**1. UNICODE**

## 1.1. Понятие кодировки. Unicode кодировка

**Кодировка – это определённый набор символов (**[**англ.**](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***character set*), в котором каждому символу сопоставляется последовательность длиной в один или несколько** [**байтов**](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82)**.**

В кодировке **ANSI** ([American National Standards Institute](http://webstore.ansi.org/)) 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 С++ (С/С++ Run-Time Library)

В с++ был введен «новый» тип данных, специально для использования Unicode-строк:

|  |
| --- |
| wchar\_t — тип данных для Unicode-символа |

Для работы с Unicode-строками существует набор Unicode-функций, которые эквивалентны строковым функциям ANSI C:

| **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**.

Пример:

|  |
| --- |
| #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. **Макросы для работы с Unicode**

Есть 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  } |

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 х – строка символов.

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

* 1. **Строковые функции Windows**

Windows так же предоставляет набор функций для работы со строками. Эти функции похожи на строковые функции из библиотеки С/С++, например на **strcpy** и **wcscpy**. Однако функции Windows являются частью операционной системы, и многие её компоненты используют именно их, а не аналоги из библиотеки С/С++. 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 |
| lstrcpy | Копирует строку в другой участок памяти |
| lstrlen | Возвращает длину строки в символах |

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. **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  ); |

Пример:

|  |
| --- |
| #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;  } |

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 С/С++ 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*  *);* |

Пример:

|  |
| --- |
| #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. **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)  ); |

Пример:

|  |
| --- |
| #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;  } |

Аналогичные преобразования выполняет функция из С/С++ 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  ); |

Пример:

|  |
| --- |
| #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;  } |

1. **Разработка 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.

1. **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>  LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);  TCHAR szClassWindow[] = TEXT("Каркасное приложение");  INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpszCmdLine, int nCmdShow)  {  HWND hWnd;  MSG lpMsg;  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("Каркас Windows приложения"),  WS\_OVERLAPPEDWINDOW,  CW\_USEDEFAULT,  CW\_USEDEFAULT,  CW\_USEDEFAULT,  CW\_USEDEFAULT,  NULL,  NULL,  hInst,  NULL);  ShowWindow(hWnd, nCmdShow);  UpdateWindow(hWnd);    while(GetMessage(&lpMsg, NULL, 0, 0))  {  TranslateMessage(&lpMsg);  DispatchMessage(&lpMsg);  }  return lpMsg.wParam;  }  LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMessage, WPARAM wParam,  LPARAM lParam)  {  switch(uMessage)  {  case WM\_DESTROY:  PostQuitMessage(0);  break;  default:  return DefWindowProc(hWnd, message, wParam, lParam);  }  return 0;  } |

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 С/С++. Это связано с тем, что в языке С 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).
   1. **«Новые» типы данных**

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

|  |
| --- |
| typedef int BOOL |

Таблица 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. |

* 1. **Венгерская нотация**

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).

Большинство функций WinAPI используют венгерскую нотацию, поэтому, по меньшей мере, знать о ней необходимо. 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 |

* 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, a 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" |
| 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, a 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  *);* |

* 1. **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  ); |

* 1. **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*, //* расширенныйстиль окна  LPCSTR lpClassName*, //* имя класса окна  LPCSTR lpWinName*, //* заголовок окна  DWORD dwStyle*, //* стиль окна  int x*,* int у, // 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, у, 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).

* 1. **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 ); |

* 1. **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 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, вызывая ее как **функцию обратного вызова.**

* 1. **Оконная процедура**

Оконная процедура вызывается операционной системой и 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.

1. **Принципы обработки сообщений мыши**

Оконная процедура получает сообщения мыши в случае, если мышь проходит через окно, а также при щелчке внутри окна, даже если окно не активно или не имеет фокуса ввода. Если мышь перемещается по клиентской области окна, то оконная процедура получает сообщение **WM\_MOUSEMOVE**. Если кнопка мыши нажимается или отпускается внутри клиентской области, то оконная процедура получает следующие сообщения:

* **WM\_LBUTTONDOWN** – нажата левая кнопка мыши;
* **WM\_MBUTTONDOWN** - нажата средняя кнопка мыши;
* **WM\_RBUTTONDOWN** – нажата правая кнопка мыши;
* **WM\_LBUTTONUP** – отпущена левая кнопка мыши;
* **WM\_MBUTTONUP** – отпущена средняя кнопка мыши;
* **WM\_RBUTTONUP** – отпущена правая кнопка мыши.
* **WM\_LBUTTONDBLCLK** – двойной щелчок левой кнопкой мыши;
* **WM\_MBUTTONDBLCLK** – двойной щелчок средней кнопкой мыши;
* **WM\_RBUTTONDBLCLK** - двойной щелчок правой кнопкой мыши.

Прокрутка колесика вызывает сообщение **WM\_MOUSEWHEEL**.

Для всех этих сообщений значение параметра **lParam** содержит положение мыши. При этом в младшем слове находится значение координаты **х**, а в старшем слове — значение координаты **у**. **Отсчет координат ведется от левого верхнего угла клиентской области окна**. Эти значения можно извлечь из **lParam** при помощи макросов **GET\_X\_LPARAM** и **GET\_Y\_LPARAM** из библиотеки windowsx.h.

Следует особо подчеркнуть, что окно будет получать сообщения о двойном щелчке (**DBLCLK**) только в том случае, если стиль соответствующего класса окна содержит флаг **CS\_DBLCLKS**. Поэтому перед регистрацией класса окна нужно присвоить полю **style** структуры **WNDCLASSEX** значение, включающее этот флаг. Если класс окна определен без флага **CS\_DBLCLKS** и пользователь делает двойной щелчок левой кнопкой мыши, то оконная процедура последовательно получает сообщения **WM\_LBUTTONDOWN**, **WM\_LBUTTONUP**, **WM\_LBUTTONDOWN** и **WM\_LBUTTONUP**. Если класс окна определен с флагом **CS\_DBLCLKS**, то после двойного щелчка оконная процедура получит сообщения **WM\_LBUTTONDOWN**, **WM\_LBUTTONUP**, **WM\_LBUTTONDBLCLK** и **WM\_LBUTTONUP**.

Модифицировать стиль класса окна можно также вызовом следующих инструкций:

|  |
| --- |
| UINT style = GetClassLong(hWnd, GCL\_STYLE);  SetClassLong(hWnd, GCL\_STYLE, style | CS\_DBLCLKS); |

1. **Принципы обработки нажатия клавиш**

Одно из широко используемых сообщений порождается при нажатии клавиши. Это сообщение называется **WM\_CHAR**. Для сообщений **WM\_CHAR** параметр **wParam** содержит ASCII-код нажатой клавиши. **LOWORD** (**lParam**) содержит количество повторов, генерируемых при удерживании клавиши в нажатом положении. **HIWORD** (**lParam**) представляет собой битовую карту со следующими значениями битов:

* 15: равен 1, если клавиша отпущена, и 0, если она нажата.
* 14: устанавливается, если клавиша уже была нажата перед посылкой сообщения.
* 13: устанавливается в 1, если дополнительно нажата клавиша <**Alt**>.
* 12-9: используется системой.
* 8: устанавливается в 1, если нажата клавиша функциональной или дополнительной части клавиатуры.
* 7-0: код клавиши (scan-код).

Символьные сообщения **WM\_CHAR** передаются в оконную процедуру в промежутке между аппаратными сообщениями клавиатуры. Например, если пользователь, удерживая клавишу <**Shift**>, нажимает клавишу <**А**>, отпускает клавишу <**А**> и затем отпускает клавишу <**Shift**>, то оконная процедура получит пять сообщений:

| **Сообщение** | **Виртуальная клавиша или ANSI-код** |
| --- | --- |
| WM\_KEYDOWN | Виртуальная клавиша VK\_SHIFT |
| WM\_KEYDOWN | Виртуальная клавиша А |
| WM\_CHAR | ANSI-код символа А |
| WM\_KEYUP | Виртуальная клавиша А |
| WM\_KEYUP | Виртуальная клавиша VK\_SHIFT |

Имеет смысл обрабатывать только те аппаратные сообщения **WM\_KEYDOWN** и **WM\_KEYUP**, которые содержат (в **wParam**) виртуальные коды для клавиш управления курсором, клавиш <**Shift**>,<**Ctrl**>,<**Alt**> функциональных клавиш (**VK\_LEFT, VK\_DOWN, VK\_SHIFT, VK\_CTRL, VK\_MENU, VK\_RETURN, VK\_TAB** и т.д.). В то же время аппаратные сообщения для символьных клавиш могут игнорироваться. Ввод информации от символьных клавиш гораздо удобнее обрабатывать, используя символьное сообщение **WM\_CHAR**.

Получить состояние указанной виртуальной клавиши можно с помощью функции API **GetKeyState**. Это состояние показывает, нажата ли клавиша, отпущена или переключена в то или иное состояние.

|  |
| --- |
| SHORT GetKeyState (int nVirtKey); |

Параметр **nVirtKey** задаёт код виртуальной клавиши. Возвращаемое значение – состояние клавиши, которое закодировано в двух битах. Если старший бит равен 1, то клавиша нажата, в ином случае, она отпущена. Если младший бит равен 1, то клавиша переключена, т.е. переведена во включенное состояние.

Полный перечень всех макросов виртуальных клавиш представлен в файле **winuser.h.**

Пример:

|  |
| --- |
| #include <windows.h>  #include <tchar.h>  LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);  TCHAR szClassWindow[] = TEXT("Каркасное приложение");  INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpszCmdLine, int nCmdShow)  {  HWND hWnd;  MSG lpMsg;  WNDCLASSEX wcl;    wcl.cbSize = sizeof (wcl);  wcl.style = CS\_HREDRAW | CS\_VREDRAW | CS\_DBLCLKS; // окно сможет  // получать сообщения о двойном щелчке (DBLCLK)    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("Каркас Windows приложения"),    WS\_OVERLAPPEDWINDOW,  CW\_USEDEFAULT,  CW\_USEDEFAULT,  CW\_USEDEFAULT,  CW\_USEDEFAULT,  NULL,  NULL,  hInst,  NULL);  ShowWindow(hWnd, nCmdShow);  UpdateWindow(hWnd);    while (GetMessage(&lpMsg, NULL, 0, 0))  {  TranslateMessage(&lpMsg);  DispatchMessage(&lpMsg);  }  return lpMsg.wParam;  }  LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMessage, WPARAM wParam,  LPARAM lParam)  {  TCHAR str[50];  switch(uMessage)  {  case WM\_LBUTTONDBLCLK:  MessageBox(  0,  TEXT("Двойной щелчок левой кнопкой мыши"),  TEXT("WM\_LBUTTONDBLCLK"),  MB\_OK | MB\_ICONINFORMATION);  break;  case WM\_LBUTTONDOWN:  MessageBox(  0,  TEXT("Нажата левая кнопка мыши"),  TEXT("WM\_LBUTTONDOWN"),  MB\_OK | MB\_ICONINFORMATION);  break;  case WM\_LBUTTONUP:  MessageBox(  0,  TEXT("Отпущена левая кнопка мыши"),  TEXT("WM\_LBUTTONUP"),  MB\_OK | MB\_ICONINFORMATION);  break;  case WM\_RBUTTONDOWN:  MessageBox(  0,  TEXT("Нажата правая кнопка мыши"),  TEXT("WM\_RBUTTONDOWN"),  MB\_OK | MB\_ICONINFORMATION);  break;  case WM\_MOUSEMOVE:  // текущие координаты курсора мыши  wsprintf(str, TEXT("X=%d Y=%d"), LOWORD(lParam), HIWORD(lParam));  SetWindowText(hWnd, str);//строка выводится в заголовок окна  break;  case WM\_CHAR:  wsprintf(str, TEXT("Нажата клавиша %c"),  (char) wParam); // ASCII-код нажатой клавиши  MessageBox(0, str, TEXT("WM\_CHAR"),   MB\_OK | MB\_ICONINFORMATION);  break;  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, uMessage, wParam, lParam);  }  return 0;  } |

При обработке сообщения **WM\_MOUSEMOVE** с помощью функции **wsprintf** форматируется строка, содержащая текущие координаты мыши, для последующего вывода в заголовок окна. Вывод строки в заголовок окна осуществляется функцией API **SetWindowText**:

|  |
| --- |
| BOOL SetWindowText (  HWND hWnd, // дескриптор окна, в котором должен быть изменен текст  LPCTSTR lpString //указатель на строку, содержащую новый текст  ); |

В WinApi есть группа функций, позволяющих получать информацию о размерах и расположении окна, а также изменять размеры, расположение и характеристики отображения окна.

Функция API **GetWindowRect** позволяет получить размеры прямоугольника окна:

|  |
| --- |
| BOOL GetWindowRect(  HWND hWnd, //дескриптор окна  LPRECT lpRect //указатель на структуру RECT  ); |
| typedef struct tagRECT {  LONG left;  LONG top;  LONG right;  LONG bottom;  } RECT; |

Поля этой структуры задают координаты левого верхнего угла (**left, top**) и правого нижнего угла (**right, bottom**) прямоугольника.

В структуре **RECT** будут указаны экранные координаты левого верхнего и правого нижнего углов окна. Отсчёт координат ведётся относительно левого верхнего угла экрана (0,0).

Функция API **GetClientRect** позволяет получить размеры прямоугольника, охватывающего клиентскую (рабочую) область окна:

|  |
| --- |
| BOOL GetClientRect(  HWND hWnd, //дескриптор окна  LPRECT lpRect //указатель на структуру RECT  ); |

В структуре **RECT** будут указаны координаты левого верхнего и правого нижнего углов клиентской области окна. Поскольку отсчёт координат в данном случае ведётся относительно левого верхнего угла рабочей области окна, то координаты (**left, top**) будут равны (0,0).

Функция API **MoveWindow** позволяет переместить окно, а также изменить его размеры:

|  |
| --- |
| BOOL MoveWindow(  HWND hWnd,//дескриптор окна  int X, //новая координата Х левого верхнего угла окна  int Y, //новая координата Y левого верхнего угла окна  int nWidth, //новая ширина окна  int nHeight, //новая высота окна  BOOL bRepaint //необходимость немедленной перерисовки окна  ); |

Функция API **BringWindowToTop** активизирует окно и переносит его в верхнее положение, если оно находится позади других окон:

|  |
| --- |
| BOOL BringWindowToTop(  HWND hWnd //дескриптор окна  ); |