# Урок №1

## Содержание

# 1. UNICODE

## 1.1. Понятие кодировки. Unicode кодировка

**Кодировка – это определённый набор символов (англ. *character set*), в котором каждому символу сопоставляется последовательность длиной в один или несколько байтов.**

В обычной кодировке **ANSI** (*American National Standards Institute*) каждый символ определяется восемью битами, т.е. всего можно закодировать 256 символов. Однако существуют такие языки и системы письменности (например, японские иероглифы), в которых столько знаков, что однобайтового набора символов недостаточно. Для поддержки таких языков и для облегчения «перевода» программ на другие языки была создана кодировка **Unicode**. Каждый символ в Unicode состоит из двух байтов, что позволяет расширить набор символов до 65536. Существенное отличие от 256 символов, доступных в ANSI–кодировке! Таким образом, стандарт Unicode позволяет представить символы практически всех языков мира, что даёт возможность легко обмениваться данными на разных языках, а также распространять единственный двоичный EXE- или DLL- файл, поддерживающий все языки.

## 1.2. Unicode и библиотека C++ (C/C++ Run-Time Library)

Специально для использования Unicode-строк был введён «новый» тип данных **wchar_t**.

| wchar_t — тип данных для Unicode-символа<br>typedef unsigned short wchar_t |
|---|

Следует отметить, что для работы с Unicode-строками существует набор Unicode-функций, которые эквивалентны строковым функциям ANSI C. Рассмотрим следующую таблицу соответствий:

| Строковая функция ANSI C | Эквивалентная Unicode-функция |
|---|---|
| char * strcat( char *, const char *); | wchar_t * wcscat( wchar_t *, const wchar t *); |
| char * strchr(const char *, int); | wchar_t * wcschr(const wchar_t *, wchar_t); |
| int strcmp(const char *, const char *); | int wcscmp(const wchar_t *, const wchar_t *); |
| char * strcpy(char *, const char *); | wchar_t * wcscpy(wchar_t *, const wchar_t *); |
| size_t strlen(const char *); | size_t wcslen(const wchar_t *); |

Из таблицы видно, что имена всех новых функций начинаются с **wcs** — это аббревиатура **wide character set** (набор широких символов) Таким образом, имена Unicode-функций образуются простой заменой префикса **str** соответствующих ANSI-функций на **wcs**.

В качестве примера работы с Unicode-функциями рассмотрим следующий код:

```
#include <iostream>
using namespace std;

void main()
{
	// ANSI-кодировка
	char szBuf1[15] = "Hello,";
	strcat(szBuf1, " world!");
	cout << sizeof(szBuf1) << " bytes\n"; // 15 байт

	// UNICODE-кодировка
	wchar_t szBuf2[15] = L"Hello,";
	wcscat(szBuf2, L" world!");
	cout << sizeof(szBuf2) << " bytes\n"; // 30 байт
}
```

Буква **L** перед строковым литералом указывает компилятору, что строка состоит из символов Unicode.

## 1.3. Макросы для работы с Unicode

Существует возможность создания универсального кода, способного задействовать как ANSI-кодировку, так и Unicode-кодировку. Для этого необходимо подключить файл **tchar.h**, в котором имеются макросы, заменяющие явные вызовы **str-** или **wcs-**функций. При этом если в программе определена символическая константа **_UNICODE**, то макросы будут ссылаться на **wcs**-функции, в противном случае макросы будут ссылаться на **str**-функции.

```
// определение символической константы _UNICODE
#define _UNICODE
#include <iostream>
#include <tchar.h>
using namespace std;

void main()
{
	TCHAR szBuf3[15] = _TEXT("Hello,");
	_tcscat(szBuf3, _TEXT(" world!"));
	wcout << szBuf3 << '\n';
	cout <<"The size of array: "<<sizeof(szBuf3)<<" bytes\n"; // 30 байт
}
```

```
// отсутствие символической константы _UNICODE
#include <iostream>
#include <tchar.h>
using namespace std;

void main()
{
	TCHAR szBuf3[15] = _TEXT("Hello");
	_tcscat(szBuf3, _TEXT(" world!"));
	wcout << szBuf3 << '\n';
	cout << "The size of array: " << sizeof(szBuf3) << " bytes\n"; //15 байт
}
```

Как следует из вышеприведенного кода, для объявления символьного массива универсального назначения (ANSI/Unicode) применяется тип данных **TCHAR**. Если макрос **_UNICODE** определен, **TCHAR** объявляется так:

```
typedef wchar_t TCHAR;
```

В ином случае **TCHAR** объявляется следующим образом:

```
typedef char TCHAR
```

Макрос **_TEXT** избирательно ставит заглавную букву L перед строковым литералом. Если **_UNICODE** определен, **_TEXT** определяется так:

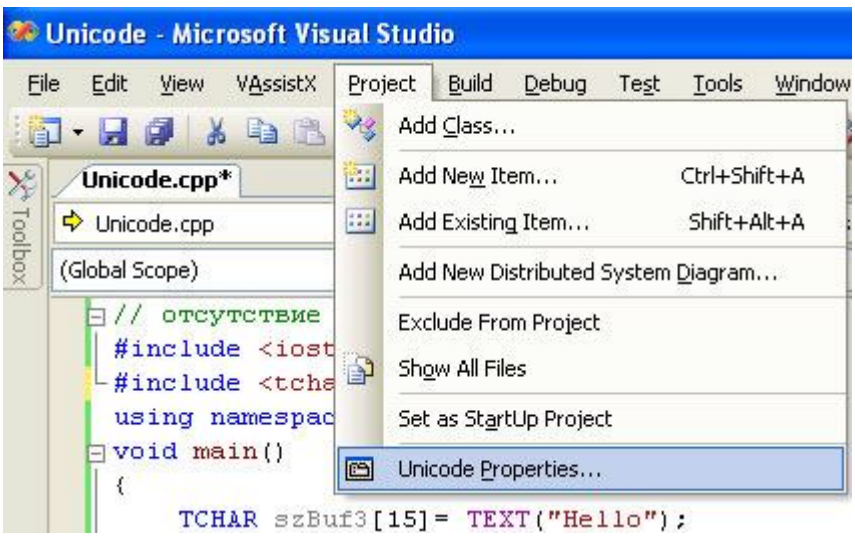
```
#define _TEXT(x)  L##x
```

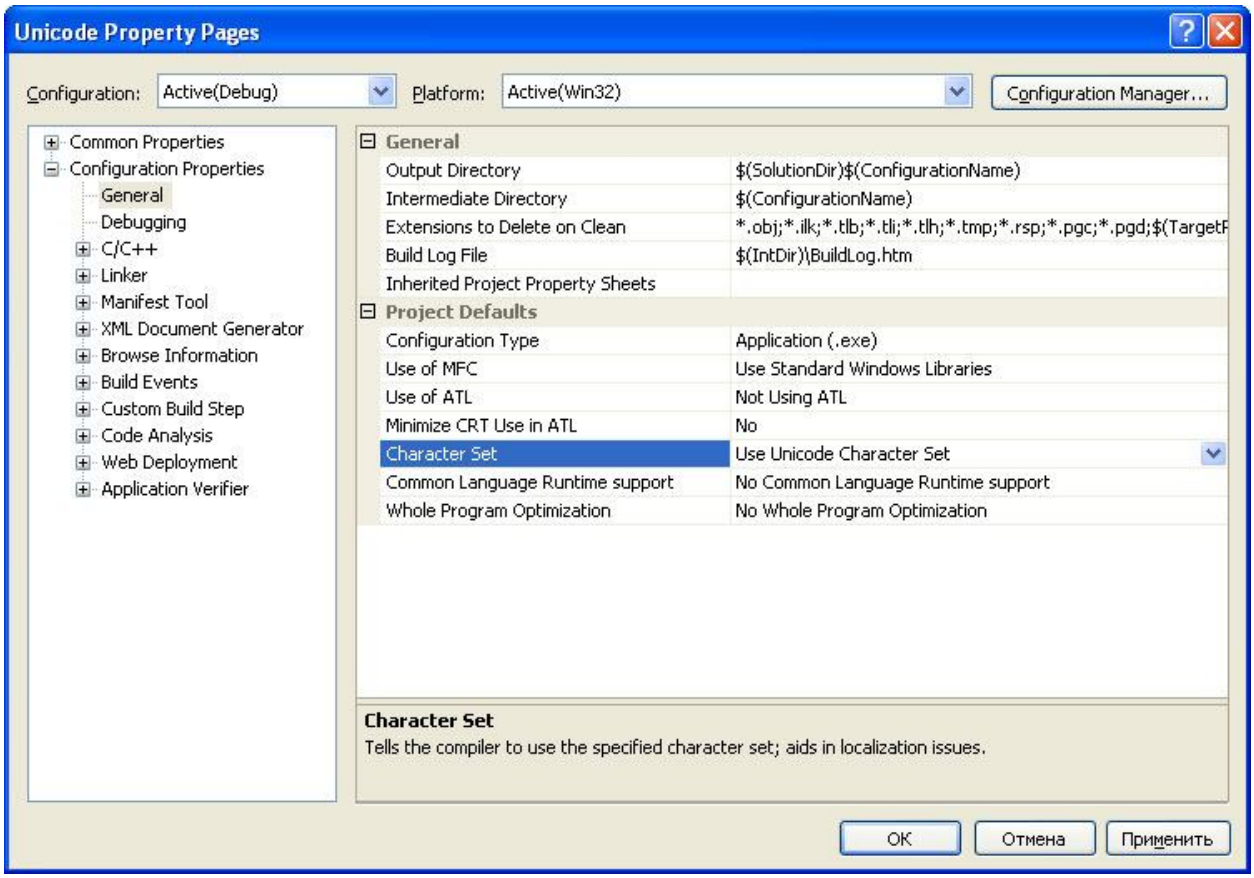
В ином случае **_TEXT** определяется следующим образом:

```
#define _TEXT(x)  x
```

где x – строка символов.

Следует отметить, что существует альтернативный способ задания Unicode-кодировки через свойства проекта.






## 1.4. Строковые функции Windows

Строковые функции Windows похожи на строковые функции из библиотеки C/C++, например на **strcpy** и **wcscpy**. Однако функции Windows являются частью операционной системы, и многие её компоненты используют именно их, а не аналоги из библиотеки C/C++. Рекомендуется отдать предпочтение функциям операционной системы. Это немного повысит быстродействие программы.

| Функция | Описание |
|---|---|
| lstrcat | Выполняет конкатенацию строк |
| lstrcmp | Сравнивает две строки с учетом регистра букв |
| lstrcmpi | Сравнивает две строки без учета регистра букв |

| Функция | Описание |
|---|---|
| lstrcpy | Копирует строку в другой участок памяти |
| lstrlen | Возвращает длину строки в символах |

Следует отметить, что функции Windows реализованы как макросы, вызывающие либо Unicode-, либо ANSI-версию функции в зависимости от того, определен ли UNICODE при компиляции исходного модуля. Например, если UNICODE не определен, **lstrcat** раскрывается в **lstrcatA**, определен — в **lstrcatW**.

## 1.5. Перекодировка строк из ANSI в Unicode

Для преобразования мультибайтовых символов строки в Unicode-строку используется следующая функция:

```
int MultiByteToWideChar(
  UINT CodePage,          // кодовая страница
  DWORD dwFlags,          // дополнительные настройки, влияющие на
//преобразование букв с диакритическими знаками
  LPCSTR lpMultiByteStr, // указатель на преобразуемую строку
  int cbMultiByte,        // длина строки в байтах
  LPWSTR lpWideCharStr,  // указатель на буфер, куда запишется Unicode-строка
  int cchWideChar         // размер буфера
);
```

В качестве примера рассмотрим следующий код:

```
#include <windows.h>
#include <iostream>
using namespace std;

void main()
{
      char buffer[] =
      "MultiByteToWideChar converts ANSI-string to Unicode-string";
```

```
	// определим размер памяти, необходимый для хранения Unicode-строки
	int length = MultiByteToWideChar(CP_ACP /*ANSI code page*/, 0, buffer,
	-1, NULL, 0);
	wchar_t *ptr = new wchar_t[length];
	// конвертируем ANSI-строку в Unicode-строку
	MultiByteToWideChar(CP_ACP, 0, buffer, -1, ptr, length);
	wcout << ptr << endl;
	cout << "Length of Unicode-string: " << wcslen(ptr) << endl;
	cout << "Size of allocated memory: " << _msize(ptr) << endl;
	delete[] ptr;
}
```

Как видно из примера, первый вызов функции **MultiByteToWideChar** определяет размер памяти, необходимый для хранения Unicode-строки, в то время как второй вызов этой функции выполняет непосредственно перекодировку. Кроме того, следует обратить внимание на использование объекта **wcout** вместо **cout** для вывода Unicode-строки.

Аналогичные преобразования выполняет функция из C/C++ Run-Time Library:

```
size_t mbstowcs(
   wchar_t * wcstr,//преобразованная Unicode-строка
   const char * mbstr,//исходная ANSI-строка
   size_t count //максимальное число символов исходной строки
);
```

В качестве примера рассмотрим следующий код:

```
#include <iostream>
using namespace std;

void main()
{
	char buffer[] = "mbstowcs converts ANSI-string to Unicode-string";
	// определим размер памяти, необходимый для хранения Unicode-строки
	int length = mbstowcs(NULL, buffer, 0);
	wchar_t *ptr = new wchar_t[length];
	// конвертируем ANSI-строку в Unicode-строку
	mbstowcs(ptr, buffer, length);
	wcout << ptr;
```

```
	cout << "\nLength of Unicode-string: " << length << endl;
	cout << "Size of allocated memory: " << _msize(ptr) << " bytes" << endl;
	delete[] ptr;
}
```

## 1.6. Перекодировка строк из Unicode в ANSI

Для преобразования Unicode-строк в ANSI используется следующая функция:

```
int WideCharToMultiByte(
  UINT CodePage,          // кодовая страница
  DWORD dwFlags,          // дополнительные настройки, влияющие на
//преобразование букв с диакритическими знаками
  LPCWSTR lpWideCharStr,  // указатель на преобразуемую Unicode-строку
  int cchWideChar,        // количество символов в строке.
  LPSTR lpMultiByteStr,  // указатель на буфер, куда запишется новая строка
  int cbMultiByte,          // размер буфера
  LPCSTR lpDefaultChar,     // функция использует символ по умолчанию, если
//преобразуемый символ не представлен в кодовой странице
  LPBOOL lpUsedDefaultChar  // указатель на флаг, сигнализирующий об успешном
//преобразовании всех символов (в этом случае - FALSE)
);
```

В качестве примера, иллюстрирующего работу функции, рассмотрим следующий код:

```
#include <windows.h>
#include <iostream>
using namespace std;

void main()
{
	wchar_t buffer[] =
	L"WideCharToMultiByte converts Unicode-string to ANSI-string";
```

```
	// определим размер памяти, необходимый для хранения ANSI-строки
	int length = WideCharToMultiByte(CP_ACP /*ANSI code page*/, 0, buffer,
	-1, NULL, 0, 0, 0);
	char *ptr = new char[length];
	// конвертируем Unicode-строку в ANSI-строку
	WideCharToMultiByte(CP_ACP, 0, buffer, -1, ptr, length, 0, 0);
	cout << ptr << endl;
	cout << "Length of ANSI-string: " << strlen(ptr) << endl;
	cout << "Size of allocated memory: " << _msize(ptr) << endl;
	delete[] ptr;
}
```

Следует отметить, что аналогичные преобразования выполняет функция из C/C++ Run-Time Library:

```
size_t wcstombs(
   char * mbstr, //преобразованная ANSI-строка
   const wchar_t * wcstr, //исходная Unicode-строка
   size_t count //максимальное число символов исходной строки
);
```

В качестве примера, иллюстрирующего работу функции, рассмотрим следующий код:

```
#include <iostream>
using namespace std;

void main()
{
	wchar_t buffer[] = L"wcstombs converts Unicode-string to ANSI-string";
	// определим размер памяти, необходимый для хранения преобразованной
	// ANSI-строки
	int length = wcstombs(NULL, buffer, 0);
	char *ptr = new char[length + 1];
	// конвертируем Unicode-строку в ANSI-строку
	wcstombs(ptr, buffer, length + 1);
	cout << ptr;
	cout << "\nLength of ANSI-string: " << strlen(ptr) << endl;
	cout << "Size of allocated memory: " << _msize(ptr) << " bytes" << endl;
	delete[] ptr;
}
```

Исходные коды приведенных выше примеров находятся в папке **SOURCE/UNICODE**.

# 2. Программирование Windows-приложений. Введение

Операционная система Windows по сравнению с операционными системами типа MS-DOS обладает серьезными преимуществами и для пользователей, и для программистов. Среди этих преимуществ обычно выделяют:

- графический интерфейс пользователя;
- многозадачность и многопоточность;
- управление памятью;
- независимость от аппаратных средств.

## 2.1. Графический интерфейс пользователя

Графический интерфейс пользователя **GUI (Graphical User Interface)** дает возможность пользователям работать с приложениями максимально удобным способом. Стандартизация графического интерфейса имеет очень большое значение для пользователя, потому что одинаковый интерфейс экономит его время и упрощает изучение новых приложений. С точки зрения программиста, стандартный вид интерфейса обеспечивается использованием подпрограмм, встроенных непосредственно в Windows, что также приводит к существенной экономии времени при написании новых программ.

## 2.2. Многозадачность и многопоточность

Многозадачные операционные системы позволяют пользователю одновременно работать с несколькими приложениями или несколькими копиями одного приложения. Многозадачность осуществляется в

Windows при помощи процессов и потоков. Любое приложение Windows после запуска реализуется как процесс. Процесс можно представить как совокупность программного кода и выделенных для его исполнения системных ресурсов. При инициализации процесса система всегда создает первичный (основной) поток, который исполняет код программы, манипулируя данными в адресном пространстве процесса. Из основного потока при необходимости могут быть запущены один или несколько вторичных потоков, которые выполняются одновременно с основным потоком. На самом деле истинный параллелизм возможен только при исполнении программы на многопроцессорной компьютерной системе, когда есть возможность распределить потоки между разными процессорами. В случае обычного однопроцессорного компьютера операционная система выделяет по очереди некоторый квант времени каждому потоку.

## 2.3. Управление памятью

**Память** — это один из важнейших разделяемых ресурсов в операционной системе. Если одновременно запущены несколько приложений, то они должны разделять память, не выходя за пределы выделенного адресного пространства. Так как одни программы запускаются, а другие завершаются, то память фрагментируется. Система должна уметь объединять свободное пространство памяти, перемещая блоки кода и данных. Операционная система Windows обеспечивает достаточно большую гибкость в управлении памятью. Если объем доступной памяти меньше объема исполняемого файла, то система может загружать исполняемый файл по частям, удаляя из памяти отработавшие фрагменты. Если пользователь запустил несколько копий, которые также называют отдельными экземплярами приложения, то система размещает в памяти только одну копию исполняемого кода, которая используется этими экземпля-

рами совместно. Программы, запущенные в Windows, могут использовать также функции из библиотек динамической компоновки — **DLL (dynamic link libraries)**. Windows поддерживает механизм связи программ во время их работы с функциями из DLL. Даже сама операционная система Windows, по существу, является набором динамически подключаемых библиотек. Эти библиотеки содержат набор функций **Win API (Application Programming Interface - интерфейс прикладного программирования)**, позволяющих программисту создавать приложения, работающие под управлением Windows.

## 2.4. Независимость от аппаратных средств

Еще одним преимуществом Windows является независимость от используемой платформы. У программ, написанных для Windows, нет прямого доступа к аппаратной части таких устройств отображения информации, как, например, экран и принтер. Вместо этого они вызывают функции графической подсистемы Win API, называемой **графическим интерфейсом устройства** (**Graphics Device Interface, GDI**). Функции GDI реализуют основные графические команды при помощи обращения к программным драйверам соответствующих аппаратных устройств. Одна и та же команда (например, LineTo — нарисовать линию) может иметь различную реализацию в разных драйверах. Эта реализация скрыта от программиста, использующего Win API, что упрощает разработку приложений. Таким образом, приложения, написанные с использованием Win API, будут работать с любым типом дисплея и любым типом принтера, для которых имеется в наличии драйвер Windows. То же самое относится и к устройствам ввода данных — клавиатуре, манипулятору «мышь» и т.д. Такая независимость Windows от аппаратных средств достигается благодаря указанию требований, которым должна удовлетво-

рять аппаратура, в совокупности с **SDK** (**Software Development Kit** — набор разработки программ) и/или **DDK** (**Driver Development Kit** — набор разработки драйверов устройств). Разработчики нового оборудования поставляют его вместе с программными драйверами, которые обязаны удовлетворять этим требованиям.

Прежде чем перейти к изучению архитектуры Windows-приложений, вкратце рассмотрим структуру и способ исполнения консольных приложений, которые разрабатывались до этого момента при изучении языка C++. Отметим, что **все консольные приложения имеют архитектуру DOS-программ (Disk Operating System). В таких программах применяется процедурный подход к программированию, при котором программа выполняется последовательно от начала (от первой строки функции main) до конца в предопределенном порядке**. Наиболее часто выполняемые блоки программы выделяются в подпрограммы (функции). В такой архитектуре взаимодействие между системой и программой инициирует сама программа. Например, программа запрашивает разрешение на ввод и вывод данных. Таким образом, консольные приложения, написанные с использованием подобной архитектуры DOS-программ, при необходимости сами обращаются к операционной системе. Операционная система никогда не вызывает прикладную программу.

## 3. Событие. Сообщение. Очередь сообщений

Взаимодействие Windows-приложения с внешним миром и операционной системой строится на основе событий и сообщений.

**Событие – это действие, инициированное пользователем, либо операционной системой, либо приложением**. Любому событию соответствует сообщение, которое однозначно идентифицирует произошедшее событие. **Сообщение - это уведомление о том, что**

**произошло некоторое событие.** Событие может быть следствием действий пользователя (например, перемещение курсора мыши, щелчок кнопкой мыши, изменение размеров окна, выбор пункта меню и т.д.). Иногда одно событие влечет за собой еще несколько событий и сообщений. Например, событие создания окна влечет за собой событие перерисовки окна, активизации окна, а также событие создания окна для дочерних окон (кнопок, полей ввода) и так далее.

Сообщение – это уникальная целочисленная константа. Для удобства программирования в программе вместо целочисленных номеров используются макроопределения. Например:

```
#define WM_PAINT 0x000F
```

Весь список мнемонических имен сообщений определен в файле **winuser.h.**

Сообщения от внешних источников, например от клавиатуры, адресуются в каждый конкретный момент времени только одному из работающих приложений, а именно — активному окну. При этом Windows играет роль диспетчера сообщений. С момента старта операционная система создает в памяти глобальный объект, называемый **системной очередью сообщений**. Все сообщения, генерируемые как аппаратурой, так и приложениями, помещаются в эту очередь. Windows периодически опрашивает эту очередь и, если она не пуста, посылает очередное сообщение нужному адресату (окну).

Сообщения, получаемые приложением, могут поступать асинхронно из разных источников. Например, приложение может работать с системным таймером, посылающим ему сообщения с заданным интервалом, и одновременно оно должно быть готовым в любой момент получить любое сообщение от операционной системы. Чтобы не допустить потери сообщений, Windows одновременно с запуском приложения создает глобаль-

ный объект, называемый **очередью сообщений приложения**. Время жизни этого объекта совпадает со временем жизни приложения.

## 4. Окно

**Окно – это некоторый объект, обладающий набором свойств и занимающий определенную область оперативной памяти.**

Для любого окна Windows выделяет **дескриптор (handle)**, который уникально идентифицирует окно в пределах системы. Именно при помощи дескриптора Windows определяет какому окну нужно отправить сообщение.

У любого окна есть **оконная процедура**, которая получает все сообщения, предназначенные окну. В ней программист может выполнить обработку поступающих сообщений. После того как сообщение получено, оконная процедура ищет функцию - обработчик данного сообщения, и если он определен в коде программы, то выполняется его тело, иначе вызывается стандартный Windows-обработчик (стандартная функция обработки этого сообщения). После завершения обработки сообщения программа ожидает следующего. Windows может посылать программе сообщения самых различных типов. Таким образом, программирование под Windows, в сущности, сводится к обработке сообщений.

**Важно отметить: сообщения поступают в программу неупорядоченным образом, т.е. невозможно предугадать какое сообщение придет в следующий момент времени. Отсюда следует, что нельзя заранее знать какой фрагмент кода оконной процедуры будет выполняться в конкретный момент времени. Это обуславливает нелинейный способ выполнения программы.**

**Оконный класс (window class), или класс окна** — это структура, определяющая основные характеристики окна. К ним относятся стиль окна и связанные с окном ресурсы, такие как пиктограмма, курсор, меню

и кисть для закрашивания фона. Кроме того, одно из полей структуры содержит адрес оконной процедуры, предназначенной для обработки сообщений, получаемых любым окном данного класса.

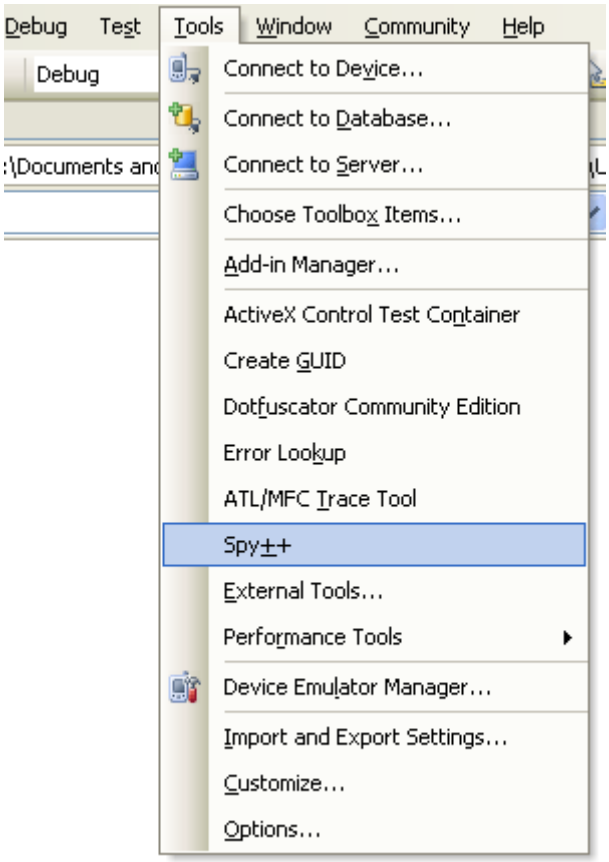
Использование класса окна позволяет создавать множество окон на основе одного и того же оконного класса и, следовательно, использовать одну и ту же оконную процедуру. Например, все кнопки в программах Windows созданы на основе оконного класса **BUTTON**. Оконная процедура этого класса, расположенная в динамически подключаемой библиотеке, управляет обработкой сообщений для всех кнопок всех окон. Аналогичные системные классы имеются и для других элементов управления (например, списки и текстовые поля ввода), которые представляют собой частный случай окна. В совокупности эти классы называются предопределенными или стандартными оконными классами.

Windows содержит предопределенный оконный класс также и для диалоговых окон, играющих важную роль в графическом интерфейсе пользователя.

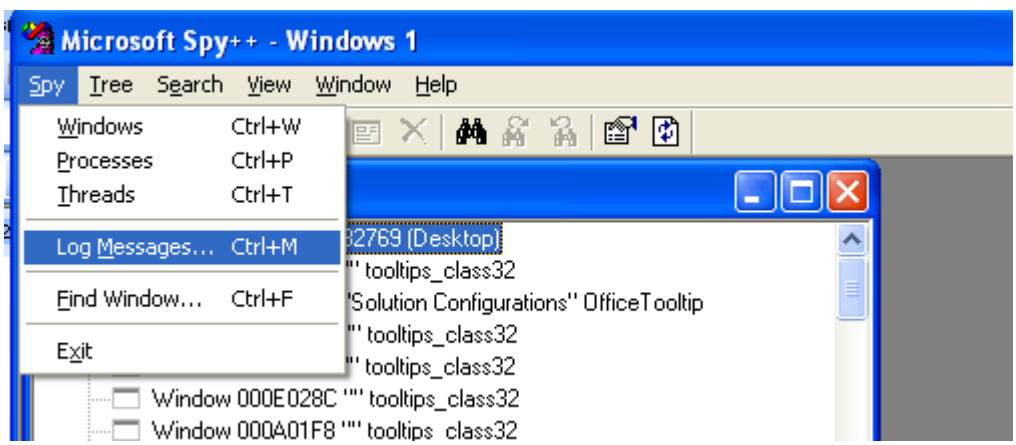
Для главного окна приложения обычно создается собственный класс окна, учитывающий индивидуальные требования к программе.

## 5. Утилита Spy++

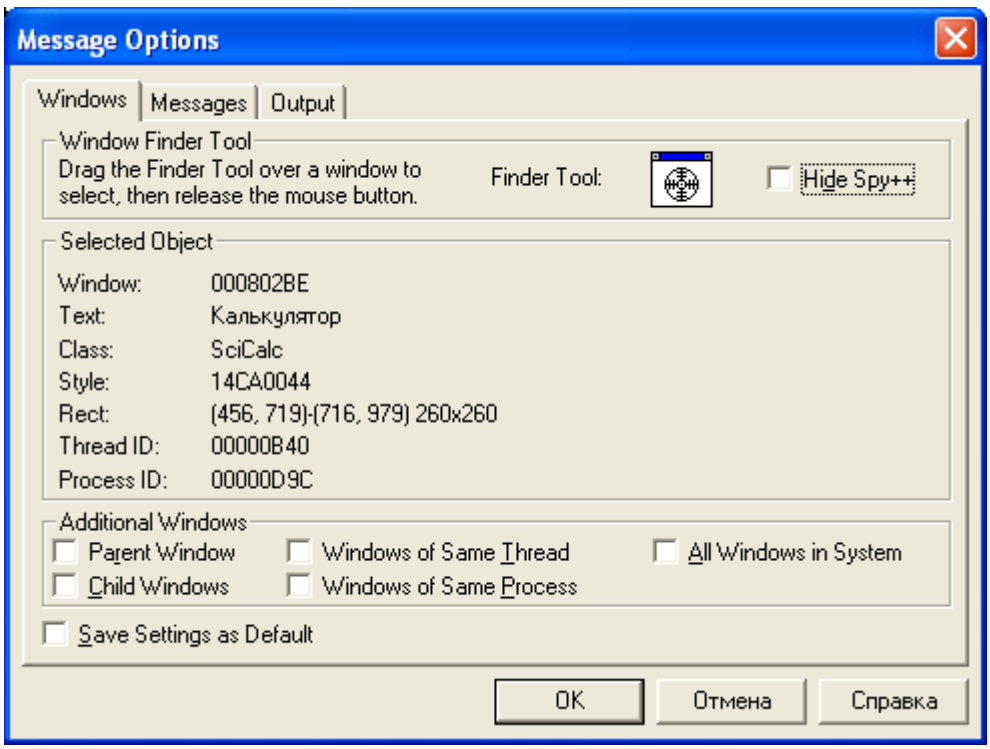
В процессе отладки приложений иногда бывает полезным увидеть, какие сообщения вырабатывает Windows в ответ на те или иные действия пользователя. В составе интегрированной среды Microsoft Visual Studio есть удобное инструментальное средство для решения данной проблемы — утилита **Spy++**. Эта программа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




Рассмотрим 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.



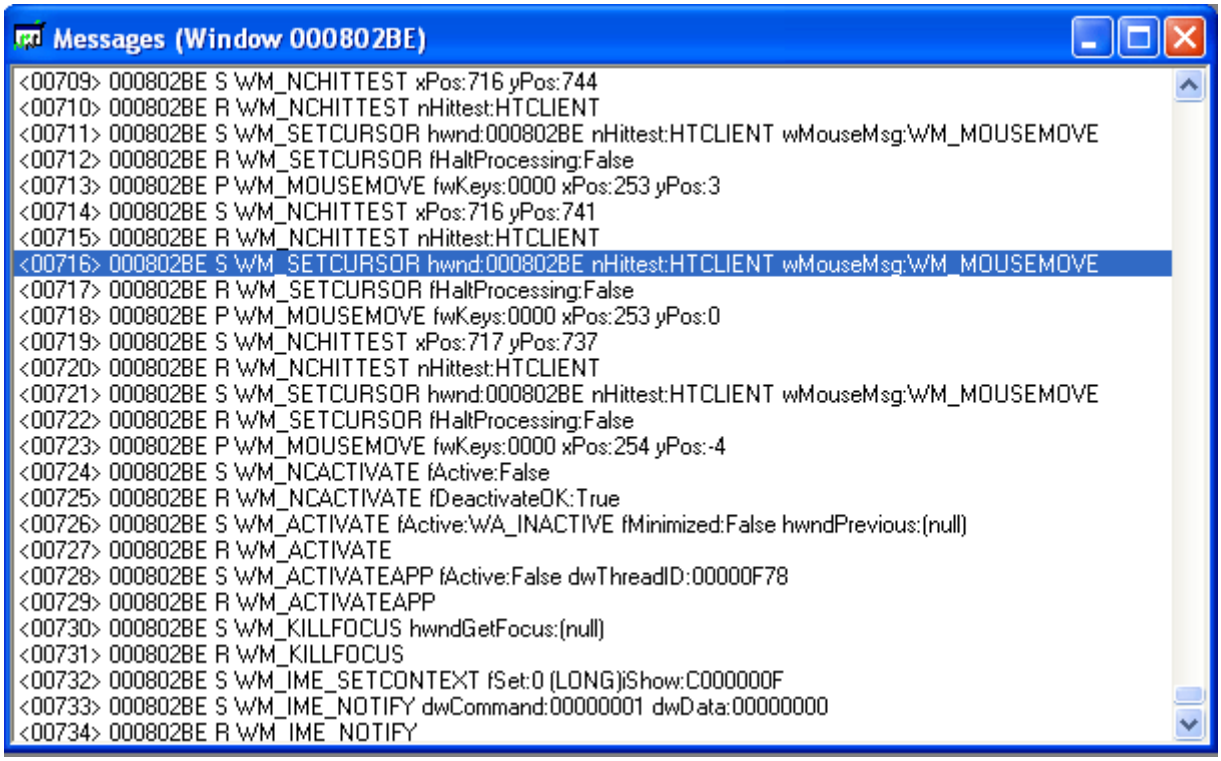
Далее выполнить команду меню **Spy -> Log Messages**. В результате будет отображено диалоговое окно **Message Options**.




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.

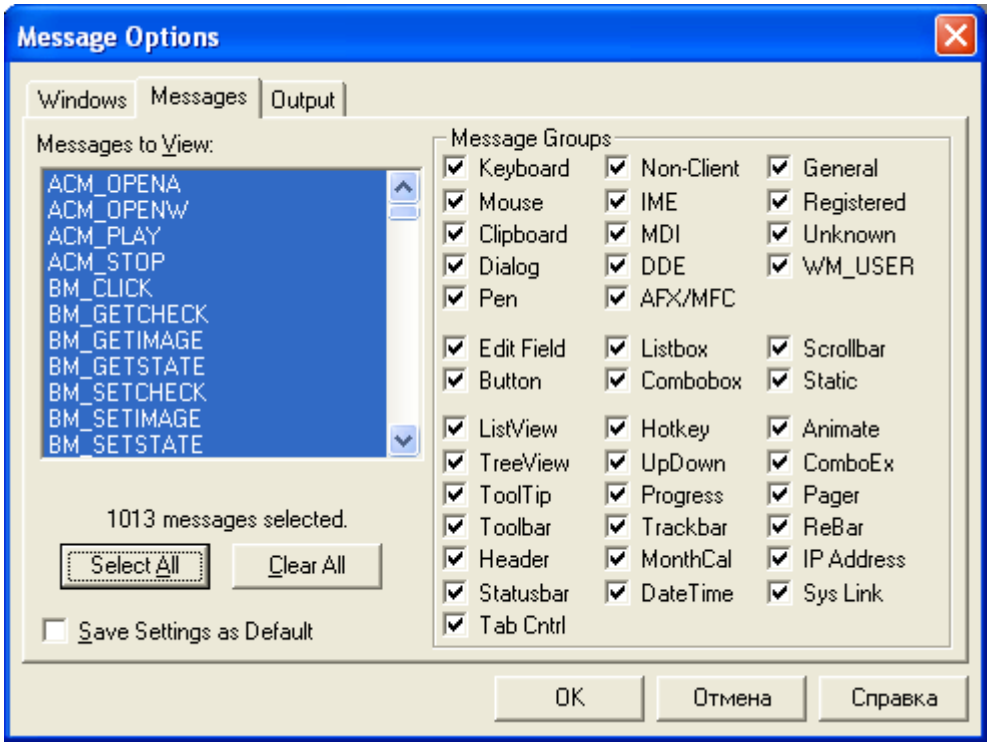



При этом каждое сообщение в окне Messages отображается в следующем формате:

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.




## 6. «Новые» типы данных

Каждый тип данных Windows является синонимом уже существующего типа языка C++. Например:

```
typedef int  BOOL
```

| 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. |

| Тип данных | Описание |
|---|---|
| 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. |

## 7. Венгерская нотация

При разработке приложений было бы удобно идентифицировать переменные так, чтобы по имени переменной можно было определить её назначение в программе, а также тип данных. Для решения этой про-

блемы 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).

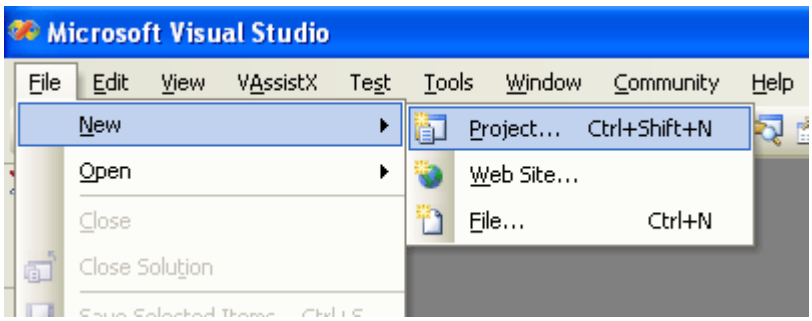
Большинство функций WinAPI используют венгерскую нотацию, поэтому, по меньшей мере, знать о ней необходимо. Использование венгерской нотации желательно для написания понятного и читабельного кода.

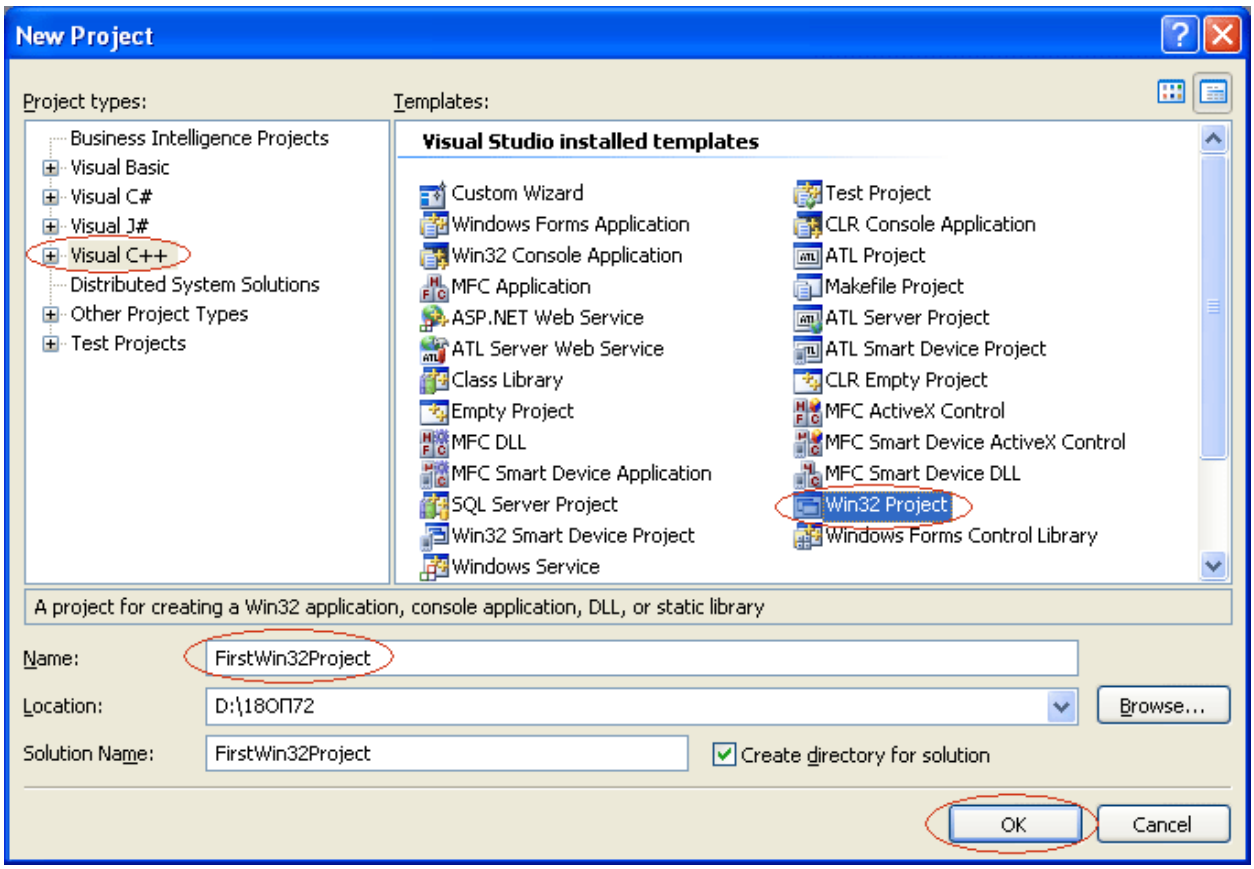
В венгерской нотации предлагаются следующие префиксы:

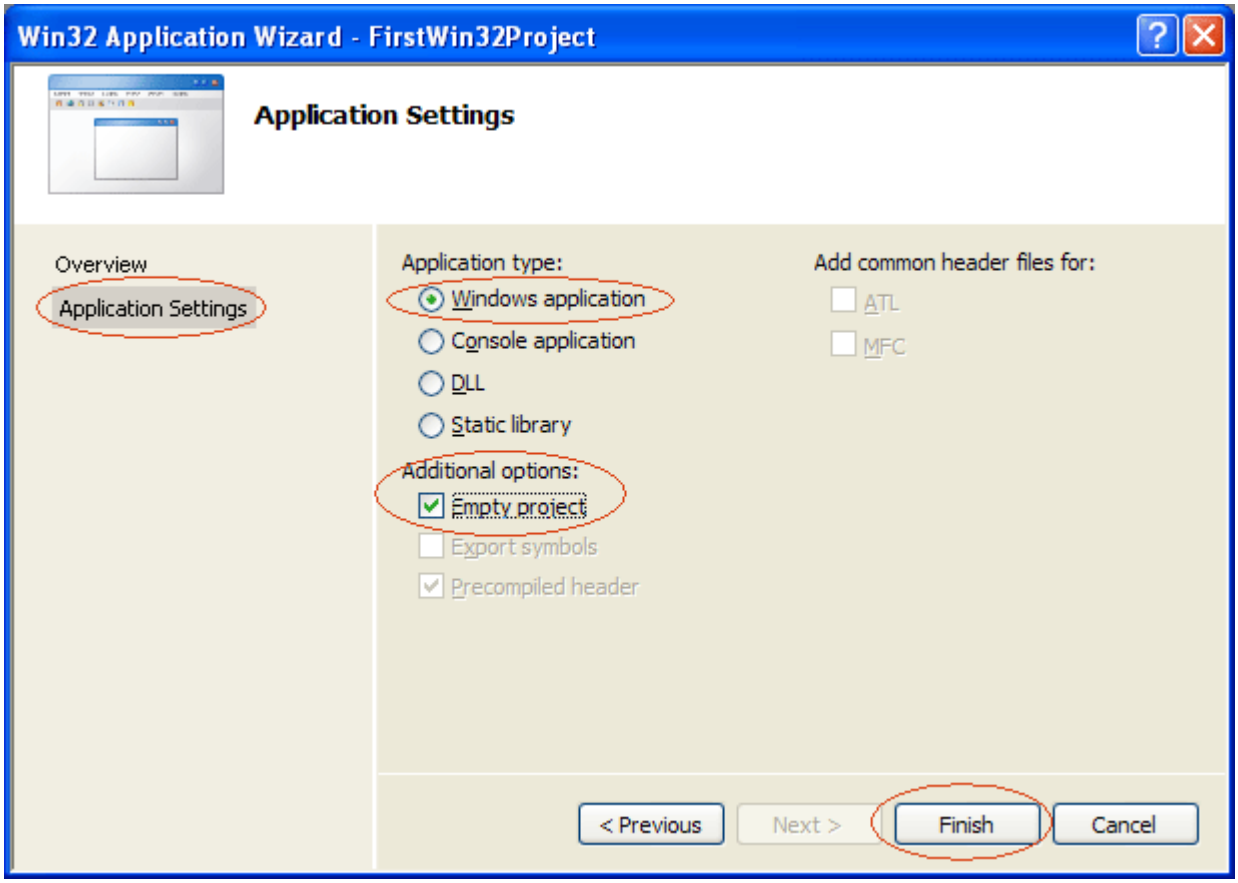
| Префикс | Тип переменной |
|---|---|
| b | Логический тип (bool или BOOL) |
| i | Целое число (индекс) |
| n | Целое число (количество чего-либо) |
| u | Целое число без знака |
| d | Число с двойной точностью |
| sz | Строковая переменная, ограниченная нулем |
| p | Указатель |
| lp | Длинный указатель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

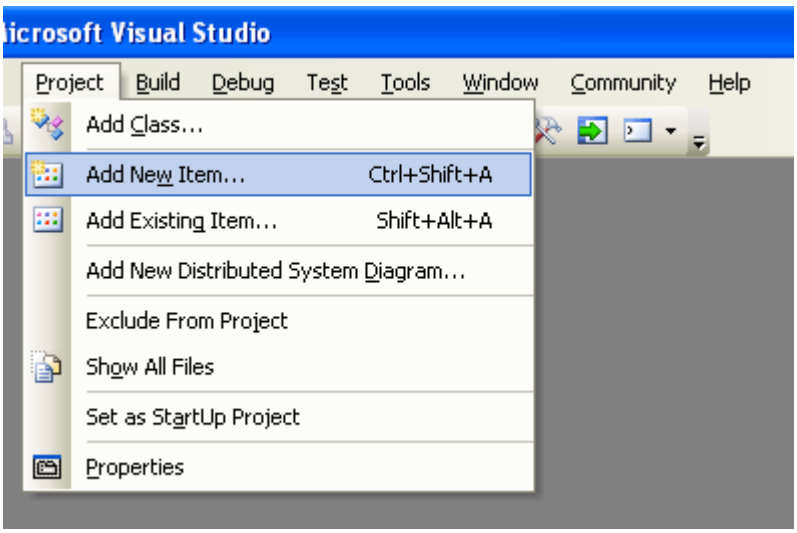
# 8. Этапы создания проекта Win32 project

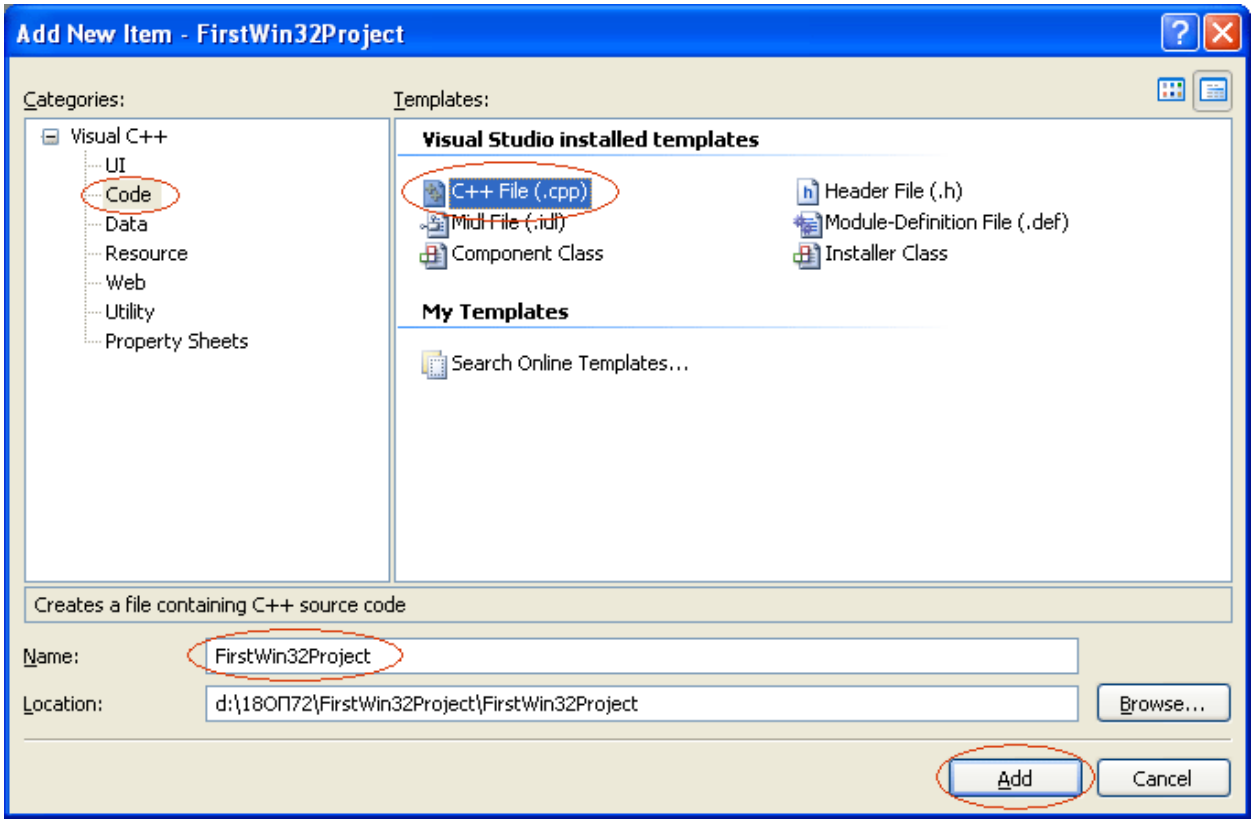
Рассмотрим этапы формирования проекта Win32 Project.




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


icrosoft 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




## 9. Минимальное Win32-приложение

Минимальное Win32-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 WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpszCmdLine,
int nCmdShow)
{
      HWND hWnd;
      MSG lpMsg;
      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(&lpMsg, NULL, 0, 0))
      {
            TranslateMessage(&lpMsg);     // трансляция сообщения
            DispatchMessage(&lpMsg);      // диспетчеризация сообщений
      }
      return lpMsg.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;
}
```

Проанализируем вышеприведенное минимальное Win32 - приложение (исходный код находится в папке **SOURCE/ FirstWin32Application**).

Заголовочный файл **Windows.h** содержит определения, макросы, и структуры, которые используются при написании приложений под Windows. Таким образом, при программировании Windows-приложений с использованием Win API данный файл следует подключать всегда.

Как отмечалось выше, функция **WinMain** является точкой входа в приложение и имеет следующий прототип:

```
INT WINAPI WinMain(
      HINSTANCE hInst, // дескриптор экземпляра приложения
      HINSTANCE hPrevInst, //равен 0 и необходим для совместимости
      LPSTR 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)** – 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).

## 9.1. 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
LPCSTR lpszName //строка, содержащая имя иконки
);
```

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

| Макрос | Форма иконки |
|---|---|
| IDI_APPLICATION | Стандартная иконка для приложения |
| IDI_ASTERISK | Иконка "информация" |
| IDI_EXCLAMATION | Иконка "восклицательный знак" |
| 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
LPCSTR lpszName //строка, содержащая имя курсора
);
```

Для того чтобы использовать встроенный тип курсора Windows, первый параметр должен быть равен NULL, а в качестве второго параметра может 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
);
```

## 9.2. 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
);
```

## 9.3. 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
LPCSTR lpClassName, // имя класса окна
LPCSTR 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
    HINSTAHCE 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).

## 9.4. 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 );
```

## 9.5. 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 msg, //адрес структуры MSG, в которую помещается выбранное сообщение
HWND hwnd, // дескриптор окна, принимающего сообщение
/* Обычно значение этого параметра равно NULL, что позволяет выбрать сообще-
ния 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.

## 9.6. 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);
```

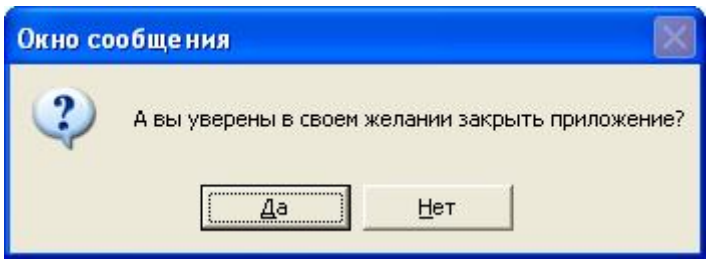
Четыре параметра оконной процедуры идентичны первым четырем полям структуры **MSG**. Первый параметр функции содержит дескриптор окна, получающего сообщение. Во втором параметре указывается идентификатор сообщения. Для системных сообщений зарезервированы номера от 0 до 1024. Третий и четвертый параметры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

## 10. 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, 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 информационное окно, необходимо комбинировать указанные значения параметра **uType** со следующими значениями:

| Значение параметра uType | Описание |
|---|---|
| MB_ICONEXCLAMATION | Отображается иконка «восклицательный знак» |
| MB_ICONINFORMATION (или MB_ICONASTERISK) | 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 |

## 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 сообще-

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