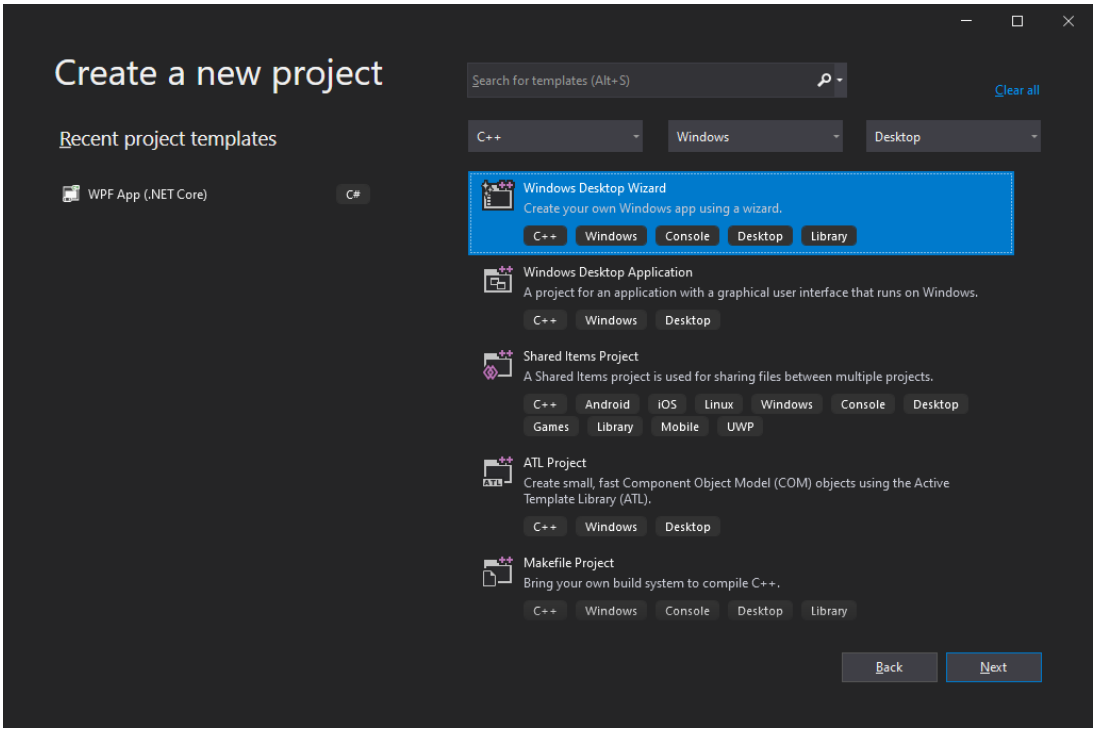
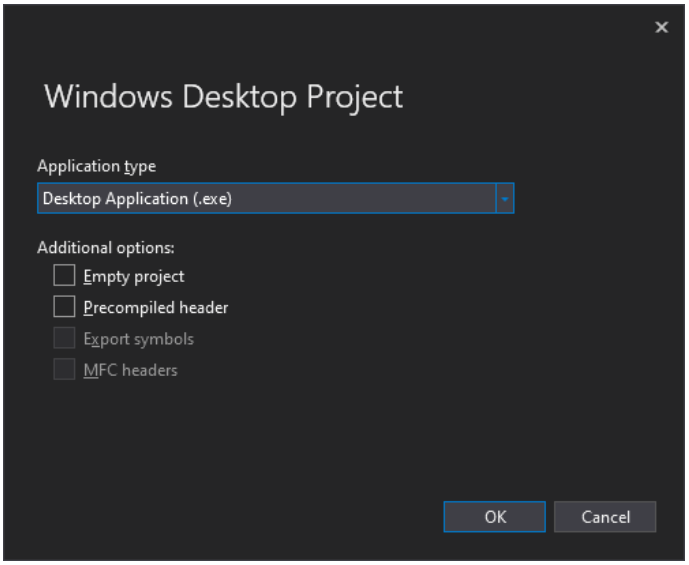
# Windows API
# (Microsoft Visual Studio 2019) - Tutorial

1. Using Visual Studio 2019 for creating Windows application using Win32 API and C++
   - Create a new project (select Windows Desktop Wizard)




   - In the Windows Desktop Wizard choose from combobox "Windows Application (.exe)" and uncheck option "Empty Project" to generate project with simple Win32 application template:




   - Be sure to understand all available application type options (to read more information about all options available in the Windows Desktop Wizard:

    - Windows application - an application with Windows graphical user interface (GUI)
    - Console application - an application without Windows, it can communicate with the user using the console
    - DLL - a dynamic linking library - it is compiled into a .dll file (so it cannot be run directly) and it can contain code using the graphical user interface
    - Static library - a library of code that can be linked statically to other modules (executable files or DLLs)
- Let the Windows Desktop Wizard to generate the sample Win32 application
  - Compile and run the application

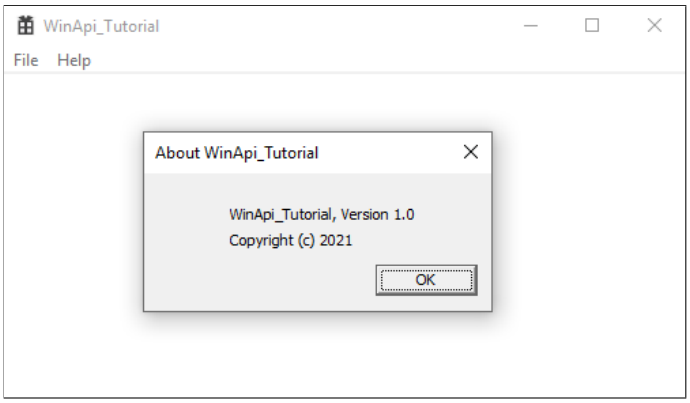



Figure 1: Application window

    - Note all standard features of a Windows application:
      - The main window: resizing, minimizing, maximizing, the icon, system menu in the icon, menu in the window
      - About - the modal dialog box: it must be closed to allow the user to do something with the main window, it is not resizeable

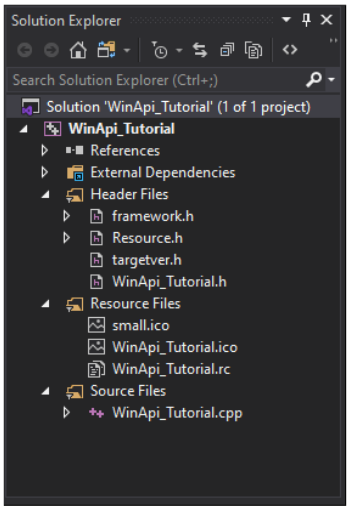



Figure 2: Solution Explorer

  - Investigate all created source files (use the Solution Explorer window, if it is not visible, show it using the menu option: View / Solution Explorer):
    - `<project_name>.h` - almost empty by default, use it as the main header file in your application
    - `Resource.h` - contains identifiers of resources, it is better to use the Resource View window in Visual Studio rather than modify it manually
    - Resource files - all files used in the application's resources: two icons and the .rc file by default
    - `<project_name>.cpp` - the main file with source code of the application

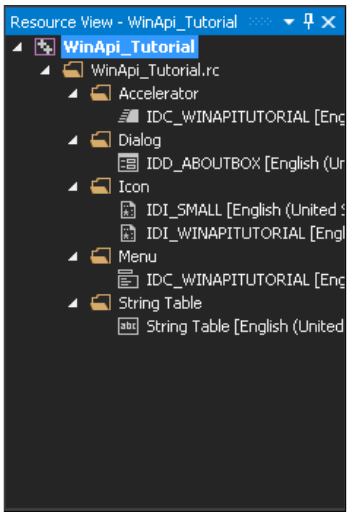



Figure 3: Resource View

- Investigate resources created by default and added to the project (use the Resource View window, if it is not visible, show it using the menu option: View / Other Windows / Resource View):
  - Accelerator - the table with keyboard shortcuts used in the application. To add a new accelerator just click below the list of used shortcuts, set the identifier of the menu item for which the keyboard shortcut is to create, right click on the new item, choose the 'Next Key Typed' option and just press the desired combination of keys
  - Dialog - dialog boxes, unlike normal windows, can be designed visually. The Toolbox window (View / Toolbox) can be used to add new so-called controls and the Properties window (View / Properties Window) can be used to alter some of their default settings (unfortunately, unlike Windows Forms, in Win32 applications some of obvious visual features can be changed only from the source code)
  - Icon - when editing an icon, be sure to modify all types of the icon (see the menu option: Image / Current Icon Image Types) - which type of the icon is used depends on the version of the system and on the context (e.g. the 16x16 type will be used for the window's icon, the 32x32 type in the Alt-Tab window, etc.)
  - Menu - use very functional editor built in Visual Studio and remember about the Properties window
  - String Table - if there is a need to display a text to the user, this text should be loaded from the string table (using the **LoadString** function) to allow to easily develop multilingual versions of the application and to set the user interface's language by the system, according to the system's settings
- Analyse the source code in the main .cpp file of the application:
  - **wWinMain** function - the entry point of the application, returning from this function means terminating the application (the `while (GetMessage(...)`x bold) loop ends when the **PostQuitMessage** function is called)
  - **MyRegisterClass** function - a helpful function which registers the class of the main window of the application (by calling the **RegisterClassEx** function)
  - **InitInstance** function - to create and show the main window
  - **WndProc** function - the function responding to all messages sent to the main window
  - **About** function - the message handling function for the About dialog box

- Getting help in Visual Studio:
  - To get information about a Win32 function, macro, or type, just set the cursor on the name and press F1 (test it for the **WNDCLASSEX**, **MAKEINTRESOURCE**, and **LoadIcon**)
  - Sometimes after pressing the F1 key, other information than expected are presented (e.g. for the **ShowWindow** function instead of information about the Win32 function, documentation of the method of one of Excel Object Model from Tools for Office is displayed)
    - To find desired information use the Index tab (menu: Help / Index) and type the name. Among all available options choose something with 'Platform SDK', 'Windows Management' or 'GDI' (e.g. try to find help about the **GetLastError** function). If something about a method of a class is displayed, you are lost, because there are no object oriented programming in Win32 API.
  - To find definition of a constant value (for example to find similar constants), set the cursor on the name and press F12 (test it for the **COLOR_WINDOW**, **WS_OVERLAPPEDWINDOW**, and **WM_PAINT**)

- Simple exercises modifying the source code generated by the wizard:
  - Modify the name of the window (hints: the string table or **CreateWindow** or **SetWindowText**)
  - Modify the background colour of the window to the colour of the active window's caption (hints: **WNDCLASSEX::hbrBackground** or **WM_ERASEBKGND**)
  - Set the default size of the window to 200 x 50 pixels (hints: **CreateWindow** or **MoveWindow**)
  - Add new menu item to the window which shows a message box (i.e. a simple window with information) (hints: resources, menu, `case WM_COMMAND` and **MessageBox**)
  - Add a keyboard shortcut for the new menu item
- More difficult exercises:
  - Move the window to the right in response for clicking on any menu item

```
case WM_COMMAND:
{
    RECT rc;
    GetWindowRect(hWnd, &rc);
    OffsetRect(&rc, 20, 0);
    MoveWindow(hWnd, rc.left, rc.top,
                rc.right - rc.left, rc.bottom - rc.top, TRUE);
}
break;

```

  - Make the window semitransparent

```
// Set WS_EX_LAYERED on this window
SetWindowLong(hWnd, GWL_EXSTYLE,
                GetWindowLong(hWnd, GWL_EXSTYLE) | WS_EX_LAYERED);
// Make this window 50% alpha
SetLayeredWindowAttributes(hWnd, 0, (255 * 50) / 100, LWA_ALPHA);
// Show this window
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

```

  - Create 9 identical windows positioned in 3 rows and 3 columns

```
int size = 150;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        hWnd = CreateWindow(szWindowClass, szTitle,
                            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                            i * 150, j * 150, 150, 150,
                            NULL, NULL, hInstance, NULL);
    }
}

```

2. Examples of handling messages

- Getting notification about changing the size of the window
  - Add the following code to the `switch (message)` statement of the **WndProc** function:

```
case WM_SIZE:
{
    // get the size of the client area
    int clientWidth = LOWORD(lParam);
    int clientHeight = HIWORD(lParam);
    // get the size of the window
    RECT rc;
    GetWindowRect(hWnd, &rc);
    // modify the caption of the window
    TCHAR s[256];
```

```
    _stprintf_s(s, 256,
            _T("Window's size: %d x %d Client area's size: %d x %d"),
            rc.right - rc.left, rc.bottom - rc.top,
            clientWidth, clientHeight);
    SetWindowText(hWnd, s);
}
    break;

```

- Explanations:
  - Both the width and height of the window's client area are passed in one parameter of the message (**LPARAM**), the **LOWORD** and **HIWORD** macros can be used to retrieve single values
  - The rectangle passed to the **GetWindowRect** function will be filled with values of the current position and size of the window
  - **TCHAR**, **_stprintf_s**, and **_T** can be used to write the source code independent on the setting of using Unicode in the application
    - To use Unicode and wide character explicitly, **wchar_t**, **swprintf_s**, and **L""** can be used:

```
wchar_t s[256];
swprintf_s(s, 256,
        L"Window's size: %d x %d   Client area's size: %d x %d",
        rc.right - rc.left, rc.bottom - rc.top,
        clientWidth, clientHeight);
SetWindowText(hWnd, s);

```

    - Unicode is supported at the system level since Windows 2000 and it is really good idea to use it (it is much easier to create multilingual applications)!
    - Setting the maximum and minimum possible size of the window
- Add the following code to the `switch (message)` statement of the **WndProc** function:

```
case WM_GETMINMAXINFO:
{
    MINMAXINFO *minMaxInfo = (MINMAXINFO*)lParam;
    minMaxInfo->ptMaxSize.x = minMaxInfo->ptMaxTrackSize.x = 500;
    minMaxInfo->ptMaxSize.y = minMaxInfo->ptMaxTrackSize.y = 200;
}
    break;

```

- Explanation: the **LPARAM** parameter of the **WM_GETMINMAXINFO** has a pointer to the **MINMAXINFO** structure, which members can be changed to set special values
- Forcing the window to be square
- Add the following code to the `switch (message)` statement of the **WndProc** function:

```
case WM_SIZING:
{
    RECT *rc = (RECT*)lParam;
    if (wParam == WMSZ_BOTTOM
        || wParam == WMSZ_BOTTOMLEFT
        || wParam == WMSZ_BOTTOMRIGHT
        || wParam == WMSZ_TOP
        || wParam == WMSZ_TOPLEFT
        || wParam == WMSZ_TOPRIGHT)
    {
        rc->right = rc->left + rc->bottom - rc->top;
    } else {
        rc->bottom = rc->top + rc->right - rc->left;
    }
}
    break;

```

- Explanation: the similar idea to the **WM_GETMINMAXINFO** message
- Using the timer
- Create a timer:

```
case WM_CREATE:
    SetTimer(hWnd, 7, 250, NULL);
    break;

```

  * Explanation: the **WM_CREATE** message is sent to the window procedure only once (when the **CreateWindow** or **CreateWindowEx** function is called); it is the first message when the handle of the window (**HWND**) is available and it is great place to do some initialization of the window (the window is already created but not visible)
  * Do something in response for the **WM_TIMER** message, for example:

```
case WM_TIMER:
{
    if (wParam == 7) //check timer id
    {
        RECT rc;
        // get the center of the work area of the system
        SystemParametersInfo(SPI_GETWORKAREA, 0, &rc, 0);
        int centerX = (rc.left + rc.right + 1) / 2;
        int centerY = (rc.top + rc.bottom + 1) / 2;
        // get current size of the window
        GetWindowRect(hWnd, &rc);
        int currentSize = max(rc.right - rc.left, rc.bottom - rc.top);
        // modify size of the window
        currentSize += stepSize;
        if (currentSize >= maxSize) {
            stepSize = -abs(stepSize);
        } else if (currentSize <= minSize) {
            stepSize = abs(stepSize);
        }
        MoveWindow(hWnd, centerX - currentSize / 2,
                         centerY - currentSize / 2,
                         currentSize, currentSize, TRUE);
        }
}
    break;

```

  * The above code needs some declarations:

```
const int minSize = 200;
const int maxSize = 400;
static int stepSize = 10;

```

  * (be sure to understand the importance of the static keyword)
  * Explanations:
    - The **SystemParametersInfo** function is very useful when there is a need to get or set values of some system parameters
    - There are many functions for changing window's size and/or position, **MoveWindow** is the simplest one

3. Examples of working with the mouse and keyboard

- Using mouse messages
  - Add the following function:

```
void GetTextInfoForMouseMsg(WPARAM wParam, LPARAM lParam,
                            const TCHAR *msgName, TCHAR *buf, int bufSize)
{
    short x = (short)LOWORD(lParam);
    short y = (short)HIWORD(lParam);
    _stprintf_s(buf, bufSize, _T("%s x: %d, y: %d, vk:"),
                msgName, x, y);
    if ((wParam == MK_LBUTTON) != 0) {
        _tcscat_s(buf, bufSize, _T(" LEFT"));
    }
    if ((wParam == MK_MBUTTON) != 0) {
        _tcscat_s(buf, bufSize, _T(" MIDDLE"));
    }
    if ((wParam == MK_RBUTTON) != 0) {
        _tcscat_s(buf, bufSize, _T(" RIGHT"));
    }
}

```

– Add the following code to the **WndProc** function:

```
case WM_LBUTTONDOWN:
    GetTextInfoForMouseMsg(wParam, lParam, _T("LBUTTONDOWN"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
case WM_LBUTTONUP:
    GetTextInfoForMouseMsg(wParam, lParam, _T("LBUTTONUP"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
case WM_RBUTTONDOWN:
    GetTextInfoForMouseMsg(wParam, lParam, _T("RBUTTONDOWN"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
case WM_RBUTTONUP:
    GetTextInfoForMouseMsg(wParam, lParam, _T("RBUTTONUP"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
case WM_LBUTTONDBLCLK:
    GetTextInfoForMouseMsg(wParam, lParam, _T("LBUTTONDBLCLK"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
case WM_MBUTTONDBLCLK:
    GetTextInfoForMouseMsg(wParam, lParam, _T("MBUTTONDBLCLK"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
case WM_RBUTTONDBLCLK:
    GetTextInfoForMouseMsg(wParam, lParam, _T("RBUTTONDBLCLK"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    break;

```

– Be sure to add declarations of variables in the **WndProc** function:

```
const int bufSize = 256;
TCHAR buf[bufSize];

```

- Run the application and test changing the caption of the window when left, middle, or right mouse button is pressed or released
- Explanations:
  * **WM_L...** messages are for the left mouse button, **WM_M...** for middle, and **WM_R...** for the right one
  * Notifications about pressing (**WM_xBUTTONDOWN**) and releasing (**WM_xBUTTONUP**) are sent
  * The **WM_xBUTTONDBLCLK** message is sent when the user use the double click. To get the notification, the **CS_DBLCLKS** style must be applied to **WNDCLASSEX::style** in calling the **RegisterClassEx** function:

```
wcex.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;

```

    When the **CS_DBLCLKS** style is applied, the following sequence of messages is sent when the user use the double click:
    · **WM_xBUTTONDOWN**
    · **WM_xBUTTONUP**
    · **WM_xBUTTONDBLCLK**
    · **WM_xBUTTONUP**
    When there is no this style, the **WM_xBUTTONDBLCLK** message is not sent:
    · **WM_xBUTTONDOWN**
    · **WM_xBUTTONUP**
    · **WM_xBUTTONDOWN**
    · **WM_xBUTTONUP**

- Client and screen coordinations
  - Note, that the above example works only when the user uses the mouse in the client area of the window. Position passed in the **LPARAM** parameter is in the client area coordinations, to transform between the client area and screen coordinations, the **ScreenToClient** and **ClientToScreen** function can be used, e.g.:

```
void GetTextInfoForMouseMsg(HWND hWnd, WPARAM wParam, LPARAM lParam,
    const TCHAR *msgName, TCHAR *buf, int bufSize)
{
    short x = (short)LOWORD(lParam);
    short y = (short)HIWORD(lParam);
    POINT pt = {x, y};
    ClientToScreen(hWnd, &pt);
    _stprintf_s(buf, bufSize,
                _T("%s x: %d, y: %d, (sx: %d, sy: %d) vk:"),
                msgName, x, y, pt.x, pt.y);

```

- Using mouse capture
  - Note also, that the **WM_xBUTTONUP** message is sent only if the user released the mouse button in the client area of the window, to be sure that this message will be sent wherever the button is released, use the mouse capture:

```
case WM_LBUTTONDOWN:
    GetTextInfoForMouseMsg(hWnd, wParam, lParam, _T("LBUTTONDOWN"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
    SetCapture(hWnd);
    break;
case WM_LBUTTONUP:
    ReleaseCapture();
    GetTextInfoForMouseMsg(hWnd, wParam, lParam, _T("LBUTTONUP"),
                           buf, bufSize);
    SetWindowText(hWnd, buf);
```

```
12      break;
13
```

- Note, that in this case negative values for mouse coordinations can be sent, so it is very important to use signed integer type to use this value: `x = (short)LOWORD(lParam);` (without casting to short type big positive values would be used)

- Changing mouse cursor
  - Add the following code to the **WndProc** function:

```
case WM_CREATE:
    cursor = LoadCursor(NULL, IDC_HAND);
    break;
case WM_SETCURSOR:
    SetCursor(cursor);
    return TRUE;

```

  - Add also a declaration of the cursor variable (if it is a local variable, the static keyword must be used)
  - static `HCURSOR cursor = NULL;`
  - Explanations:
    - To change the mouse cursor, the **WM_SETCURSOR** message and the **SetCursor** function can be used
    - The **SetCursor** function needs an **HCURSOR** parameter which represents a cursor. It can be loaded from resources or from the system using the **LoadCursor** function
    - The above code changes the cursor for all areas of the window
    - It is possible to use simpler code: `SetCursor(LoadCursor(NULL, IDC_HAND));` but it would be less efficient
    - The NULL value as the first parameter of the **LoadCursor** method means, that the cursor must be loaded from the system (not from resources of the application)
    - To use a cursor from applications resources:
      - Add the cursor to resources (using the Resource View window in Visual Studio)
      - Pass the **hInst** variable as the first parameter for the **LoadCursor** function, e.g. `LoadCursor(hInst, IDC_MYCURSOR)`

- Moving the window by dragging by any point of the window
  - Add the following code to the **WndProc** function:

```
case WM_NCHITTEST:
    return HTCAPTION;

```

  - Note, that in real applications the code should be more complicated (e.g. to allow the user to close the window using the close button or just to use the menu). To close the window press Alt+F4.

- Using keyboard messages
  - Add the following function:

```
void GetTextInfoForKeyMsg(WPARAM wParam, const TCHAR *msgName,
                          TCHAR *buf, int bufSize)
{
    static int counter = 0;
    counter++;
    _stprintf_s(buf, bufSize, _T("%s key: %d (counter: %d)"), msgName,
    wParam, counter);
}

```

  - Add the following code to the **WndProc** function:

```
case WM_KEYDOWN:
    GetTextInfoForKeyMsg(wParam, _T("KEYDOWN"), buf, bufSize);
    SetWindowText(hWnd, buf);
    break;
```

```
case WM_KEYUP:
    GetTextInfoForKeyMsg(wParam, _T("KEYUP"), buf, bufSize);
    SetWindowText(hWnd, buf);
    break;

```

- Run the application and test changing the caption of the window when keys are pressed and released
- Note, that because of the autorepeat feature, more than one **WM_KEYDOWN** message may be posted before a **WM_KEYUP** message is posted (the counter added at the end of the text is being increased when the key is being pressed)
- Virtual key codes are constant values which identify keys on the keyboard
- (see https://docs.microsoft.com/en-us/windows/desktop/inputdev/virtual-key-codes)
- When the character is important instead of the virtual key code, the **WM_CHAR** message can be used:

```
case WM_CHAR:
    _stprintf_s(buf, bufSize, _T("WM_CHAR: %c"), (TCHAR)wParam);
    SetWindowText(hWnd, buf);
    break;

```

- The **WM_CHAR** message automatically recognizes the state of shift and caps lock keys and transforms the character

4. Examples of drawing using GDI

- The code for drawing should be placed in responding for the **WM_PAINT** message
- Simple drawing:
  - Drawing texts
    - The simplest way to draw a text:

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    TCHAR s[] = _T("Hello world!");
    TextOut(hdc, 0, 0, s, (int)_tcslen(s));
    EndPaint(hWnd, &ps);
}
    break;

```

    - More options are available when the **DrawText** function is used:

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    TCHAR s[] = _T("Hello world!");
    RECT rc;
    GetClientRect(hWnd, &rc);
    DrawText(hdc, s, (int)_tcslen(s), &rc,
                DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    EndPaint(hWnd, &ps);
}
    break;

```

    - Explanations:
      - The **GetClientRect** function gets the rectangle of the client area of the window (the left and top members are always set to 0)
      - Using the last parameter of the **DrawText** function, the alignment of the text can be changed
  - Using pens

* The **HPEN** is a handle to a pen and represents a pen in drawing using GDI

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    HPEN pen = CreatePen(PS_SOLID, 2, RGB(255, 0, 0));
    HPEN oldPen = (HPEN)SelectObject(hdc, pen);
    MoveToEx(hdc, 0, 0, NULL);
    LineTo(hdc, 100, 100);
    SelectObject(hdc, oldPen);
    DeleteObject(pen);
    EndPaint(hWnd, &ps);
}
    break;

```

* Explanations:
  - GDI always uses current objects (one pen, one brush, one font, etc.) to draw, the object can be selected as current on the device context (HDC) using the **SelectObject** function.
  - The colour contains 3 values: for the red, green, and blue. The RGB macro can be used to set specific colour, its parameters are integer values from the range [0..255])
  - VERY IMPORTANT: All created GDI objects must be destroyed. The pen created using the **CreatePen** function must be destroyed using the **DeleteObject**.
  - VERY IMPORTANT: Objects selected as current on the device context cannot be destroyed (so always remember the old object and restore it before calling the **DeleteObject** function)

– Using brushes

* Brushes are similar to pens, they are used when something must be filled, e.g. a rectangle:

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    HPEN pen = CreatePen(PS_DOT, 1, RGB(255, 0, 0));
    HPEN oldPen = (HPEN)SelectObject(hdc, pen);
    HBRUSH brush = CreateSolidBrush(RGB(0, 128, 0));
    HBRUSH oldBrush = (HBRUSH)SelectObject(hdc, brush);
    Rectangle(hdc, 20, 20, 120, 120);
    SelectObject(hdc, oldPen);
    DeleteObject(pen);
    SelectObject(hdc, oldBrush);
    DeleteObject(brush);
    EndPaint(hWnd, &ps);
}
    break;

```

– Using fonts

```

case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    TCHAR s[] = _T("Hello world!");
    HFONT font = CreateFont(
            -MulDiv(24, GetDeviceCaps(hdc, LOGPIXELSY), 72), // Height
            0,                              // Width
            0,                              // Escapement
            0,                              // Orientation
            FW_BOLD,                        // Weight
            false,                          // Italic
            FALSE,                          // Underline
```

```
            0,                          // StrikeOut
            EASTEUROPE_CHARSET,         // CharSet
            OUT_DEFAULT_PRECIS,         // OutPrecision
            CLIP_DEFAULT_PRECIS,        // ClipPrecision
            DEFAULT_QUALITY,            // Quality
            DEFAULT_PITCH | FF_SWISS,   // PitchAndFamily
            _T("Verdana"));             // Facename
    HFONT oldFont = (HFONT)SelectObject(hdc, font);
    RECT rc;
    GetClientRect(hWnd, &rc);
    DrawText(hdc, s, (int)_tcslen(s), &rc,
                DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    SelectObject(hdc, oldFont);
    DeleteObject(font);
    EndPaint(hWnd, &ps);
}
    break;

```

- Using bitmaps
  - Add a bitmap to resources (right click on the project in the Solution Explorer window in Visual Studio, Add / Resource, and choose Bitmap in the dialog box)
  - Draw something on the bitmap (there is a simple image editor built in Visual Studio)
  - Save the bitmap (note, that a .bmp file is created)
  - Check an identifier of the bitmap in the Resource View window (it should be **IDB_BITMAP1** by default)
  - Copy the following code for the **WM_PAINT** message

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    TCHAR s[] = _T("Hello world!");
    HBITMAP bitmap = LoadBitmap(hInst,
                                MAKEINTRESOURCE(IDB_BITMAP1));
    HDC memDC = CreateCompatibleDC(hdc);
    HBITMAP oldBitmap = (HBITMAP)SelectObject(memDC, bitmap);
    BitBlt(hdc, 0, 0, 48, 48, memDC, 0, 0, SRCCOPY);
    StretchBlt(hdc, 200, 100, -200, 100, memDC,
                    0, 0, 48, 48, SRCCOPY);
    SelectObject(memDC, oldBitmap);
    DeleteObject(bitmap);
    DeleteDC(memDC);
    EndPaint(hWnd, &ps);
}
    break;

```

  - Explanations:
    - There is no function for drawing a bitmap on a device context, another device context must be used
    - To create a memory device context use the **CreateCompatibleDC** function. Each created device context must be destroyed using the **DeleteDC** function
    - The **BitBlt** function copies a bitmap from one device context to another one, the **StretchBlt** function allows to resize the bitmap

- Drawing in response for messages other than **WM_PAINT**
  - Copy the following code for the **WM_LBUTTONDOWN** message:

```
case WM_LBUTTONDOWN:
{
    HDC hdc = GetDC(hWnd);
```

```
    HBRUSH brush = CreateSolidBrush(RGB(128, 128, 0));
    HBRUSH oldBrush = (HBRUSH)SelectObject(hdc, brush);
    short x = (short)LOWORD(lParam);
    short y = (short)HIWORD(lParam);
    const int rad = 5;
    Ellipse(hdc, x - rad, y - rad, x + rad, y + rad);
    SelectObject(hdc, oldBrush);
    DeleteObject(brush);
    ReleaseDC(hWnd, hdc);
}
    break;

```

- Explanations:
  - The **BeginPaint** and **EndPaint** functions can be used only in response for the **WM_PAINT** message, in all other cases the **GetDC** and **ReleaseDC** functions must be used
  - When the window is refreshed (e.g. after changing its size), only the code for the **WM_PAINT** message is called, this is the reason why the content is cleared

- Invalidating and updating the window
  - A better solution of previous example (with storing clicked points in the list):

```
#include <list>
using namespace std;
static list<POINT> pointsList;
...
case WM_LBUTTONDOWN:
{
    POINT pt;
    pt.x = (short)LOWORD(lParam);
    pt.y = (short)HIWORD(lParam);
    pointsList.push_back(pt);
    InvalidateRect(hWnd, NULL, TRUE);
}
break;
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    HBRUSH brush = CreateSolidBrush(RGB(128, 128, 0));
    HBRUSH oldBrush = (HBRUSH)SelectObject(hdc, &brush);
    list<POINT>::const_iterator iter = pointsList.begin();
    while (iter != pointsList.end()) {
        POINT pt = *iter;
        const int rad = 5;
        Ellipse(hdc, pt.x - rad, pt.y - rad, pt.x + rad, pt.y + rad);
        iter++;
    }
    SelectObject(hdc, oldBrush);
    DeleteObject(brush);
    EndPaint(hWnd, &ps);
}
    break;

```

  - Explanations:
    - The **InvalidateRect** function sets the specified rectangle (or full client area when the NULL value is used) as a region that must be redrawn and inserts the **WM_PAINT** message to the message queue
    - To force the window to redraw as soon as possible, call the **UpdateWindow** functions immediately after calling the **InvalidateRect** function

- Flicker-free drawing

– Copy the following code for the **WM_PAINT** message and resize the window - it flickers

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    RECT rc;
    GetClientRect(hWnd, &rc);
    HBRUSH oldBrush = (HBRUSH)SelectObject(hdc,
                                (HBRUSH)GetStockObject(GRAY_BRUSH));
    Rectangle(hdc, 0, 0, rc.right, rc.bottom);
    SelectObject(hdc, (HBRUSH)GetStockObject(BLACK_BRUSH));
    const int margin = 50;
    Rectangle(hdc, margin, margin,
                    rc.right - margin, rc.bottom - margin);
    SelectObject(hdc, oldBrush);
    EndPaint(hWnd, &ps);
}
    break;

```

– There are two reasons of flickering, the first one is the default background of the window. To disable drawing the background, set the NULL value for the background brush or use the **WM_ERASEBKGND** message:

```
case WM_ERASEBKGND:
    return 1;

```

– The second reason of flickering is drawing figures one after another (firstly the grey rectangle is drawn and secondly the black one). For two rectangle it is easy to modify the code to avoid such covering, but in general the only way to avoid flickering is off-screen drawing (i.e. drawing using a memory bitmap):

```
static HDC offDC = NULL;
static HBITMAP offOldBitmap = NULL;
static HBITMAP offBitmap = NULL;
case WM_CREATE:
{
    HDC hdc = GetDC(hWnd);
    offDC = CreateCompatibleDC(hdc);
    ReleaseDC(hWnd, hdc);
    break;
}
case WM_SIZE:
{
    int clientWidth = LOWORD(lParam);
    int clientHeight = HIWORD(lParam);
    HDC hdc = GetDC(hWnd);
    if (offOldBitmap != NULL) {
        SelectObject(offDC, offOldBitmap);
    }
    if (offBitmap !=- NULL) {
        DeleteObject(offBitmap);
    }
    offBitmap = CreateCompatibleBitmap(hdc, clientWidth,clientHeight);
    offOldBitmap = (HBITMAP)SelectObject(offDC, offBitmap);
    ReleaseDC(hWnd, hdc);
}
    break;
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    RECT rc;
```

```
32      GetClientRect(hWnd, &rc);
33      HBRUSH oldBrush = (HBRUSH)SelectObject(offDC,
34                  (HBRUSH)GetStockObject(GRAY_BRUSH));
35      Rectangle(offDC, 0, 0, rc.right, rc.bottom);
36      SelectObject(offDC, (HBRUSH)GetStockObject(BLACK_BRUSH));
37      const int margin = 50;
38      Rectangle(offDC, margin, margin,
39                          rc.right - margin, rc.bottom - margin);
40      SelectObject(offDC, oldBrush);
41      BitBlt(hdc, 0, 0, rc.right, rc.bottom, offDC, 0, 0, SRCCOPY);
42      EndPaint(hWnd, &ps);
43  }
44      break;
45  case WM_ERASEBKGND:
46      return 1;
47  case WM_DESTROY:
48      if (offOldBitmap != NULL) {
49          SelectObject(offDC, offOldBitmap);
50      }
51      if (offDC != NULL) {
52          DeleteDC(offDC);
53      }
54      if (offBitmap != NULL) {
55          DeleteObject(offBitmap);
56      }
57      PostQuitMessage(0);
58      break;
59
```

---

* Explanations:
  - In response for the **WM_CREATE** message, a memory device context is created (it has no dependency on the size of the window, so there is no need to recreate it during responding for the **WM_SIZE** message)
  - In response for the **WM_SIZE** message, a memory bitmap is created (so whenever the window changes its size, the bitmap is recreated - that's why resizing of the window is much slower for the above code)
    (a) If such slow resizing is unacceptable, a bitmap with the biggest possible size (use desktop's size) should be created in response for the **WM_CREATE** message
  - In the **WM_PAINT** message everything is painted on the memory device context and at the end copied to the screen device context using the **BitBlt** function
* This solution is not perfect - there is no response for changing the image depth of the display (in bits per pixel). The **WM_DISPLAYCHANGE** message should be used.

5. Additional task:
   - Create window inside other window (check out **WS_CHILD** flag for **CreateWindow** function)
   - Use timer to change position or/and size of child window
6. More references and useful links:
   - https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-api-list
   - http://cpp0x.pl/kursy/Kurs-WinAPI-C++/167
   - www.winprog.org/tutorial/
   - Debugging tutorials
     - Short debugging tutorial: http://www.dotnetperls.com/debugging
     - More debugging tips & tricks: https://blogs.msdn.microsoft.com/brunoterkaly/2009/09/28/the-art-of-debugging-a-developers-best-friend-intro-lesson-1-repost/