle, // расширенный стиль окна
LPCTSTR lpClassName, // имя класса окна
LPCTSTR lpWinName, // заголовок окна
DWORD dwStyle, // стиль окна
int x, int y, // координаты верхнего левого угла
int Width, int Height, // размеры окна
HWND hParent, // дескриптор родительского окна
HMENU hMenu, // дескриптор главного меню
HINSTANCE hThisInst, // дескриптор приложения
LPVOID lpszAdditional // указатель на дополнительную информацию
);
```

Первый параметр **dwExStyle** задает расширенный стиль окна, применяемый совместно со стилем, определенным в параметре **dwStyle**.
Например, в качестве расширенного стиля можно задать один или несколько флагов, приведенных в следующей таблице.

Стиль	Описание
WS_EX_ACCEPTFILES	Создать окно, которое принимает перетаскиваемые файлы
WS_EX_CLIENTEDGE	Рамка окна имеет утопленный край
WS_EX_CONTROLPARENT	Разрешить пользователю перемещаться по дочерним окнам с помощью клавиши Tab
WS_EX_MDICHILD	Создать дочернее окно многодокументного интерфейса
WS_EX_STATICEDGE	Создать окно с трехмерной рамкой. Этот стиль предназначен для элементов, которые не принимают ввод от пользователя
WS_EX_TOOLWINDOW	Создать окно с инструментами, предназначенное для реализации плавающих панелей инструментов
WS_EX_TRANSPARENT	Создать прозрачное окно. Любые окна того же уровня, накрываемые этим окном, получат сообщение WM_PAINT в первую очередь
WS_EX_WINDOWEDGE	Создать окно, имеющее рамку с активизированным краем

В нашем приложении в качестве основного стиля (параметр **dwStyle**) используется макрос **WS_OVERLAPPEDWINDOW,** который определяет стандартное окно, имеющее системное меню, заголовок, рамку для изменения размеров, а также кнопки минимизации, развертки и закрытия. Используемый стиль окна является наиболее общим. Допускается создавать окна с другими стилями, некоторые из которых приведены в следующей таблице.

Стиль	Описание
WS_OVERLAPPED	Стандартное окно с рамкой
WS_MAXIMIZEBOX	Наличие кнопки развертки
WS_MINIMIZEBOX	Наличие кнопки минимизации
WS_SYSMENU	Наличие системного меню
WS_HSCROLL	Наличие горизонтальной панели прокрутки
WS_VSCROLL	Наличие вертикальной панели прокрутки

В нашем приложении для параметров **x, y, Width** и **Height** функции **CreateWindowEx** используется макрос **CW_USEDEFAULT**, что позволяет системе самостоятельно выбирать координаты и размеры окна. Если окно не имеет родительского окна, как в случае нашего приложения, то параметр

hParent должен быть равен **HWND_DESKTOP** (или **NULL** - это тоже допускается).

6.4. Отображение окна

Для отображения на экране созданного окна вызывается функция **ShowWindow**, имеющая следующий прототип:

```
BOOL ShowWindow(
HWND hWnd, //дескриптор окна
int nCmdShow //способ отображения окна
);
```

При начальном отображении главного окна рекомендуется присваивать второму параметру то значение, которое передается приложению через параметр **nCmdShow** функции **WinMain**. При последующих отображениях можно использовать любое из значений, приведенных в следующей таблице.

Макрос	Эффект
SW_HIDE	Скрыть окно
SW_MAXIMIZE	Развернуть окно
SW_MINIMIZE	Свернуть окно
SW_SHOW	Активизировать окно и показать в его текущих размерах и позиции
SW_RESTORE	Отобразить окно в нормальном представлении

Рекомендуется после вызова функции **ShowWindow** вызвать функцию **UpdateWindow**, которая посылает оконной процедуре сообщение **WM_PAINT**, заставляющее окно перерисовать свою клиентскую область.

```
BOOL UpdateWindow( HWND hWnd );
```

6.5. Цикл обработки сообщений

Последней частью функции **WinMain** является **цикл обработки сообщений**. Его целью является получение и обработка сообщений, передаваемых

операционной системой. Эти сообщения ставятся в очередь сообщений приложения, откуда они затем (по мере готовности программы) выбираются функцией API **GetMessage:**

```
BOOL GetMessage(
LPMSG lpMsg, //адрес структуры MSG, в которую помещается выбранное сообщение
HWND hwnd, // дескриптор окна, принимающего сообщение
/* Обычно значение этого параметра равно NULL, что позволяет выбрать сообщения для любого окна приложения. */
UINT min, // минимальный номер принимаемого  сообщения
UINT max // максимальный номер принимаемого сообщения
/* Если оба последних параметра равны нулю, то функция выбирает из очереди любое очередное сообщение. */
);
```

```
typedef struct tagMSG {
HWND hwnd; - дескриптор окна, которому адресовано сообщение
UINT message; - идентификатор сообщения
WPARAM wParam; - дополнительная информация
LPARAM lParam; - дополнительная информация
DWORD time; - время отправки сообщения
POINT pt; - экранные координаты курсора мыши в момент отправки сообщения
} MSG;
```

```
typedef struct tagPOINT {
  LONG x; //координата X точки
  LONG y; //координата Y точки
} POINT, *PPOINT;
```

Функция **GetMessage** возвращает значение **TRUE** при извлечении любого сообщения, кроме одного — **WM_QUIT**. Получив сообщение **WM_QUIT**, функция возвращает значение **FALSE**. В результате этого происходит немедленный выход из цикла, и приложение завершает работу, возвращая операционной системе код возврата **msg.wParam.**

Вызов **TranslateMessage** нужен только в тех приложениях, которые должны обрабатывать ввод данных с клавиатуры. Дело в том, что для обеспечения независимости от аппаратных платформ и различных

национальных раскладок клавиатуры в Windows реализована двухуровневая схема обработки сообщений от символьных клавиш. Сначала система генерирует сообщения о так называемых виртуальных клавишах, например, сообщение **WM_KEYDOWN** — когда клавиша нажимается, и сообщение **WM_KEYUP** — когда клавиша отпускается. В сообщении **WM_KEYDOWN** содержится также информация о так называемом скан-коде нажатой клавиши.

Функция API **TranslateMessage** преобразует пару аппаратных сообщений, **WM_KEYDOWN** и **WM_KEYUP**, в символьное сообщение **WM_CHAR**, которое содержит ASCII-код символа (wParam). Сообщение **WM_CHAR** помещается в очередь, а на следующей итерации цикла функция **GetMessage** извлекает его для последующей обработки.

Функция API **DispatchMessage** передает структуру MSG обратно в Windows. Windows отправляет сообщение для его обработки соответствующей оконной процедуре.

6.6. Оконная процедура

Оконная процедура является **функцией обратного вызова** или **CALLBACK–функцией**. В коде приложения прямого вызова CALLBACK-функции нет! Такая функция всегда вызывается операционной системой.

Оконная процедура получает в качестве параметров сообщения из очереди сообщений данного приложения.

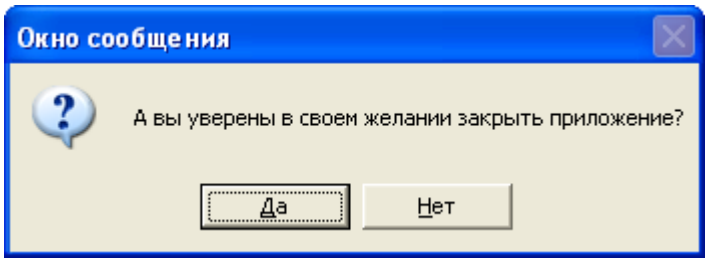
```
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMessage, WPARAM wp, LPARAM lp);
```

Четыре параметра оконной процедуры идентичны первым четырем полям структуры **MSG**. Первый параметр функции содержит дескриптор окна, получающего сообщение. Во втором параметре указывается идентификатор сообщения. Для системных сообщений зарезервированы номера от 0 до 1024. Третий и четвертый параметры содержат дополнительную информацию, которая распознается системой в зависимости от типа полученного сообщения.

Обычно в оконной процедуре используют оператор **switch** для определения того, какое сообщение получено и как его обрабатывать. Если сообщение обрабатывается, то оконная процедура обязана вернуть нулевое значение. Все сообщения, не обрабатываемые оконной процедурой, должны передаваться системной функции **DefWindowProc**. В этом случае оконная процедура должна вернуть то значение, которое возвращает **DefWindowProc**.

При обработке сообщения **WM_DESTROY** оконная процедура вызывает функцию API **PostQuitMessage**. Значение параметра этой функции будет использовано как код возврата программы. Вызов **PostQuitMessage** приводит к посылке приложению сообщения **WM_QUIT**, получив которое, функция **GetMessage** возвращает нулевое значение и завершает тем самым цикл обработки сообщений и, следовательно, приложение.

7. Окна сообщений



Окно сообщений, которое вызывается функцией API **MessageBox**, является простейшим типом диалогового окна. Функция **MessageBox** позволяет создавать, отображать и выполнять различные действия с окном сообщения. Окно сообщения содержит текст, определяемый приложением, и заголовок, а также любое сочетание предопределенных пиктограмм и кнопок.

Функция **MessageBox** чаще всего используется для сообщений об ошибках и предупреждающих сообщений. Она имеет следующий прототип:

```
int MessageBox(
HWND hWnd, // дескриптор родительского окна
/* Если этот параметр равен 0, то окно сообщения не имеет окна-владельца. */
LPCTSTR lpText, //указатель на строку, содержащую текст, который должен быть
```

```
//отображен в окне
LPCTSTR lpCaption, //указатель на строку, которая отображается в заголовке
// диалогового окна

/* Если этот параметр равен NULL, то применяется  заданный по умолчанию
заголовок Error. */
UINT uType /* этот параметр является комбинацией значений, которые определяют
свойства окна сообщения, включающие типы кнопок, которые должны
присутствовать, и дополнительную иконку рядом с текстом сообщения */
);
```

Значение параметра uType	Описание
MB_OK	Окно содержит только кнопку OK (Подтверждение)
MB_OKCANCEL	Окно содержит две кнопки: OK (Подтверждение) и Cancel (Отмена)
MB_RETRYCANCEL	Окно содержит две кнопки: Retry и Cancel (Повтор и Отмена)
MB_ABORTRETRYIGNORE	Окно содержит три кнопки: Abort, Retry и Ignore (Стоп, Повтор и Пропустить)
MB_YESNO	Окно содержит две кнопки: Yes и No (Да и Нет)
MB_YESNOCANCEL	Окно содержит три кнопки: Yes, No и Cancel (Да, Нет и Отмена)

Для добавления иконки в **MessageBox** необходимо комбинировать указанные значения параметра **uType** со следующими значениями:

Значение параметра uType	Описание
MB_ICONEXCLAMATION	Отображается иконка «восклицательный знак»
MB_ICONINFORMATION	Отображается иконка «информация»
MB_ICONQUESTION	Отображается иконка «вопросительный знак»
MB_ICONSTOP (или MB_ICONHAND)	Отображается иконка – «стоп»

Для назначения кнопки по умолчанию, можно использовать одну из следующих констант:

- **MB_DEFBUTTON1** (кнопка по умолчанию - первая);
- **MB_DEFBUTTON2** (вторая);
- **MB_DEFBUTTON3** (третья).

Функция **MessageBox** возвращает одно из следующих значений:

Значение	Описание
IDOK	Была нажата кнопка OK
IDCANCEL	Была нажата кнопка Cancel (или клавиша <Esc>)
IDABORT	Была нажата кнопка Abort
IDIGNORE	Была нажата кнопка Ignore
IDYES	Была нажата кнопка Yes
IDNO	Была нажата кнопка No
IDRETRY	Была нажата кнопка Retry
0	Произошла ошибка при создании окна сообщений

8. Подведение итогов

Подвести общие итоги занятия. Еще раз акцентировать внимание слушателей на необходимости использования утилиты Spy++, помогающей иногда понять, почему приложение работает не так, как ожидается. Напомнить слушателям, что минимальное WinAPI-приложение должно содержать, как минимум, две функции: главную функцию программы WinMain и оконную процедуру. Отметить, какие основные действия выполняются в функции WinMain и какую роль играет оконная процедура. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

9. Домашнее задание

1. Написать приложение, позволяющее вывести на экран краткое резюме с помощью последовательности окон сообщений (количество окон сообщений – не менее трёх). На заголовке последнего окна сообщения должно отобразиться среднее число символов на странице (общее число символов в резюме поделить на количество окон сообщений).

2. Написать приложение, которое «угадывает» задуманное пользователем число от 1 до 100. Для запроса к пользователю использовать окна сообщений. После того, как число отгадано, необходимо вывести количество попыток, потребовавшихся для этого, и предоставить пользователю возможность сыграть

еще раз, не завершая программу. Окна сообщений следует оформить кнопками и иконками в соответствии с конкретной ситуацией.

При выполнении вышеприведенных заданий ввиду отсутствия необходимости создания главного окна приложения, реализацию алгоритма можно привести непосредственно в главной функции программы. Тогда каркас приложения может иметь следующий вид:

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

INT WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                        LPTSTR lpszCmdLine, int nCmdShow)
{
     MessageBox(
     0,
     TEXT("Реализация алгоритма программы непосредственно в функции WinMain"),
     TEXT("Окно сообщения"),
     MB_OK | MB_ICONINFORMATION);
    return 0;
}
```