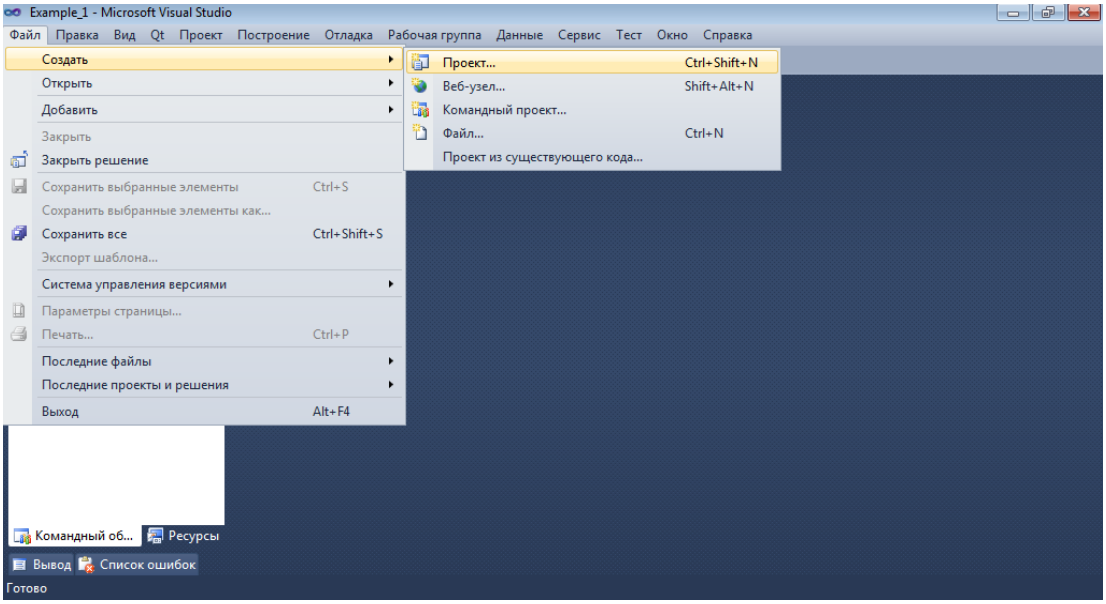
Win32 API (далее WinAPI) – это набор функций (*API – Application Programming Interface*), работающих под управлением ОС Windows. Они содержатся в библиотеке `windows.h`.

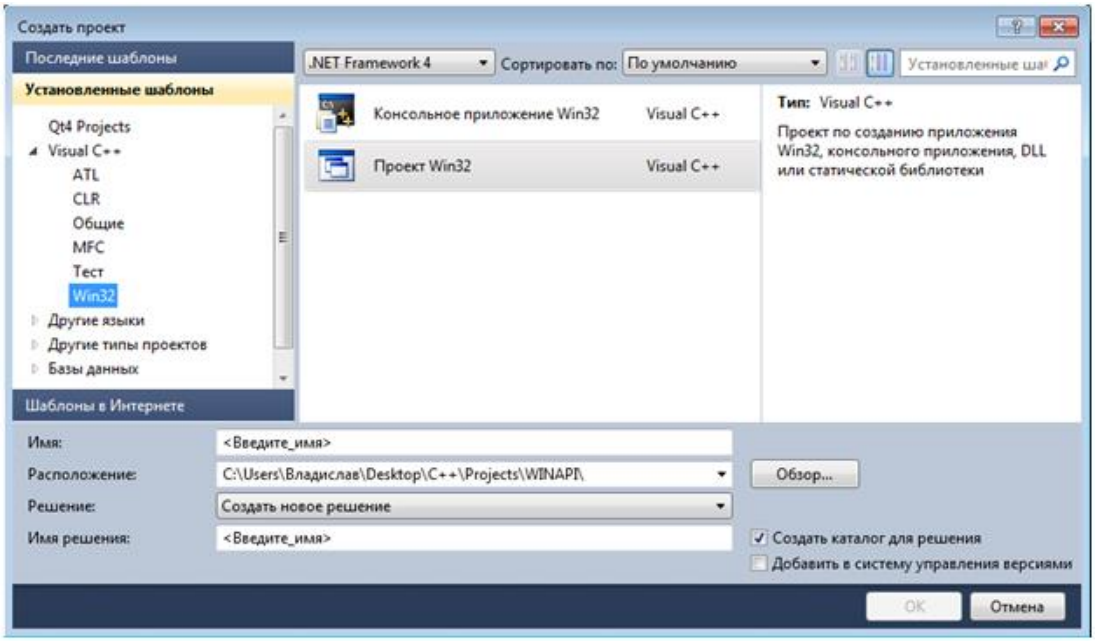
С помощью WinAPI можно создавать различные оконные процедуры, диалоговые окна, программы и даже игры. Эта, скажем так, библиотека является базовой в освоении программирования Windows Forms, MFC, потому что эти интерфейсы являются надстройками этой библиотеки. Освоив её, Вы без труда будете создавать формы, и понимать, как это происходит.

Не будем внедряться в теорию. Начнём с того, как создать этот проект в MVS, а в конце статьи будет разобран простой пример.

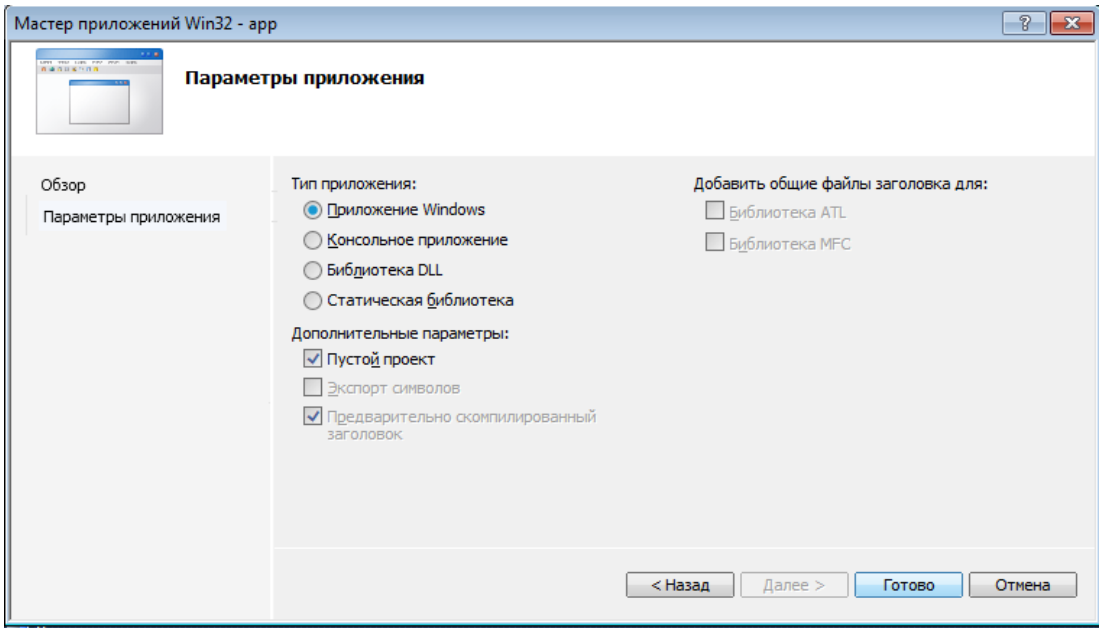
Итак. Сначала открываем Visual Studio, затем, нажимаем на вкладку «Файл», далее «Создать проект»:



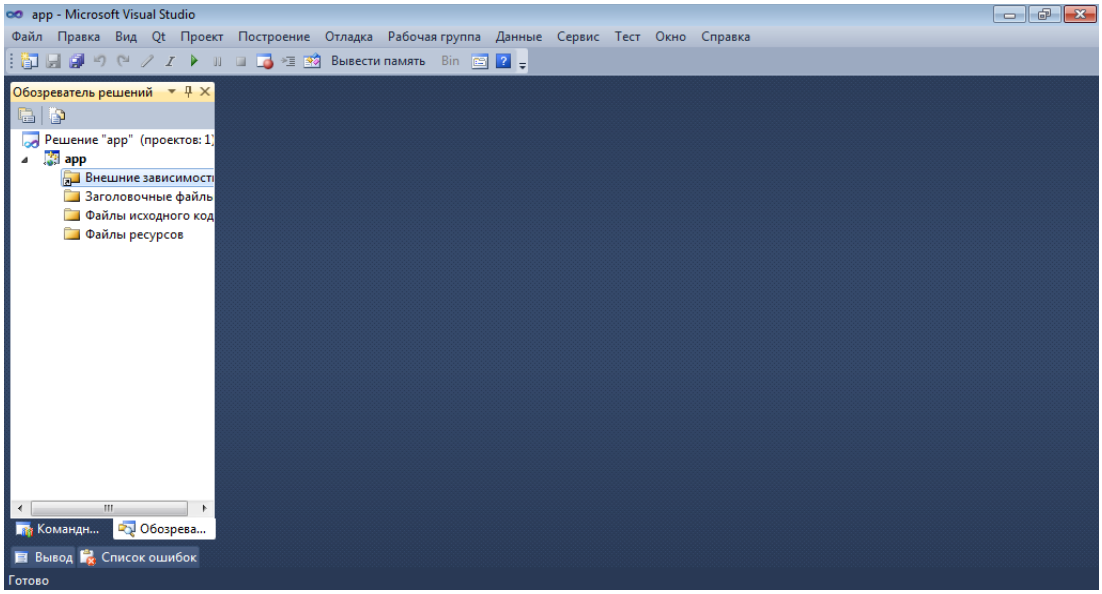
Затем, в раскрывающемся списке Visual C++ выбираем пункт Win32, там и будет «Проект Win32». Щелкаем по нему:



Вводим название проекта, указываем путь и нажимаем «ОК». Далее будет написано: «Добро пожаловать в мастер приложения Win32». Нажимаем далее. По-умолчанию у надписи «Пустой проект» галочка отсутствует. Нам нужно её поставить и убедиться, что у нас «Тип Приложения» — Приложение Windows. Если всё верно, нажимаем – «Готово».

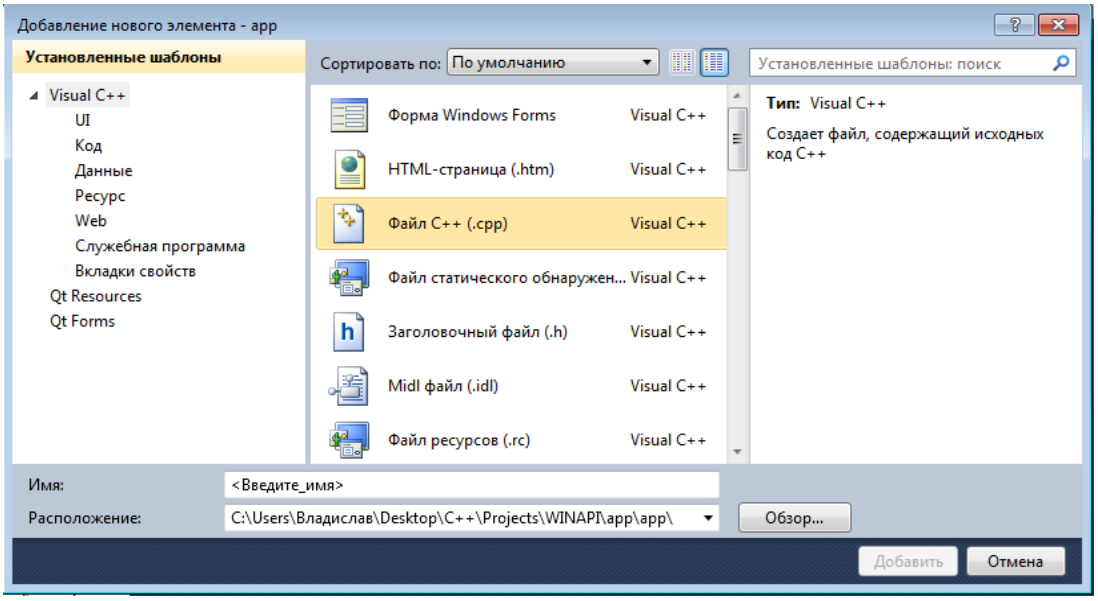


У нас должен быть пустой проект такого вида:



Ну а теперь начнём писать простую программу, которая традиционно будет выводить на экран надпись: «Привет, Мир!!!».

Естественно, к проекту нужно добавить файл типа «имя».cpp. Кликаем по «Файлы исходного кода» правой кнопкой мыши, в раскрывающемся списке выбираем вкладку – «Добавить», далее «Создать элемент…». В результате у нас должно появиться такое окно:

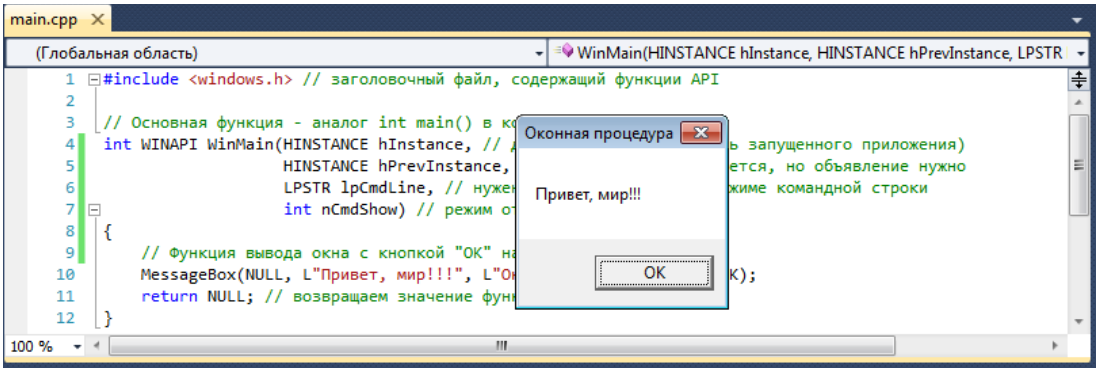


Выбираем «Файл C++», вводим имя, нажимаем «Добавить». Затем открываем этот файл и вставляем в него такой код (подробности далее):

```
#include <windows.h> // заголовочный файл, содержащий функции API

// Основная функция - аналог int main() в консольном приложении:
int WINAPI WinMain(HINSTANCE hInstance, // дескриптор экземпляра приложения
                   HINSTANCE hPrevInstance, // в Win32 не используется
                   LPSTR lpCmdLine, // нужен для запуска окна в режиме
командной строки
                   int nCmdShow) // режим отображения окна
{
        // Функция вывода окна с кнопкой "OK" на экран (о параметрах позже)
```

```
9           MessageBox(NULL, L"Привет, мир!!!", L"Оконная процедура", MB_OK);
10          return NULL; // возвращаем значение функции
11  }
12
```

Результат должен быть таким:



Теперь остановимся поподробнее на коде программы.

В первой строке мы подключаем заголовочный файл `windows.h`. В нём содержатся все необходимые «апишные» функции. Здесь всё понятно.

В **4-7 строках** у нас описание функции `int WINAPI WinMain()`.

Квалификатор WINAPI, нужен для функции WinMain всегда. Просто запомните это. WinMain – название функции. Она имеет четыре параметра. Первый из них – `HINSTANCE hInstance` (**строка 4**). hInstance является дескриптором экземпляра окна (это некий код оконной процедуры, идентификатор, по которой ОС будет отличать её от остальных окон). Через него можно обращаться к окну в процессе работы в других функциях (об этом позже), что-либо менять в параметрах окна. HINSTANCE является одним из многочисленных типов данных определенных в WinAPI, таким же как int, например. А запись HINSTANCE hInstance говорит нам о том, что мы создаём новую переменную типа HINSTANCE с названием hInstance.

О типах данным мы поговорим позже, поэтому переходим к следующему параметру: `HINSTANCE hPrevInstance` (**строка 5**). Как написано в комментариях, в Win32 он не используется, так как он создан для 3.x разрядной системы, из предыдущего понятно, что это дескриптор экземпляра окна. Далее у нас переменная типа `LPSTR` (**строка 6**) с именем `lpCmdLine`. Она используется в том случае, если мы запускаем окно через командную строку с прописью параметров. Очень экзотический способ, поэтому мы не будем на нём задерживаться.

И последний параметр: целочисленный, определяет способ показа окна. Нужен для функции `ShowWindow`, которая будет описана позже. Например, с помощью него мы можем развернуть окно на весь экран, сделать его определённой высоты, прозрачным или поверх остальных.

Переходим к функции `MessageBox()` (**строка 10**). Она имеет четыре параметра и нужна для вывода сообщений о ошибках, например. В данном случае мы использовали её для вывода сообщения. В общем виде описание функции выглядит следующим образом:

```
int MessageBox(HWND hWnd, // дескриптор родительского окна
               LPCTSTR lpText, // указатель на строку с сообщением
               LPCTSTR lpCaption, // указатель на строку с текстом
заголовка
               UINT uType);// флаги для отображения кнопок, стиля пиктограммы
 и прочее
```

В нашем случае, первому параметру присвоен ноль. Всё потому, что у нас нет родительских окон (оно не запущено какой-нибудь программой).

Далее у нас идут две переменные типа `LPCTSTR`: `lpText` и `lpCaption`. Первая сообщает информацию, которая будет выведена в окне в текстовом виде. Вторая сообщает, что будет написано в тексте заголовка к окну. Это аналог `char *str`, но всё же нет. Для того, чтобы текст выводился корректно, нужно перед строкой поставить букву `L` (*UNICODE* строка).

Ну и последний тип данных – `UINT` – 32-х битное целое без знака. То есть аналог `unsigned int`. Этому параметру можно передавать некоторые значения (о них тоже позже), за счёт чего можно менять вид кнопки. В нашем случае – это `MB_OK` — означает, что окно создаёт кнопку с надписью «ОК» и соответствующим действием при её нажатии (закрытием приложения).

В **строке 11** мы возвращаем значение функции, так как она имеет не тип `void`.

Таким образом, общее представление о WinAPI теперь есть.

Типы данных в Win32 API

В WINAPI определено множество типов данных, так же, как и в C/C++ (`int`, `char`, `float` и т.д.). Учить их определения не обязательно. Достаточно помнить, что они существуют, а когда они появятся или потребуются где-нибудь в программе, посмотреть их определения. В дальнейшем мы будем использовать их все. Условно их можно разделить на несколько видов: основные, дескрипторные, строковые и вспомогательные.

Основные типы

С основными типами данных трудностей возникнуть не должно. Если всё же возникнут, то нужно сюда.

`BOOL` – этот тип данных аналогичен `bool`. Он также имеет два значения – 0 или 1. Только при использовании WINAPI принято использовать вместо 0 спецификатор `NULL`. О нём ниже.

`BYTE` – байт, ну или восьмибитное беззнаковое целое число. Аналог – `unsigned char`.

`DWORD` — 32-битное беззнаковое целое. Аналоги: `unsigned long int`, `UINT`.

`INT` – 32-битное целое. Аналог – `long int`.

`LONG` – 32-битное целое – аналог всё также `long int`.

`NULL` – нулевой указатель. Вот его объявление:

```
1void *NULL=0;
```

UINT – 32-битное беззнаковое целое. Аналоги: `unsigned long int`, `DWORD`.

Дескрипторные типы данных

Про дескрипторные типы немного рассказывалось на вводном уроке в WINAPI. Дескриптор, как говорилось ранее, — это идентификатор какого-либо объекта. Для разных типов объектов существуют разные дескрипторы. Дескриптор объекта можно описать так:

```
1HANDLE h;
```

Есть также дескрипторы кисти, курсора мыши, шрифта и т.д. С их помощью мы можем при инициализации или в процессе работы приложения поменять какие-нибудь настройки, чего, например, мы не могли сделать в консольном приложении. Используются они в описательных функциях, управляющих типа: CreateProcess(), ShowWindow() и т.д. или как возвращаемое значение некоторых функций :

```
1// получает дескриптор для устройства ввода или вывода:
2HANDLE h = GetStdHandle(DWORD nStdHandle);
```

В этой функции мы получили дескриптор считывания потоков std_in и std_out. И можем, например, его использовать в каком-нибудь условии.

Не будем вдаваться в физику создания дескрипторов. Разве что, при необходимости или для большего понимания процессов.

Примечание: для удобства в WINAPI предусмотрены сокращения для типов данных. Первая буква **H** – означает, что это дескриптор, от слова handle.

HANDLE – дескриптор объекта.

HBITMAP – дескриптор растрового изображения. От фразы handle bitmap.

HBRUSH – дескриптор кисти. От фразы handle brush.

HCURSOR – дескриптор курсора. От фразы handle cursor.

HDC – дескриптор контекста устройства. От фразы handle device context.

HFONT – дескриптор шрифта. От фразы handle font.

HICONS – дескриптор криптограммы. От фразы handle icons.

HINSTANCE – дескриптор экземпляра приложения. От фразы handle instance.

HMENU – дескриптор меню. От фразы handle menu.

HPEN – дескриптор пера. От фразы handle pen.

HWND – дескриптор окна. От фразы handle window.

Строковые типы данных

Для начала начнём, с того, какие кодировки существуют в Windows ОС.

Есть два вида кодировок символов: *ANSI* и *UNICODE*. Однобайтные символы относятся к ANSI, двухбайтные — к кодировке UNICODE. Мы можем с лёгкостью подключить UNICODE кодировку в свойствах проекта. И тогда в коде создать переменную типа char можно будет так:

```
1// создаём строку из 10 элементов:
2wchar_t str[10];
```

Если же мы хотим использовать кодировку ANSI, то мы традиционно напишем:

```
1// тоже создаём строку из 10 элементов:
2char str[10];
```

В WINAPI, в зависимости от того, подключён юникод или нет, используются два вида строк UNICODE-ные или TCHAR-ные. Ниже описаны строковые типы данных.

Всё также для удобства, первые две буквы LP – от фразы long pointer сигнализируют о том, что это указатель.

LPCSTR – указатель на константную строку, заканчивающуюся нуль-терминатором. От фразы long pointer constant string.

LPCTSTR – указатель на константную строку, без UNICODE. От фразы long pointer constant TCHAR string. Это надстройка функции LPCSTR.

LPCWSTR – указатель на константную UNICODE строку. От фразы фразы long pointer constant wide character string. Это надстройка функции LPCSTR.

LPSTR – указатель на строку, заканчивающуюся нуль-терминатором. От фразы long pointer string.

LPTSTR – указатель на строку, без UNICODE. От фразы long pointer TCHAR string. Это надстройка функции LPSTR.

LPWSTR – указатель на UNICODE строку. От фразы long pointer wide character string. Это надстройка функции LPSTR.

TCHAR – символьный тип — аналог char и wchar_t.

Вспомогательные типы

Вспомогательные типы данных используются в некоторых функциях. В частности, параметры, описанные ниже, используются при работе с функцией обратного вызова оконной процедуры такого вида:

```
1LRESULT CALLBACK ИмяФункции(HWND hWnd, UINT uMSg,
2                            WPARAM wParam, LPARAM lParam);
```

Работа с данной функцией будет в следующих разделах.

LPARAM – тип для описания lParam (long parameter). Используются вместе с wparam в некоторых функциях.

LRESULT – значение, возвращаемое оконной процедурой имеет тип `long`.

WPARAM – тип для описания wParam (word parameter). Используются вместе с lParam в некоторых функциях.

На этом типы данных не закончены.

Создание полноценной оконной процедуры в Win32 API (Часть 1)

В первой статье по WINAPI мы разобрали программу, которая выводит окно с сообщением. Кроме того, после нажатия «ОК» окно закрывалось. И это всё, что она могла. Нетрудно догадаться, что при помощи Win32 API можно реализовывать более информативные окна. Поговорим о создании настоящей оконной процедуры.

Обобщённый алгоритм создания оконной процедуры в WINAPI:

1. Создать две функции: WinMain() – основная функция с параметрами, оговоренными в первой статье – аналог main в консоли, и функцию обрабатывающую процессы (название своё, пусть будет традиционно – WndProc()), потоки сообщений к/от ОС Windows.
2. Создать дескриптор окошка hMainWnd и зарегистрировать класс окошка WNDCLASSEX (то есть указать характеристики окна, которое будем создавать). Должен содержаться в WinMain.
3. Создать оболочку нашего окна. Должна содержаться в WinMain.
4. Создать циклы для обработки сообщений. Содержатся в функции WndProc.
5. Создать функцию для вывода окна ShowWindow и другие вспомогательные.

Теперь будет трудновато, потому что сейчас появятся различные функции с кучей параметров, классы, которые следует знать, понимать, ну и некоторые другие премудрости языков C и C++.

Для начала опишем общую структуру функции `WinMain()`:

```
1 int WINAPI WinMain(HINSTANCE hInst,
2                     HINSTANCE hPreviousInst,
3                     LPSTR lpCommandLine,
```

```
                    )
{
    // создаём дескриптор окна
    // описываем класс окна
    // создаём окошко, показываем его на экране
    // возвращаем значение при неудаче или при выходе
}
```

Для начала создадим дескриптор окна:

```
HWND hMainWnd; // создаём дескриптор будущего окошка
// и т.д.
```

Чтобы инициализировать поля класса окна, нужно создать его экземпляр, далее через него заполнить поля класса.

Объявление этого класса в `windows.h` выглядит так:

```
struct tagWNDCLASSEX{
      UINT cbSize; // величина структуры (в байтах)
      UINT style; // стиль класса окошка
      WNDPROC WndProc; // указатель на имя пользовательской функции
      int cbWndExtra; // число освобождаемых байтов в конце структуры
      int cbClsExtra; // число освобождаемых байтов при создании экземпляра
приложения
      HICON hIcon; // дескриптор значка
      HICON hIconMini; // .... маленького значка (в трэе)
      HCURSOR hCursor; // .... курсора мыши
      HBRUSH hbrBack; // .... цвета фона окошка
      HINSTANCE hInst; // .... экземпляра приложения
      LPCTSTR lpszClassName; // указатель на const-строку, содержащую имя
класса
      LPCTSTR lpszMenuName; // указатель на const-строку, содержащую имя
меню, применяемого для класса
}WNDCLASSEX;
```

То есть мы должны создать переменную типа `WNDCLASSEX`, обычно это `wc`, затем через неё инициализировать поля класса, примерно так:

```
// создавали дескриптор окна
WNDCLASSEX wc; // создаём экземпляр, для обращения к членам класса
WNDCLASSEX
wc.cbSize = sizeof(wc); // размер структуры (в байтах)
wc.lpfnWndProc = WndProc; // указатель на пользовательскую функцию
// и т.д.
```

Это необходимо, чтобы использовать этот класс в дальнейшем (если мы захотим). Это будет шаблоном для создания кучи окон. Конечно, на первых порах нам не нужно столько окошек. Но регистрация нужна однозначно! Это — формальная сторона. Мы даже можем задать параметры по умолчанию при инициализации полей, а не придумывать каким должно быть окно (о параметрах, которыми должны инициализировать поля класса мы обсудим в следующем уроке).

Итак, ещё раз: **если хоть одно поле класса не будет инициализировано, то окно не создастся**. Для проверки существует полезная функция `RegisterClassEx()`. Это и есть

следующий этап после заполнения полей класса `WNDCLASSEX`: обязательная проверка регистрации класса:

```
1
2// регистрировали класс
3if(!RegisterClassEx(&wc)){
      // в случае отсутствия регистрации класса:
4     MessageBox(NULL,
5                L"Не получилось зарегистрировать класс!",
6                L"Ошибка", MB_OK);
7     return NULL; // возвращаем, следовательно, выходим из WinMain
}
8// и т.д.
9
```

Как видим, в этом коде всё просто. Если `RegisterClassEx()` возвращает адекватный `ATOM` (таково описание:

```
1ATOM WINAPI RegisterClassEx(const WNDCLASSEX *lpWindowClass);
```

), то мы выводим соответствующее сообщение, которое мы разбирали в первой статье. Её параметр – это ссылка на экземпляр класса `WNDCLASSEX wc`.

Следующее, что мы обязаны сделать в функции `WinMain()` – вызвать функцию `CreateWindow()` и присвоить её значение дескриптора, который мы создавали на первом этапе. Примерно это так (описание всех этих параметров будет в следующем уроке):

```
  // проверяли, зарегистрирован ли класс
1 hMainWnd = CreateWindow(szClassName, // имя класса
2                 L"Полноценная оконная процедура", // имя окна (то что
  сверху)
3                 WS_OVERLAPPEDWINDOW | WS_VSCROLL, // режимы отображения
4 окошка
5                 CW_USEDEFAULT, // положение окна по оси x (по умолчанию)
6                 NULL, // позиция окна по оси y (раз дефолт в x, то писать
7 не нужно)
                  CW_USEDEFAULT, // ширина окошка (по умолчанию)
8                 NULL, // высота окна (раз дефолт в ширине, то писать не
9 нужно)
10                HWND(NULL), // дескриптор родительского окошка (у нас нет
11род. окон)
                  NULL, // дескриптор меню (у нас его нет)
12                HINSTANCE(hInst), // .... экземпляра приложения
13                NULL); // ничего не передаём из WndProc
  // и т.д.
```

Описание `CreateWindow()` в `windows.h` выглядит так:

```
1 HWND CreateWindow(
2                  LPCTSTR lpClassName, // имя нашего класса
3                  LPCTSTR lpWindowName, // название окошка (надпись сверху)
4                  DWORD dwStyle, // стиль окошка
5                  int x, // позиция окошка по оси x
6                  int y,  // позиция окна по оси y (отсчёт вниз)
7                  int nWidth, // ширина окошка
8                  int nHeight, // высота окошка
9                  HWND hWndParent, // идентификатор родительского окошка
                   HMENU hMenu, // .....меню
                   HINSTANCE hInst, // дескриптор экз-ра прил-ния
```

```
                    LPVOID lParam
                    ); // указатель на данные, передаваемые из
пользовательской функции


```

В случае удачи, функция в переменную `hMainWnd` возвратит не `NULL`. Наверное, не трудно догадаться, что надо сделать проверку (по аналогии с `RegisterClassEx()`):

```
// создали окно
if(!hMainWnd){
    // в случае некорректного создания окна (неверные параметры и тп):
    MessageBox(NULL, L"Не получилось создать окно!", L"Ошибка", MB_OK);
    return NULL; // выходим из приложения
}
// и т.д.
```

После чего нам необходимо вызвать две функции:

```
// проверяли, создано ли окно
ShowWindow(hMainWnd, nCommandShow);
UpdateWindow(hMainWnd);
// и т.д.
```

Подробно описывать их нет надобности, так как имеют несколько параметров.

Первая функция отображает окно на экране ПК. Её первый параметр – дескриптор окошка (он возвращался `CreateWindow()`). Второй параметр – стиль отображения. При первом запуске окна должен быть равен последнему параметру функции `WinMain()`, а в последующие разы можно вписывать свои данные (об этом в следующих статьях).

Вторая функция – видно по названию, отвечает за обновления окошка на экране при сворачиваниях или при динамической информации. Его параметр – всё тот же дескриптор окна.

В функции остался цикл и возвращение значение функции:

```
// показывали, обновляли окно
while(GetMessage(&msg, NULL, NULL, NULL)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
// всё!
```

Так как объекты структуры `MSG` связаны с рассылкой сообщений системному окну и наоборот и это всё содержится в пользовательской функции `WndProc`, поэтому описание этих функций будет в следующем уроке.

Где-то за 3 урока программа, выводящая полноценную оконную процедуру на экран, будет разобрана и у Вас появятся сведения о том, каким образом они создаются.

Напоследок модифицированная программа, выводящая окно с сообщением (из первой статьи) с другими кнопками:

```
1
2 #include <windows.h> // содержит API
3
4 // Основная функция:
5 int WINAPI WinMain (HINSTANCE hInst, // дескриптор экземпляра приложения
6                     HINSTANCE hPreviousInst, // в Win32 не используется, но
  объявление нужно
7                     LPSTR lpCommandLine, // нужен для запуска окошка в
8  режиме командной строки
9                     int nCommandShow) // режим отображения окна
10 {
11     int result = MessageBox(NULL, L"Вам нравится WINAPI?!", L"Задача",
12                              MB_ICONQUESTION | MB_YESNO);
13     switch (result)
14     {
15     case IDYES: MessageBox (NULL, L"Продолжайте в том же духе!!!",
16                             L"Ответ", MB_OK| MB_ICONASTERISK); break;
17     case IDNO:  MessageBox (NULL, L"Очень жаль!!!", L"Ответ",
18                             MB_OK| MB_ICONSTOP); break;
19     }
return NULL;
}
```

Скомпилируйте данный код, и Вы обнаружите, что с функцией `MessageBox()` тоже можно поэкспериментировать.

Черточки между параметрами в **строках 10,14 и 17** означают, что параметры используются вместе.

Третьим параметром мы можем записать следующие идентификаторы:

параметры кнопок:

MB_ABORTRETRYIGNORE — три кнопки: ABORT, RETRY, IGNORE
MB_CANCELTRYCONTINUE — три кнопки: CANCEL, TRY, CONTINUE **MB_HELP**
MB_OK MB_OKCANCEL — 2 кнопки: OK, CANCEL **MB_RETRYCANCEL** — 2 кнопки: RETRY, CANCEL **MB_YESNO** — 2 кнопки: YES, NO **MB_YESNOCANCEL** — три кнопки: YES, NO, CANCEL

параметры пиктограммы:

MB_ICONSTOP — выводит крестик **MB_ICONQUESTION** — …….знак вопроса
MB_ICONEXCLAMATION — …….восклицательный знак в треугольнике
MB_ICONASTERISK — …….восклицательный знак

А возвращать значения при нажатии вышеуказанных кнопок функция MessageBox будет такие:

IDABORT — при нажатии на ABORT **IDCANCEL** – …….на кнопку CANCEL
IDCONTINUE – ……..на кнопку CONTINUE **IDIGNORE** – …….на кнопку IGNORE
IDNO – …….на кнопку NO **IDOK** – …….на кнопку OK **IDRETRY** – …….на кнопку RETRY **IDTRYAGAIN** – …….на кнопку TRY AGAIN **IDYES** – …….на кнопку YES

Поэкспериментируйте с данными идентификаторами. С параметрами отображения мы будем иметь дело во многих функциях.

Создание полноценной оконной процедуры в Win32 API (Часть 2)

В предыдущей статье мы на половину закончили разработку полноценной оконной процедуры. В этой статье она будет закончена. Итак, мы остановились на функции `WinMain()`. В её последних строках содержались вызовы двух функций, взаимодействующих с нашей функцией `WndProc()` (о ней здесь и будет идти речь) и оператор возврата значения функции. Конструкция функции `WndProc()` должна быть условно такая:

```
LRESULT CALLBACK WndProc(HWND hWnd, // дескриптор окошка
                         UINT uMsg, // сообщение, посылаемое ОС
                         WPARAM wParam, // параметры
                         LPARAM lParam) // сообщений, для последующего
обращения
{
        // 1) создаём нужные переменные
        // 2) расписываем условия, при которых нужно выполнить нужное
действие (switch)
        // 3) Возвращаем значение функции
}
```

`LRESULT` – это возвращаемое значение. `CALLBACK` нужно писать, так как это функция обратного вызова. Сначала создаём необходимые переменные. Для начала создадим экземпляр контекста устройства `HDC` для нужной ориентации текста в окне. Кроме того, для обработки области окна нам нужны ещё две переменные `RECT` (прямоугольная область окна) и `PAINTSTRUCT` (в структуре информация об окне для прорисовки) соответственно:

```
// создаём нужные переменные:
HDC hDC;
PAINTSTRUCT ps;
RECT rect;
// и т.д.
```

Чтобы поменять цвет текста (и не только) нужно создать переменную типа `COLORREF` и присвоить ей возвращаемое значение функции `RGB()` (от англ. Red Green Blue) с тремя параметрами:

```
// создавали нужные переменные
COLORREF colorText = RGB(255, 0, 0); // параметры int
// и т.д.
```

Эта функция преобразовывает целочисленный тип в интенсивность цвета и возвращает значение смешивания трёх цветов (интенс. красн. + интенс. зел.+ интенс. син.). Как видно с помощью градаций этих цветов, можно создать 255*255*255= 16’581’375 цветов. ОС Windows и различные функции наших программ ежесекундно отправляют тысячи сообщений каждому отдельному приложению. Эти сообщения могут быть отправлены из-за нажатия клавиши или нажатия кнопки на мыши и т.д. Для этих случаев у нас существует структура `MSG`, описанная в `WinMain()`, которая хранит информацию об этих сообщениях. В `WndProc()` есть условия выбора для этих сообщений. Если ОС отправляет сообщение `WM_PAINT`, например, то в окне должно что-то рисоваться. Эти сообщения обрабатываются условием `switch()`(оператор множественного выбора), параметрами

которого являются `uMsg`. `uMsg` мы создали при описании нашей функции `WndProc()`. Основной значение в этом операторе, без которых окно не будет обновляться после сворачивания и не закроется: `WM_DESTROY`. Также есть `WM_PAINT`, который нужен для прорисовки в клиентской области. WM – от слов Window Message. То есть:

```
1
2// создавали переменные
3 switch(uMsg){
4     case WM_PAINT:
5     // что то рисуем
6     case WM_DESTROY:
7     // обязательно делаем условие закрытия окошка
8     default:
9         return DefWindowProc(hWnd, uMsg, wParam, lParam); // об этом далее
}
```

И что же мы должны делать в этих «кейсах»? При посылке сообщения `WM_PAINT` – вызываем функции рисования `BeginPaint()`, `GetClientRect()`, `SetTextColor()`, `DrawText()`, `EndPaint()`. По названию видно, что эти функции делают. Функция `BeginPaint()` в прямом смысле начинает рисовать. Только для этого ей нужно иметь дескриптор окна и объект `PAINTSTRUCT` (у нас это `ps`). Она возвращает значение типа `HDC`, поэтому нам нужно присвоить ей `hDc`. `GetClientRect()` выбирает область. Параметры у неё аналогичны предыдущей функции: дескриптор окна и указатель на объект класса `RECT` (у нас это `rect`). Функция `SetTextColor()` возвращает цвет текста. Её параметры: возвращаемое значение функции `BeginPaint()` – `hDC` и указатель на объект класса `COLORREF`. Мы могли и не задавать отдельно цвет текста, создавая при этом переменную `colorText`, а могли сделать это прямо в ней. Но с точки зрения читаемости кода и его понятливости – это в корне не правильно. Старайтесь всегда объявлять переменные отдельно и писать в комментариях, зачем они нужны и тогда не будет вопросов, какие параметры имеет функция, спустя год как вы последний раз закрыли проект по WinAPI. Также соблюдайте венгерскую нотацию по программированию, суть которой: имена переменных должны нести смысл их существования и показывать тип данных. Объявление функции `DrawText()`:

```
1
2int DrawText(
3       HDC hDC,// дескриптор контекста устр-ва
4       LPCTSTR lpchText, // указатель на нашу строку
5       int nCount, // длина текста (если равно -1, то определяет сам)
6       LPRECT lpRect, // указатель на объект RECT
7       UINT uFormat // формат отображения текста
);
```

На счёт первых 4х параметров всё ясно. Четвёртый `uFormat` – имеет несколько видов. Обычно используются `DT_SINGLELINE`, `DT_CENTER` и `DT_VCENTER` для отображения текста в центре области, в одну линию. Но Вы можете задать другие параметры (о чём поговорим в следующих уроках). Функция `EndPaint()` имеет два параметра: дескриптор окна и объект `ps`. Заметили аналогию с `BeginPaint()`? Что делать при вызове `WM_PAINT`, мы знаем (не забываем в конце дописать `break`). `WM_DESTROY` посылается окошку функцией `DestroyWindow()`, которая вызывается в случае, если мы его закрыли. А это происходит в операторе `default`. При этом происходит вызов функции `DefWindowProc()`, параметры которой те же, что и у `WndProc()`. В своём теле `WM_DESTROY` должен иметь функцию `PostQuitMessage()`, которая посылает `WinMain()` сообщение `WM_QUIT`. Её параметр обычно `NULL`, интерпретирующийся для главной функции `WinMain()`, как `WM_QUIT`.

```
VOID WINAPI PostQuitMessage(int nExitCode);
```

По завершению `switch()` мы возвращаем нулевое значение функции `WndProc()`. А теперь вернёмся к `WinMain()`, в частности к обработчику сообщений:

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

Функция `GetMessage()` имеет такое описание:

```
BOOL WINAPI GetMessage(LPMSG lpMsg, // указатель на структуру MSG
                       HWND hWnd, // дескриптор окошка
                       UINT wMsgFilterMin, // фильтры
                       UINT wMsgFilterMax // фильтры для выборки сообщений
);
```

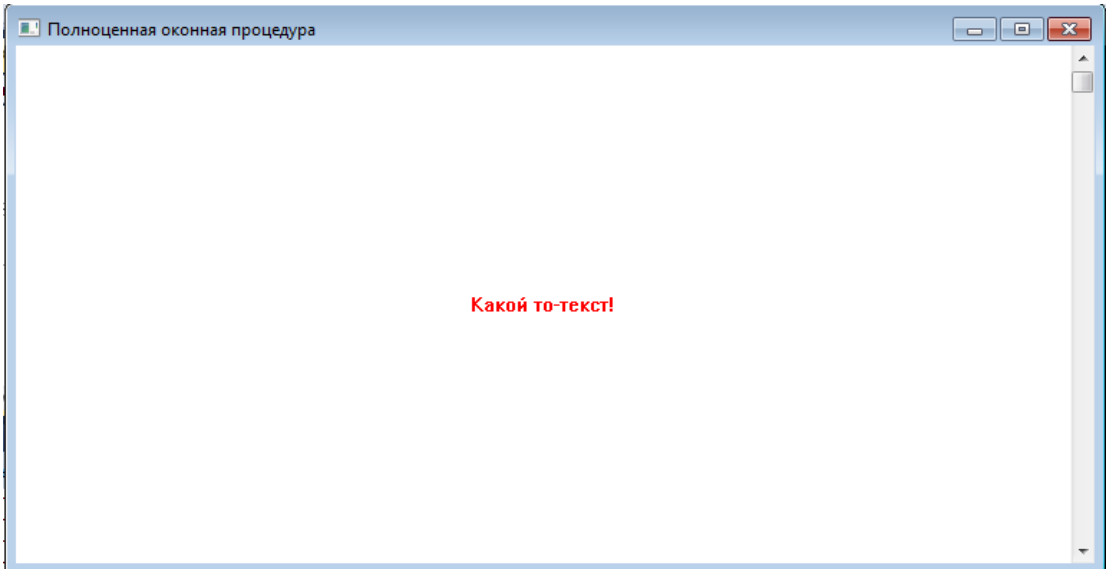
Она обрабатывает сообщения, посылаемые ОС. Первый параметр – это адрес структуры `MSG`, в которую помещается очередное сообщение. Второй параметр — дескриптор окна. Третий и четвёртый параметры указывают порядок отбора сообщений. Обычно это нулевые значения и функция отбирает любые значения из очереди. Цикл прекращается, если она получает сообщение `WM_QUIT`. В таком случае она возвращает `FALSE` и мы выходим из программы. Функции `TranslateMessage()` и `DispatchMessage()` в цикле нужны для интерпретации самих сообщений. Обычно это используется при обработке нажатых кнопок на клавиатуре. В следующих уроках мы познакомимся с этими хитростями. По окончанию цикла мы возвращаем ОС код возврата `msg.wParam`. В итоге у нас должен получиться такой код:

```
#include <windows.h> // заголовочный файл, содержащий WINAPI

// Прототип функции обработки сообщений с пользовательским названием:
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
TCHAR mainMessage[] = L"Какой то-текст!"; // строка с сообщением

// Управляющая функция:
int WINAPI WinMain(HINSTANCE hInst, // дескриптор экземпляра приложения
                   HINSTANCE hPrevInst, // не используем
                   LPSTR lpCmdLine, // не используем
                   int nCmdShow) // режим отображения окошка
{
    TCHAR szClassName[] = L"Мой класс"; // строка с именем класса
    HWND hMainWnd; // создаём дескриптор будущего окошка
    MSG msg; // создём экземпляр структуры MSG для обработки сообщений
    WNDCLASSEX wc; // создаём экземпляр, для обращения к членам класса
WNDCLASSEX
    wc.cbSize        = sizeof(wc); // размер структуры (в байтах)
    wc.style         = CS_HREDRAW | CS_VREDRAW; // стиль класса окошка
    wc.lpfnWndProc   = WndProc; // указатель на пользовательскую функцию
    wc.lpszMenuName  = NULL; // указатель на имя меню (у нас его нет)
    wc.lpszClassName = szClassName; // указатель на имя класса
    wc.cbWndExtra    = NULL; // число освобождаемых байтов в конце
структуры
    wc.cbClsExtra    = NULL; // число освобождаемых байтов при создании
экземпляра приложения
    wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO); // декриптор
```

```
пиктограммы
    wc.hIconSm       = LoadIcon(NULL, IDI_WINLOGO); // дескриптор маленькой
пиктограммы (в трэе)
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW); // дескриптор курсора
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); // дескриптор
кисти для закраски фона окна
    wc.hInstance     = hInst; // указатель на строку, содержащую имя меню,
применяемого для класса
    if(!RegisterClassEx(&wc)){
        // в случае отсутствия регистрации класса:
        MessageBox(NULL, L"Не получилось зарегистрировать класс!",
L"Ошибка", MB_OK);
        return NULL; // возвращаем, следовательно, выходим из WinMain
    }
    // Функция, создающая окошко:
    hMainWnd = CreateWindow(
        szClassName, // имя класса
        L"Полноценная оконная процедура", // имя окошка (то что сверху)
        WS_OVERLAPPEDWINDOW | WS_VSCROLL, // режимы отображения окошка
        CW_USEDEFAULT, // позиция окошка по оси x
        NULL, // позиция окошка по оси y (раз дефолт в x, то писать не
нужно)
        CW_USEDEFAULT, // ширина окошка
        NULL, // высота окошка (раз дефолт в ширине, то писать не нужно)
        (HWND)NULL, // дескриптор родительского окна
        NULL, // дескриптор меню
        HINSTANCE(hInst), // дескриптор экземпляра приложения
        NULL); // ничего не передаём из WndProc
    if(!hMainWnd){
        // в случае некорректного создания окошка (неверные параметры и
тп):
        MessageBox(NULL, L"Не получилось создать окно!", L"Ошибка", MB_OK);
        return NULL;
    }
    ShowWindow(hMainWnd, nCmdShow); // отображаем окошко
    UpdateWindow(hMainWnd); // обновляем окошко
    while(GetMessage(&msg, NULL, NULL, NULL)){ // извлекаем сообщения из
очереди, посылаемые фу-циями, ОС
        TranslateMessage(&msg); // интерпретируем сообщения
        DispatchMessage(&msg); // передаём сообщения обратно ОС
    }
    return msg.wParam; // возвращаем код выхода из приложения
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam){
    HDC hDC; // создаём дескриптор ориентации текста на экране
    PAINTSTRUCT ps; // структура, сод-щая информацию о клиентской области
(размеры, цвет и тп)
    RECT rect; // стр-ра, определяющая размер клиентской области
    COLORREF colorText = RGB(255, 0, 0); // задаём цвет текста
    switch(uMsg){
    case WM_PAINT: // если нужно нарисовать, то:
        hDC = BeginPaint(hWnd, &ps); // инициализируем контекст устройства
        GetClientRect(hWnd, &rect); // получаем ширину и высоту области для
рисования
        SetTextColor(hDC, colorText); // устанавливаем цвет контекстного
устройства
        DrawText(hDC, mainMessage, -1, &rect,
DT_SINGLELINE|DT_CENTER|DT_VCENTER); // рисуем текст
        EndPaint(hWnd, &ps); // заканчиваем рисовать
        break;
    case WM_DESTROY: // если окошко закрылось, то:
        PostQuitMessage(NULL); // отправляем WinMain() сообщение WM_QUIT
        break;
```

```
    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam); // если закрыли
окошко
    }
    return NULL; // возвращаем значение
}
```

А после компиляции такое окошко:



Ввод и вывод данных Win32 API

В этой статье мы научимся обрабатывать данные, поступающие с клавиатуры. В дальнейшем это будет полезным пособием при создании простейшего текстового редактора типа notepad.exe.

Ниже представлен листинг:

```
#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInst,
                   HINSTANCE hPrevInst,
                   LPSTR pCommandLine,
                   int nCommandShow){
    TCHAR className[] = L"Мой класс";
    HWND hWindow;
    MSG message;
    WNDCLASSEX windowClass;

    windowClass.cbSize        = sizeof(windowClass);
    windowClass.style         = CS_HREDRAW | CS_VREDRAW;
    windowClass.lpfnWndProc   = WindowProcess;
    windowClass.lpszMenuName  = NULL;
    windowClass.lpszClassName = className;
    windowClass.cbWndExtra    = NULL;
    windowClass.cbClsExtra    = NULL;
    windowClass.hIcon         = LoadIcon(NULL, IDI_WINLOGO);
```

```
    windowClass.hIconSm       = LoadIcon(NULL, IDI_WINLOGO);
    windowClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    windowClass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    windowClass.hInstance     = hInst;

    if(!RegisterClassEx(&windowClass))
    {
        MessageBox(NULL, L"Не получилось зарегистрировать класс!",
L"Ошибка", MB_OK);
        return NULL;
    }
    hWindow = CreateWindow(className,
                           L"Программа ввода символов",
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT,
                           NULL,
                           CW_USEDEFAULT,
                           NULL,
                           (HWND)NULL,
                           NULL,
                           HINSTANCE(hInst),
                           NULL
                          );
    if(!hWindow){
        MessageBox(NULL, L"Не получилось создать окно!", L"Ошибка",
MB_OK);
        return NULL;
    }
    ShowWindow(hWindow, nCommandShow);
    UpdateWindow(hWindow);
    while(GetMessage(&message, NULL, NULL, NULL)){
        TranslateMessage(&message);
        DispatchMessage(&message);
    }
    return message.wParam;
}

LRESULT CALLBACK WindowProcess(HWND hWindow,
                               UINT uMessage,
                               WPARAM wParameter,
                               LPARAM lParameter)
{
    HDC hDeviceContext;
    PAINTSTRUCT paintStruct;
    RECT rectPlace;
    HFONT hFont;
    static char text[2]={' ','\0'};
    switch (uMessage)
    {
    case WM_CREATE:
        MessageBox(NULL,
              L"Пожалуйста, вводите символы и они будут отображаться на
экране!",
              L"ВНИМАНИЕ!!!", MB_ICONASTERISK|MB_OK);
        break;
    case WM_PAINT:
        hDeviceContext = BeginPaint(hWindow, &paintStruct);
        GetClientRect(hWindow, &rectPlace);
        SetTextColor(hDeviceContext, NULL);
        hFont=CreateFont(90,0,0,0,0,0,0,0,
                         DEFAULT_CHARSET,
                         0,0,0,0,
                         L"Arial Bold"
```

```
                        );
        SelectObject(hDeviceContext,hFont);
        DrawText(hDeviceContext, (LPCWSTR)text, 1, &rectPlace,
DT_SINGLELINE|DT_CENTER|DT_VCENTER);
        EndPaint(hWindow, &paintStruct);
        break;
    case WM_KEYDOWN:
        switch (wParameter)
        {
        case VK_HOME:case VK_END:case VK_PRIOR:
        case VK_NEXT:case VK_LEFT:case VK_RIGHT:
        case VK_UP:case VK_DOWN:case VK_DELETE:
        case VK_SHIFT:case VK_SPACE:case VK_CONTROL:
        case VK_CAPITAL:case VK_MENU:case VK_TAB:
        case VK_BACK:case VK_RETURN:
            break;
        default:
        text[0]=(char)wParameter;
        InvalidateRect(hWindow, NULL, TRUE);
        break;
        }break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWindow, uMessage, wParameter, lParameter);
    }
    return NULL;
}
```

Данная программа выводит цифры и буквы, при этом не реагирует на нажатие дополнительных клавиш, типа enter, shift и т.д.

В функции `WinMain()` ничего нового. А в `WindowProcess()` мы добавили обработчики событий нажатия клавиши.

Нажатие любой клавиши автоматически посылает системе аппаратное сообщение, которое обрабатывается и высылается приложению. За эту операцию отвечает сообщение `WM_KEYDOWN`.

Его описание сложно и на данном этапе не имеет смысла, так как мы не включили кириллицу. При посылке этого сообщения в `wParameter` записывается значение нажатой

клавиши. Конечно же, всё зависит от клавиатуры, раскладки. Поэтому всего кодов для клавиш много и мы не будем описывать, что делать при нажатии той или иной, а просто исключим значения не нужных нам клавиш.

С 87-й по 101-ю строки мы обрабатываем сообщение `WM_KEYDOWN` (нажатие клавиши). Сначала мы создаём переключатель `switch` для значения `wParameter`, куда записан идентификатор клавиши. **С 90-й по 95-ю** строки мы исключаем значения home, end, page up, page down, стрелка влево, стрелка вправо, стрелка вверх, стрелка вниз, delete, shift, space, ctrl, caps lock, alt, tab, backspace, enter соответственно. Возможно, на вашей клавиатуре есть ещё клавиши, вы можете также их исключить при необходимости, потому что при нажатии будет «абракадабра». То есть если в идентификаторе `wParameter` у нас эти значения, мы ничего не делаем – **в 96-й** строке оператор `break`.

А под значениями `default` у нас выполняется преобразование. **В 66** строке мы создали статическую строку типа `char` из 2-х элементов. Статическая она, потому что в процессе работы программы мы неоднократно выходим из `WindowProcess()` в цикл обработки с `GetMessage()`, расположенный в `WinMain()`, а значение клавиши нам нужно сохранять. Ну а **в 98** строке мы инициализируем её первый элемент `wParameter`-ом.

В следующей строке мы вызываем функцию `InvalidateRect()`, после чего выходим из `WM_KEYDOWN`. Первым параметром идёт – дескриптор окна, вторым – область будущего «затирания» окна (`NULL` – всё окно перерисовывается в `WM_PAINT`), третьим параметром идёт «затирание» фона (если `TRUE`, то стираем). При этом мы автоматически посылаем приложению message `WM_PAINT` и окно перерисовывается заново. Для чего это нужно?

Как мы помним в случаях сворачивания, перемещения окна вызывается `WM_PAINT`. Нам же это нужно сделать явно, то есть сделать окно недействительным, для того, чтобы сразу вывести значение нажатой клавиши на экран. Иначе она появится только тогда, когда мы начнём манипулировать окном.

Теперь возвратимся **65-й** строке. В ней объявлен экземпляр класса `HFONT`, который отвечает за шрифт, его размер и другие премудрости. Он будет использоваться в `WM_PAINT`.

В 69-й строке есть обработчик сообщения `WM_CREATE`. Он срабатывает, когда создаётся окно, то есть вызывается функция `CreateWindow()`. На этом этапе решено создать дополнительное окно с сообщением для удобства (**строки 70-72**).

Наконец, обработчик `WM_PAINT`. **С 75-й по 77-ю** строки происходит вызов функций контекста устройства (описывалось в статье).

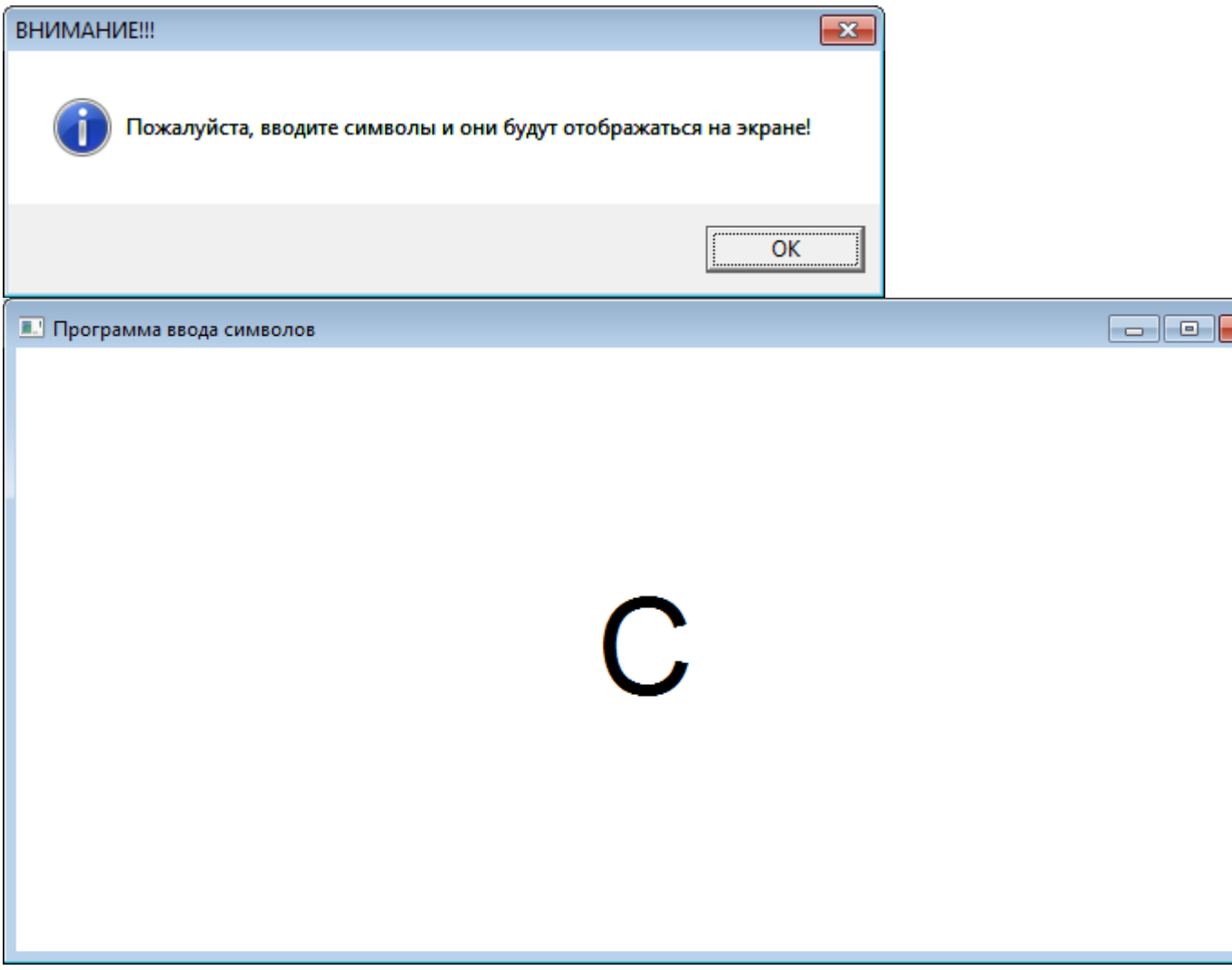
А **с 78-й по 82-ю** строки вызывается функция `CreateFont()`. С её помощи мы меняем шрифт, и размер буквы. Вот её описание:

```
1 HFONT CreateFont(int,     // высота шрифта
2                  int,     // ширина символов
3                  int,     // угол наклона букв
4                  int,     // угол наклона строки
5                  int,     // толщина букв («жирность»)
6                  DWORD,   // курсив
7                  DWORD,   // подчеркивание
8                  DWORD,   // перечеркивание
                   DWORD,   // набор символов
```

```
9                   DWORD,  // точность вывода
10                  DWORD,  // точность отсечения
11                  DWORD,  // качество вывода
12                  DWORD,  // шаг между буквами
13                  LPCTSTR // имя шрифта
14                 );
15
```

Шрифт выставлен Arial Bold – то есть Arial жирным. **В строке 83** мы задаём контексту изменённые параметры шрифта. **С 84-й по 85-ю** строки ничего нового. Мы вводим на экран полученный идентификатор клавиши, преобразованный в тип `char`. В функции `DrawText()` его нужно преобразовать к типу `LPCWSTR`, что она и требует.

Результат работы программы при вводе буквы «с» такой:



В следующей статье мы научимся обрабатывать дополнительные клавиши, переключать раскладку, создавать таймеры отсчёта.

Ввод и вывод текста. Буфер памяти, таймер отсчёта

В этой статье речь пойдёт о буфере и о таймере отсчёта.

Для начала убедитесь, что Вы разбираетесь в выделении памяти с помощью функции alloc (malloc), free, с помощью операторов new, delete и понимаете функцию преобразованию itoa().

Рассмотрим листинг, представленный ниже:

```
LRESULT CALLBACK WindowProcess(HWND hWindow, UINT uMessage,
                    WPARAM wParameter, LPARAM lParameter)
{
    HDC hDeviceContext;
    PAINTSTRUCT paintStruct;
    RECT rectPlace;
    HFONT hFont;

    static PTCHAR text;
    static int size = 0;

    switch (uMessage)
    {
    case WM_CREATE:
        text=(PTCHAR)GlobalAlloc(GPTR,50000*sizeof(TCHAR));
        break;
    case WM_PAINT:
        hDeviceContext = BeginPaint(hWindow, &paintStruct);
        GetClientRect(hWindow, &rectPlace);
        SetTextColor(hDeviceContext, NULL);
        hFont=CreateFont(50,0,0,0,0,0,0,0,
                 DEFAULT_CHARSET,
                 0,0,0,VARIABLE_PITCH,
                 "Arial Bold");
        SelectObject(hDeviceContext,hFont);
        if(wParameter != VK_RETURN)
            DrawText(hDeviceContext,
                 (LPCSTR)text,
                 size, &rectPlace,
                 DT_SINGLELINE|DT_CENTER|DT_VCENTER);
        EndPaint(hWindow, &paintStruct);
        break;
    case WM_CHAR:
        switch(wParameter)
        {
        case VK_RETURN:
            size=0;
            break;
        default:
            text[size]=(char)wParameter;
            size++;
            break;
        }
        InvalidateRect(hWindow, NULL, TRUE);
                 break;
    case WM_DESTROY:
        PostQuitMessage(NULL);
        GlobalFree((HGLOBAL)text);
        break;
    default:
        return DefWindowProc(hWindow, uMessage, wParameter, lParameter);
    }
    return NULL;
}
```

47
48
49
50
51
52
53
54

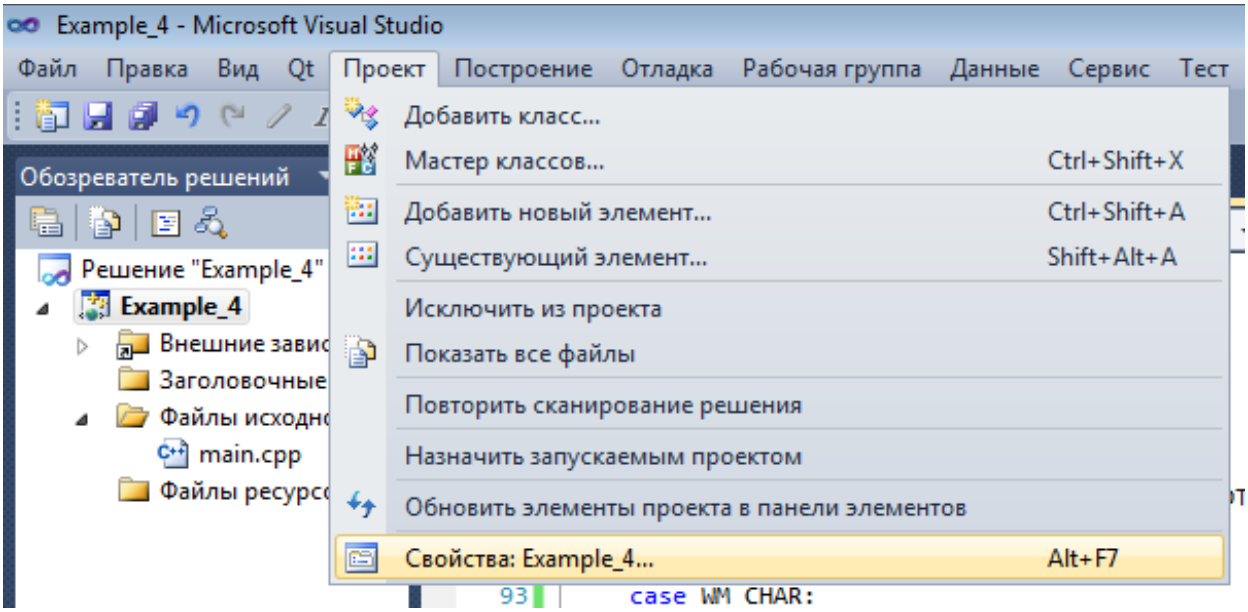
Результат после ввода CppStudio.com будет таким:

Код для функции `WinMain` не приводится, так как он не менялся с предыдущих уроков. Данная программа позволяет вводить текст по мере возможности и стирать его клавишей ENTER.

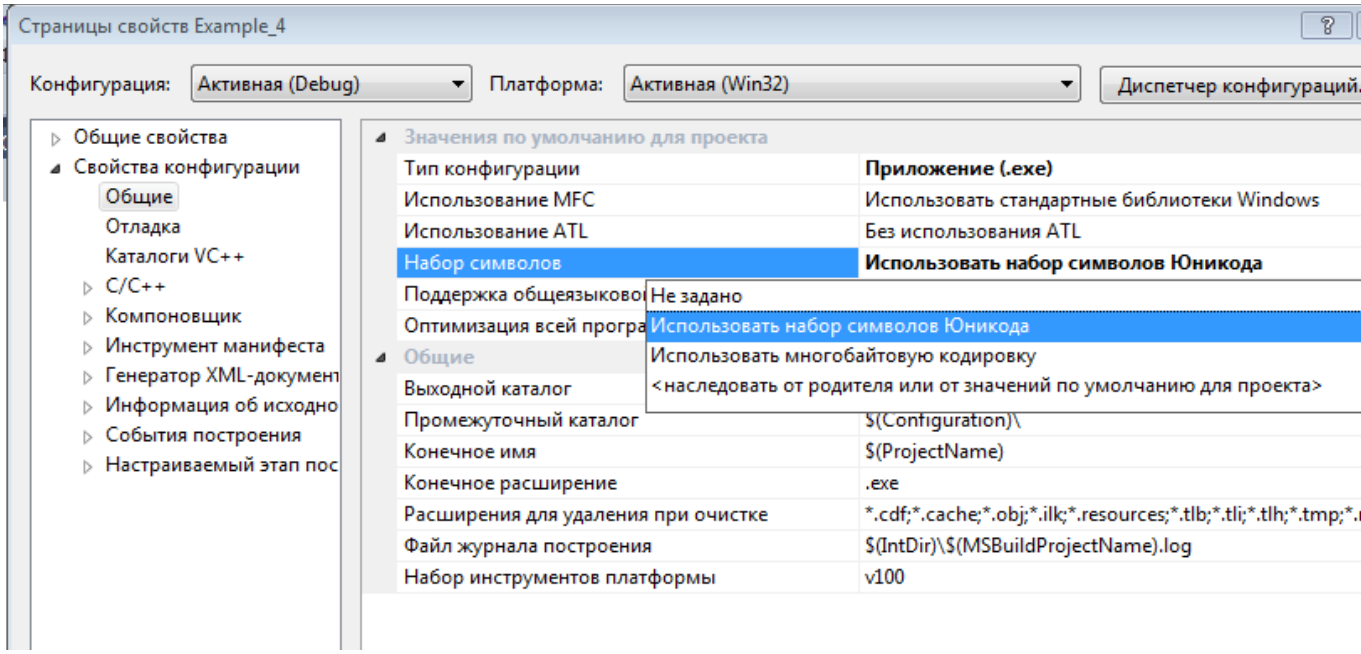
Наверное, Вы заметили, что в программе в предыдущем уроке для считывания нажатой клавиши мы использовали message `WM_KEYDOWN`, а здесь что-то новое — `WM_CHAR`.

Отличительная особенность в том, что `WM_KEYDOWN`нужен для обработки специальных клавиш типа shift, ctrl и т.п., а `WM_CHAR`– отвечает за обработку символов. Обычно они работают вместе для комбинаций клавиш и вызывают друг друга. В этом Вы убедитесь, когда мы будем разбирать работу текстового редактора.

По умолчанию в MVS используется Юникод. Его отключение производится следующим образом: заходим в свойства проекта



и отключаем Юникод, нажав «Использовать многобайтовую кодировку».



После того, как Вы это сделаете, перед строками не нужно будет писать L, и представленный код не будет иметь ошибок. По этой причине в программе прошлого урока было не правильное отображение некоторых символов.

Теперь второе отличие: мы можем вводить текст (примерно до 50 кБ), который сохранится в буфере.

Конструкция создания, использования, удаления буфера выглядит так:

```
static PTCHAR text;
text=(PTCHAR)GlobalAlloc(GPTR,50000*sizeof(TCHAR));
if()
{
   //....
   // заполняем буфер информацией, используем её, где нам нужно
```

```
6    //.....
7    GlobalFree((HGLOBAL)text);
8}
9
```

Она аналогична `alloc()` и `free()` и её смысл абсолютно такой же.

PTCHAR — указатель на тип `TCHAR`.

GPTR — режим выделения памяти (в данном случае фиксированное) **HGLOBAL** — тип возвращаемого значения функции `GlobalAlloc()`

Третье достоинство нашей программы — возможность стирать текст (**строки 36-38**).

В случае нажатия клавиши ENTER – мы делаем окно не действительным с помощью функции `InvalidateRect()` (**строка 44**), тем самым вызываем сообщение `WM_PAINT`. В нём прописано, что если параметр `wParameter` был равен `VK_RETURN` (то есть ENTER), то мы перерисовываем окно и не вызываем функцию `DrawText()`.

И, как и обещалось ранее, добавим в функцию таймер и некоторые другие особенности.

```
1
2
3 static int sec = 0;
4 TCHAR workTime[10];
5 //...
6 switch (wParameter)
7 {
8 case WM_CREATE:
9      SetTimer(hWindow, 1, 1000, NULL);
10     //...
11     SetForegroundWindow(hWindow);
12     break;
13case WM_CHAR:
14     //...
15     case VK_BACK:
16        if(size!=0)
17           size--;
18        break;
19case WM_TIMER:
20     sec++;
21     break;
22case WM_DESTROY:
23     KillTimer(hWindow, 1);
24     _itoa(sec, workTime, 10); MessageBox(NULL, (LPCSTR)workTime, "Время
25работы программы (сек.):", MB_ICONASTERISK|MB_OK);
26     //...
27     break;
28}
29
30
```

Добавьте в соответствующие места программы выше представленный код.

Функция `SetTimer()` создаёт таймер. Первый параметр — дескриптор окна, третий — интервал срабатывания `WM_TIMER` в миллисекундах, второй — количество вызовов `WM_TIMER` в заданный интервал времени. Четвёртый параметр — нам не нужен.

В сообщении `WM_TIMER` мы увеличиваем `sec` на единицу.

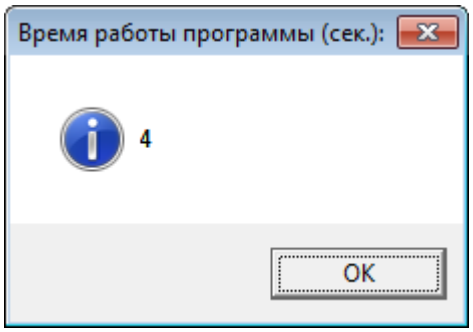
Функция `KillTimer()` — удаляет таймер. Параметры те же, что и в `SetTimer()`.

Чтобы отобразить время работы программы, в `WM_DESTROY`мы преобразовываем `int` в `TCHAR` массив при помощи функции `_itoa`, выводим строку на экран, удаляем таймер и освобождаем память, выделенную под буфер.

Функция `SetForegroundWindow(HWND)` – делает окно активным. Например, если Вы находитесь на сайте CppStudio.com и при этом идёт компиляция данной программы, это окно появится поверх браузера с фокусом для ввода.

Для возможности удаления одного или нескольких символов добавлен обработчик `VK_BACK`– нажатие клавиши backspace. Если `size` не равен нулю (то есть не был нажат ENTER), мы уменьшаем его на единицу и последний элемент при обновлении окна не отображается.

В результате, при закрытии окна, мы получим такое сообщение:



Коммент:

Надо так делать:

```
1HFONT hOldFont;//Добавили новую переменную
2.............................................
3//Строку SelectObject(hDeviceContext,hFont) заменяем на:
4hOldFont=static_cast<HFONT>(SelectObject(hDeviceContext,hFont));//Запомнили
5старый шрифт
6.............................................
7//Перед  EndPaint добавляем строки:
8SelectObject(hDeviceContext,hOldFont);//Вернули старый шрифт
9DeleteObject(hFont);//Удалили шрифт созданный через CreateFont
```

Иначе утечка памяти будет. И в предыдущей статье тоже.