Programming in Graphical Environment

Windows API Lecture 3

Paweł Aszklar
P.Aszklar@mini.pw.edu.pl

Faculty of Mathematics and Information Science
Warsaw Univeristy of Technology

Warsaw 2021

Keyboard Input

- Keyboard input sent to foreground window's thread's message queue
- Recipient: focal window (or active window, if none have focus) of that thread
- Received as keystroke messages `WM_KEYDOWN`, `WM_KEYUP`, `WM_SYSKEYDOWN`, `WM_SYSKEYUP`
- `SYS` variants generated for:
  - Alt and any key pressed while it is down
  - F10 and any following key
  - every key if no window has focus
- `SYS` keystrokes usually need to be passed to `DefWindowProcW` (to handle menu, system shortcuts, etc.)
- Keystroke messages inform about particular keys on keyboard (via *virtual-key codes*), not characters user intended to type

Auto-Repeat Feature

- While key held down, `WM_KEYDOWN` or `WM_SYSKEYDOWN` send repeatedly at certain interval
- If not removed from queue fast enough, auto-repeating messages for a given key-press will be combined

Configured via `SystemParametersInfoW`

- `SPI_GETKEYBOARDDELAY`, `SPI_SETKEYBOARDDELAY` — delay to first repeated message (between 0 — $\sim 250ms$, and 3 — $\sim 1s$)
- `SPI_GETKEYBOARDSPEED`, `SPI_SETKEYBOARDSPEED` — auto-repeat speed (between 0 — ~ 2.5 times per second, and 31 — ~ 30 times per second)

Character Messages

- Conversion done by passing messages to `TranslateMessage`, which posts (if message translated):

Message	Translated from
`WM_CHAR`	`WM_KEYDOWN`
`WM_SYSCHAR`	`WM_SYSKEYDOWN`
`WM_DEADCHAR`	`WM_KEYUP`
`WM_SYSDEADCHAR`	`WM_SYSKEYDOWN`

- `WM_DEADCHAR`, `WM_SYSDEADCHAR` for characters that combine with next one to form composite character (e.g. umlaut on German keyboard)

Keyboard Message Parameters

`wParam`

- `KEY` messages — virtual-key code (ASCII codes for digits, uppercase letters, various `VK_` constants, e.g. `VK_RETURN`, `VK_NUMPAD0`, etc.)
- `CHAR` messages — character code (`wchar_t` for Unicode windows)

`lParam`

- `LOWORD(lParam)` — how many combined auto-repeated messages it represents (usually 1)
- `HIWORD(lParam)&0xFF` — scan-code, driver-dependent, usually ignored
- `HIWORD(lParam)&KF_ALTDOWN` — Alt pressed flag
- `HIWORD(lParam)&KF_EXTENDED` — extended flag, differentiates duplicated keyboard keys (e.g. left and right control, return and numpad return, etc.)
- `HIWORD(lParam)&KF_REPEAT` — previous key state, i.e. if key was down before this message (0 for first `DOWN` message; 1 for `UP` and auto-repeated `DOWN` messages)
- `HIWORD(lParam)&KF_UP` — transition flag (0 for `DOWN`, 1 `UP` messages)

(for `CHAR` messages, copy from `DOWN` or `UP` message it was translated from)

Key State

Application can check current key states (via their virtual-key code):

- Input-synchronized, i.e. at the time of last message retrieved from queue:

```
SHORT GetKeyState(int virtualKeyCode);
BOOL GetKeyboardState(BYTE keyStates[256]);
BOOL SetKeyboardState(BYTE keyStates[256]);
```

- Asynchronous, interrupt-level, as physically pressed at time of call:

```
SHORT GetAsyncKeyState(int virtualKeyCode);
```

- If `BYTE state` — key state (return by `GetKeyState`, `GetAsyncKeyState` or `keyStates` element)
  - `state&0x80` — flag (high bit) set if key is down
  - `state&0x01` — flag (low bit) set if key is toggled (toggleable keys only: NumLock, CapsLock, …)

Media Keys

Some special keys (media, power management keys, etc.) additionally generate `WM_APPCOMMAND`

- `wParam` — handle to window that received input (recipient or child)
- `lParam`:
  - `GET_APPCOMMAND_LPARAM(lParam)` — specific command type, eg.: `APPCOMMAND_BROWSER_HOME`, `APPCOMMAND_MEDIA_PLAY`, `APPCOMMAND_VOLUME_DOWN`, …
  - `GET_DEVICE_LPARAM(lParam)` — device which generated input (usually `FAPPCOMMAND_KEY` for keyboard)
  - `GET_KEYSTATE_LPARAM(lParam)` — down-state of shift, control keys, mouse buttons (see next slide here)
- Return `TRUE` if message processed
- Generated by `DefWindowProcW` in response to keys, mouse x-buttons, …
- When passed to `DefWindowProcW` propagated to parent window

System-Wide Hot-Keys

`BOOL RegisterHotKey(HWND hWnd, int id, UINT modFlags, UINT vkCode)`

- Registers system wide hot-key
- `hWnd` — recipient window (can be `nullptr`)
- `id` — hot-key id (application-defined, must be between `0x0000` and `0xBFFF`)
- `vkCode` — virtual-key code of key that needs to be pressed
- `modFlags` — modifier keys that must be held (Alt, Ctrl, ⇧, ⊞); if auto-repeat enabled
- To remove call `UnregisterHotKey`

`WM_HOTKEY`

- Received when hot-key pressed
- `wParam` — hot-key id
- `lParam`:
  - `HIWORD(lParam)` — virtual-key code
  - `LOWORD(lParam)` — flags of modifier keys held

Key Code Conversion

Besides `TranslateMessage`, various other function can translate between key codes and text:

- `int GetKeyNameW(LONG lParam, LPWSTR str, int len)` — retrieves string representing key
- `MapVirtualKeyW` — converts virtual-key code⇆scan-code, virtual-key code→character code
- `ToAscii`, `ToUnicode` — (virtual-key code, scan-code and keyboard state)→characters
- `ToAsciiEx`, `ToUnicodeEx` – as above, but for specific keyboard layout

Keyboard layout (locale identifier):

- determines input language, mapping between physical keys, virtual key-codes and generated characters
- `GetKeyboardLayout`, `ActivateKeyboardLayout` — access current layout
- `GetKeyboardLayoutList` — available layouts
- `GetKeyboardLayoutNameW` — string representing layout
- `LoadKeyboardLayoutW`, `UnloadKeyboardLayout` — manage custom layouts

Caret

- Temporary resource, indicates window has keyboard focus
- One caret per message queue, only one thread window can own it (just like focus)
- Blinking rectangle (or bitmap), inverts pixels under it
- `BOOL CreateCaret(HWND hWnd, HBITMAP bmp, int w, int h)`
  - Creates caret owned by `hWnd`
  - `w`, `h` — caret width and height (pass 0 to use default)
`bmp`	Caret Shape	Underlying Pixels
0	rectangle	invert all (*black* caret)
1	rectangle	invert every other (*grey* caret)
bitmap handle	bitmap	XOR with bitmap pixels
- `HideCaret`, `ShowCaret` temporarily hides/restores caret (uses hide counter)
- `DestroyCaret` destroys it
- Create caret on `WM_SETFOCUS`, destroy on `WM_KILLFOCUS`
- `GetCaretPos`, `SetCaretPos` to modify caret position (its top-left corner in owner's client coordinates).

Mouse Input

- Mouse represented by cursor on screen
- Input represents mouse movement, change of button states, wheel movement
- Messages usually sent to window under cursor (under one selected pixel, *hot spot*)
- Supports up to 5 buttons (left, right, middle, two x-buttons) and 2 wheels (vertical, horizontal)
- Call `GetSystemMetrics` to check capabilities:
  - `SM_MOUSEPRESENT` — if mouse installed
  - `SM_CMOUSEBUTTON` — number of buttons available
  - `SM_MOUSEWHEELPRESENT` — if vertical wheel available
  - `SM_MOUSEHORIZONTALWHEELPRESENT` — if horizontal wheel available
  - `SM_SWAPBUTTON` — if left and right buttons swapped

Mouse Messages

- Button Messages: `WM_[NC|][L|R|M|X]BUTTON[DOWN|UP|DBLCLK]`
  (e.g. `WM_LBUTTONDOWN`, `WM_NCXBUTTONDBLCLK`, …)
- One set of messages for both x-buttons
- X-Button messages propagated to parent by `DefWindowProcW`
- `DefWindowProcW` sends `WM_APPCOMMAND` on `WM_XBUTTONUP`, `WM_NCXBUTTONUP`
  Usually: browser forward, back command, device `FAPPCOMMAND_MOUSE`. See previous slide here
- Move message: `WM_MOUSEMOVE`, `WM_NCMOUSEMOVE`
- Wheel messages: `WM_MOUSEWHEEL`, `WM_MOUSEHWHEEL`
- Mouse tracking events: `WM_MOUSEHOVER`, `WM_NCMOUSEHOVER`, `WM_MOUSELEAVE`, `WM_NCMOUSELEAVE`
- Client and Non-Client Area:
  - `NC` prefixed if event over window's frame (non-client area)
  - otherwise, event over window's client area
  - Non-client area messages need to be passed to `DefWindowProcW`
- Double-click and tracking events must be enabled (see next slide here)

Mouse Message Parameters

- `lParam` — mouse position, use `GET_X_LPARAM`, `GET_Y_LPARAM` to extract
  - For non-client messages, position in screen coordinates
  - For client area messages, position in window's client coordinates
- `wParam`
  - `GET_KEYSTATE_WPARAM(wParam)` (client area messages) — modifier keys and mouse buttons down-state flags:

`MK_CONTROL`	Ctrl	`MK_SHIFT`	⇧
`MK_LBUTTON`	left mouse button	`MK_RBUTTON`	right mouse button
`MK_MBUTTON`	middle mouse button	`MK_XBUTTON1`	x-button 1
`MK_XBUTTON2`	x-button 2		

  - `GET_NCHITTEST_WPARAM(wParam)` (non-client area messages) — hit-test value (see next slide)
  - `GET_WHEEL_DELTA_WPARAM(wParam)` (wheel messages) — wheel rotation distance, usually multiples of `WHEEL_DELTA` (120).
  - `GET_XBUTTON_WPARAM(wParam)` (x-button messages) — `XBUTTON1` or `XBUTTON2`
- Exceptions: `lParam`, `wParam` unused in `WM_MOUSELEAVE`, `WM_NCMOUSELEAVE`

Mouse Hit-Testing

`WM_NCHITTEST`

- Sent only when mouse in client area (frame hit-testing done automatically by the system)
- Allows client area regions to be treated as part of window frame
- `lParam` — mouse position in screen coordinates (use `GET_X_LPARAM`, `GET_Y_LPARAM` to extract)
- Return value indicates position over window elements, examples:
  - `HTCLIENT` — windows client area
  - `HTTOP`, `HTLEFT`, `HTBOTTOMRIGHT`, …— edges/corners of window's sizing border
  - `HTMINBUTTON`, `HTMAXBUTTON`, `HTCLOSE`, `HTHELP` — caption bar buttons
  - `HTBORDER`, `HTCAPTION` — non-sizing border, caption bar
  - `HTSYSMENU` — in system menu or over child's close button
  - `HTVSCROLL`, `HTHSCROLL`, `HTMENU` — window scroll bars, main menu
- Subsequent mouse message will be non-client unless `HTCLIENT` returned.
- Automatic behaviour handling of regions pretending to be non-client areas limited
  - Only works if real corresponding frame element visible
  - Doesn't properly simulate caption buttons, etc.

Opt-In Messages

Double-click messages:

- Enabled if window class has `CS_DBLCLKS` style
- Arrive in a sequence with other messages: `DOWN`→`UP`→`DBLCLK`→`UP`

Mouse tracking messages:

- `TrackMouseEvent` to selectively enable, cancel or query current tracked mouse events
- `HOVER` messages — mouse remains stationary for a time over client/non-client area
- `LEAVE` messages — mouse leaves client/non-client area
- When `HOVER` or `LEAVE` message arrives, tracking is disabled.
- Call `TrackMouseEvent` again (e.g. in mouse move message) to re-enable

Mouse Activation

`WM_MOUSEACTIVATE`

- Sent when user clicks inactive window or a child of inactive window
- Before `WM_ACTIVATE`, etc.
- `lParam` — top-level parent window handle
- `HIWORD(wParam)` — mouse message type identifier of event that caused activation
- `LOWORD(wParam)` — hit-test result for mouse position where event occurred
- Return
  - `MA_NOACTIVATE`, `MA_NOACTIVATEANDEAT` to prevent window activation
  - `MA_ACTIVATEANDEAT`, `MA_NOACTIVATEANDEAT` to prevent further processing
- `DefWindowProcW` propagates message to parent window.

Mouse Capture

`HWND SetCapture(HWND hWnd)`

- Sets mouse capture to window
- Window will receive all mouse messages, even if cursor moves outside of it
- Only foreground window can capture mouse, when user clicks window of another process moving it to foreground, capture is automatically released.
- Useful for dragging: capturing on `BUTTONDOWN` guarantees `BUTTONUP` will be received, even outside of window.

`BOOL ReleaseCapture()` — releases capture held by a current thread's window (if any)
`HWND GetCapture()` — retrieves current thread's window holding capture (if any)
`WM_CAPTURECHANGED` — received when window is loosing capture (user action or `ReleaseCapture`)

Mouse Cursor

- Show/hide cursor (internal shown/hidden counter, visible ≥ 0):
  `int ShowCursor(BOOL show)`
- Current mouse cursor position:
  `BOOL SetCursorPos(int x, int y), BOOL GetCursorPos(POINT* p)`
- Mouse cursor confinement (also affects future `SetCursorPos`):
  `BOOL ClipCursor(const RECT* rc), BOOL GetClipCursor(RECT * rc)`
- `BOOL GetCursorInfo(CURSORINFO * pci)`
  - `cbSize` — `sizeof(CURSORINFO)`, must be set before call
  - `flags` — 0 if cursor hidden; `CURSOR_SHOWING` if show; `CURSOR_SUPPRESSED` forced invisible (e.g. when using touch or pen input)
  - `hCursor` — current cursor shape
  - `ptScreenPos` — current cursor position

```
struct CURSORINFO{
    DWORD   cbSize;
    DWORD   flags;
    HCURSOR hCursor;
    POINT   ptScreenPos;
};
```

- For customization of cursor appearance, see next slides ▸ here

Mouse Configuration

`SystemParametersInfoW`

- `SPI_GETMOUSESPEED`, `SPI_SETMOUSESPEED` — mouse speed from 1 to 20
- `SPI_GETMOUSE`, `SPI_SETMOUSE` — mouse acceleration parameters
- `SPI_GETMOUSEHOVERTIME`, `SPI_GETMOUSEHOVERWIDTH`, `SPI_GETMOUSEHOVERHEIGHT`, `SPI_SETMOUSEHOVERTIME`, `SPI_SETMOUSEHOVERWIDTH`, `SPI_SETMOUSEHOVERHEIGHT`
  period of time and rectangle within which mouse must stay to generate mouse hover event
- `SPI_GETMOUSEWHEELROUTING`, `SPI_SETMOUSEWHEELROUTING` — if mouse wheel button events received by window under cursor or window with focus
- `SPI_GETWHEELSCROLLCHARS`, `SPI_GETWHEELSCROLLLINES`, `SPI_SETWHEELSCROLLCHARS`, `SPI_SETWHEELSCROLLLINES` — characters/lines to scroll for `WHEEL_DELTA`
- `SPI_SETDOUBLECLICKTIME`, `SPI_SETDOUBLECLKWIDTH`, `SPI_SETDOUBLECLKHEIGHT`
  max time and distance between clicks to generate double-click event
  (also `GetDoubleClickTime`, `SetDoubleClickTime`)
- `SPI_SETMOUSEBUTTONSWAP` — if left/right mouse buttons swapped (also `SwapMouseButton`)

Simulating Input

- `UINT SendInput(UINT count, INPUT inputs[], int size)`
  - `count` — number of elements in `inputs`
  - `inputs` — array of input events to generate
  - `size` — `sizeof(INPUT)`

```
struct INPUT {
    DWORD type; // INPUT_MOUSE, INPUT_KEYBOARD or INPUT_HARDWARE
    union {
        MOUSEINPUT    mi;
        KEYBDINPUT    ki;
        HARDWAREINPUT hi;
    };
};
```

- Legacy functions: `keybd_event`, `mouse_event`

Simulating Keyboard Input

- `wVk` — virtual-key code
- `wScan` — scan-code or Unicode character
- `time` — event timestamp in ms (0 to auto-generate)
- `dwExtraInfo` — value associated with event (`GetMessageExtraInfo` to retrieve)
- `flags`
  - `KEYEVENTF_KEYUP` — key is released (otherwise pressed)
  - `KEYEVENTF_EXTENDEDKEY` — extended (duplicate) key
  - `KEYEVENTF_SCANCODE` — `wScan` as scan-code identifies key, `wVk` ignored
  - `KEYEVENTF_UNICODE` — Unicode character `wScan` as `VK_PACKET` keystroke
    `wVk` must be 0 and `KEYEVENTF_KEYUP` must be set.
    mostly for non-keyboard input, e.g. handwriting, voice recognition, …

```
struct KEYBDINPUT {
    WORD      wVk;
    WORD      wScan;
    DWORD     dwFlags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
};
```

Simulating Mouse Input

- `dx`, `dy` — position (normalized coordinates) or movement
- `mouseData` — amount of wheel movement or which x-button was pressed (`XBUTTON1`, `XBUTTON2`)
- `time`, `dwExtraInfo` — same as for keyboard
- `flags` — combination of flags:
  - `MOUSEEVENTF_MOVE`, `MOUSEEVENTF_WHEEL`, `MOUSEEVENTF_HWHEEL`, `MOUSEEVENTF_LEFTDOWN`, `MOUSEEVENTF_LEFTUP`, …— event type, can be combined (don't combine wheel, x-button flags, down and up flags for the same button)
  - `MOUSEEVENTF_ABSOLUTE` — `dx`, `dy` indicate position instead of relative movement
  - `MOUSEEVENTF_MOVE_NOCOALESCE` — prevent coalescing of mouse move messages
  - `MOUSEEVENTF_VIRTUALDESK` — indicates normalized coordinates for virtual desktop
- Normalized coordinates $(0, 0)$ to top-left and $(65535, 65535)$ to bottom-right corner of primary display or entire virtual desktop (`MOUSEEVENTF_VIRTUALDESK`)
- Relative mouse movement subject to post-processing (mouse speed, acceleration)

```
struct MOUSEINPUT {
    LONG      dx;
    LOGN      dy;
    DWORD     mouseData;
    DWORD     dwFlags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
};
```

Other Input Methods

- Raw Input:
  - Input from all Human Interface Devices (HIDs, includes mouse, keyboard), disabled by default
  - `GetRawInputDeviceList`, `GetRawInputDeviceInfo`
  - `RegisterRawInputDevices` — enables raw input from selected devices, `GetRegisteredRawInputDevices`
  - Unbuffered reading: `WM_INPUT`, `GetRawInputData`, `RAWINPUT`
  - Buffered reading: `GetRawInputBuffer`
  - Very complex, read docs! (or use wrapper library: Direct Input, XInput, etc.)
- Touch Input: `RegisterTouchWindow`, `WM_TOUCH`, `SetGestureConfig`, `WM_GESTURE`
- Ink input

Resources

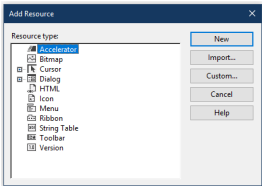
- Binary data often embedded in executable or library file (.exe, .dll, .mui)
- Standard (predefined) resource types:
  - icons, cursors, images (bitmaps, enhanced metafiles)
  - fonts,
  - menu and dialog box templates,
  - string- and message-tables, keyboard accelerator tables,
  - executable/library version information, manifest files
- Applications can define own custom resource types
- Resource identified by executable or library's module handle, type, name and locale id
- Module handle:
  - For current process's main executable use its `HINSTANCE`
  - `LoadLibraryW`, `LoadLibraryExW`
  - `GetModuleHandleW`, `GetModuleHandleExW` — if library already loaded
- Resource type and name:
  - String or integer identifier
  - `MAKEINTRESOURCEW(id)` to convert integer to resource type/name
  - `IS_INTRESOURCE(name)` to check if type/name is integer or string

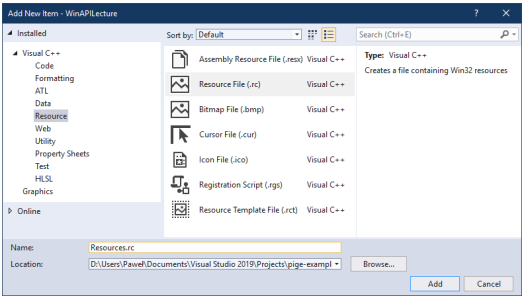
Embedding Resources

- Resource definition (`.rc`) file defines tables, templates, names of external files
- Resource Compiler (`rc.exe`) compiles resources into an object file linked with output binary
- When adding resources to project, Visual Studio adds `.rc` file automatically if needed

  Project > Add > Resource
- Alternatively, empty `.rc` file can be added:

  Project > Add > New Item > Visual C++ > Resource > Resource File (.rc)
- In addition, `resource.h` file is generated. It will contain symbolic constants for identifiers of resource types, names, commands, …





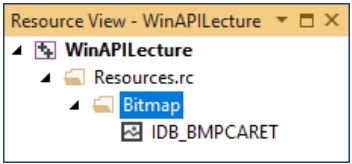
Embedding Resources

Resources.rc Example:

```
// Microsoft Visual C++ generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
...
#undef APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////
// Polish (Poland) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_PLK)
LANGUAGE LANG_POLISH, SUBLANG_DEFAULT

#ifdef APSTUDIO_INVOKED
...
#endif    // APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
// Bitmap
//
IDB_BMPCARET            BITMAP                  "caret.bmp"

#endif    // Polish (Poland) resources
//////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
...
#endif    // not APSTUDIO_INVOKED
```

resource.h Example:

```
// Used by Resources.rc
//
#define IDB_BMPCARET                    101

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
...
#endif
```



Accessing Embedded Resources

- Enumerating resource types in module:
  `EnumResourceTypesW`, `EnumResourceTypesExW`
- Enumerating resource names of a given type in module:
  `EnumResourceNamesW`,`EnumResourceNamesExW`
- Enumerating language variants of resource (given by type and name) in module:
  `EnumResourceLanguagesW`, `EnumResourceLanguagesExW`
- Loading resources (general case, use for application-defined resource types):
  1. `HRSRC FindResourceW(HMODULE module, LPCWSTR rName, LPCWSTR rType)`
  2. *(optional)* `DWORD SizeofResource(HMODULE module, HRSRC res)` — size of resource data
  3. `HGLOBAL LoadResource(HMODULE module, HRSRC res)`
  4. `void* LockResource(HGLOBAL resData)` — pointer to resource data

  No *unlocking/unloading/freeing* of `HRSRC`, `HGLOBAL` or `void*` necessary (or even possible)!
- Always prefer resource type-specific loading functions!

String Tables

- Associate resource names with strings
- LoadStringW
  - Allows to retrieve pointer to string and its length
  - Or copy the string to a buffer
  - Strings might not be null-terminated!

```
wstring clsname(100, L'\0');
//Option 1: Copy string to buffer
auto len = LoadStringW(hInst, IDS_WINDOWCLASS,
    clsname.data(), clsname.length());
clsname.resize(len);
window::register_class(hInst, clsname);

const wchar_t* text;
//Option 2: Obtain pointer to string
//and string length
len = LoadStringW(hInst, IDS_WINDOWTITLE,
    reinterpret_cast<LPWSTR>(&text), 0);
window w{ wstring(text, len) };
```

```
//////////////////////////////////////
//
// String Table
//
STRINGTABLE
BEGIN
    IDS_WINDOWTITLE      "Witaj Świecie!"
    IDS_WINDOWCLASS      "MyWindowClass"
END
```

Resources.rc - String Table

ID	Value	Caption
IDS_WINDOWTITLE	102	Witaj Świecie!
IDS_WINDOWCLASS	103	MyWindowClass

Bitmaps

```
HANDLE LoadImageW(HINSTANCE hInst, LPCWSTR name, UINT type, int cx, int cy, UINT flags)
```

- Loads bitmap, cursor or icon, set `type` to `IMAGE_BITMAP`, `IMAGE_CURSOR` or `IMAGE_ICON`
- `cx`, `cy` — desired cursor/icon size, pass `LR_DEFAULTSIZE` for system default, 0 for actual size
- Some useful `flags`:
  - `LR_LOADFROMFILE` — loads from external file instead of resource
  - `LR_SHARED` — multiple calls for the same resource will return the same handle
- `hInst` — module that contains the resource, pass `nullptr` system resources and external files
- `name` — resource name (use `MAKEINTRESOURCE(ID)` when passing integer ids), or path to a file
  - For system bitmaps, cursors, icons, use identifiers prefixed: `OBM_`, `OIC_`, `OCR_`
  - Define `OEMRESOURCE` before including `"windows.h"` to access them
  - System cursors and icons must be loaded as shared!
- When no longer needed, non-shared images must be released, depending on `type`, using:

`IMAGE_BITMAP`	`DeleteObject`
`IMAGE_CURSOR`	`DestroyCursor`
`IMAGE_ICON`	`DestroyIcon`

Cursors and Icons

Very similar resources, share many characteristics, functions and often can be used interchangeably

- Shape defined by
  - XOR mask — color map of cursor/icon
  - AND mask — 1bpp transparency map, transparent pixels are XOR-ed with background (optional for 32bbp color map)
  - hot-spot — selected pixel corresponding to cursor/icon position (usually center for icons)
- Almost identical file formats for icons (.ico) and non-animated cursors (.cur) — .ico doesn't define hotspot
- Additional file format for animated cursors (.ani)
- File formats can store multiple shapes for different resolutions and/or color depths

Cursor

Creating cursor:

- `LoadImageW` — see previous slide here
- `HCURSOR LoadCursor(HINSTANCE hInst, LPCWSTR resName)`
  - Loads shared cursor from resource with system default size
  - For system cursors pass `nullptr` as `hInst`, and for `name` use one of `IDC_` constants: `IDC_ARROW`, `IDC_IBEAM`, `IDC_WAIT`, …
  - System cursors can be replaced using `SetSystemCursor`
- `LoadCursorFromFileW` — loads a shared cursor from .cur or .ani file with system default size
- `CreateCursor` — creates cursor programmatically (from XOR and AND masks) — avoid, loading should be preferred

Destroying non-shared cursor: `DestroyCursor`

Cursor

Setting cursor appearance:

- `HCURSOR SetCursor(HCURSOR c)` — change cursor shape (`nullptr` hides it)

  Note: window class cursor must be `nullptr`, otherwise by default it's restored on mouse events (see below)
- `hCursor` of `WNDCLASSEXW` on window class registration
- `SetClassLongPtrW` with `GCL_HCURSOR` (affects all window of a class)
- `WM_SETCURSOR` message sent before each mouse message (after hit-testing), use to `SetCursor`
  - `wParam` — handle to window containing mouse cursor
  - `LOWORD(lParam)` — hit-test result
  - `HIWORD(lParam)` — mouse message type identifier
  - `DefWindowProcW` propagates message to parent before processing, sets arrow over non-client area or class cursor over client (if available)
  - Return `TRUE` to halt further processing (e.g. child's window procedure)

Icons

Creating icon:

- `LoadImageW` — see previous slide here
- `HICON LoadIconW(HINSTANCE hInst, LPCWSTR resName)`
  - Loads shared icon from resource with system default size
  - For system icons pass `nullptr` as `hInst`, and for `name` use one of `IDI_` constants: `IDI_INFORMATION`, `IDI_SHIELD`, `IDI_WINLOGO`, …
- `CreateIcon` — creates icon programmatically (from XOR and AND masks) — avoid, loading should be preferred
- `DuplicateIcon` — creates non-shared copy
- `ExtractIconW`, `ExtractIconExW` — extract large icon/large and small icons from .exe, .dll or .ico
- `ExtractAssociatedIconW`, `ExtractAssociatedIconExW` — extract large icon from file or from executable of a program associated with it

Destroying non-shared icons: `DestroyIcon`

Icons

Icon Sizes:

- Large system icon
  - Default for various icons and cursors loading functions
  - `GetSystemMetrics` with `SM_CXICON`, `SM_CYICON`
  - Used in Alt +  , taskbar, etc.
- Small system icon
  - `GetSystemMetrics` with `SM_CXSMICON`, `SM_CYSMICON`
  - Displayed e.g. on window caption bar

Setting icons

- For executable icon, first icon resource is used
- `hIcon`, `hIconSm` of `WNDCLASSEXW` (for large and small icon) on window class registration
- `SetClassLongPtrW` with `GCLP_HICON`, `GCLP_HICONSM` (affects all windows of a class)

Cursor and Icon Functions

- `BOOL GetIconInfo(HICON icon, ICONINFO* info)`
  - Retrieve cursor or icon data
  - Pass any of `IDC_` or `IDI_` constants for system cursors and icons
  - `info.fIcon` — `true` if icon, otherwise cursor
  - `info.hbmMask` — AND mask
  - `info.hbmColor` — XOR mask

```
struct ICONINFO{
    BOOL    fIcon;
    DWORD   xHotspot;
    DWORD   yHotspot;
    HBITMAP hbmMask;
    HBITMAP hbmCOLOR;
};
```

- `HICON CreateIconIndirect(ICONINFO *info)` — creates non-shared copy of cursor/icon (even if cursor is created `DestroyIcon` must be used to release it!)
- `CreateIconFromResource`, `CreateIconFromResourceEx` — Creates icon/cursor from binary resource data (e.g. obtained from `LoadResource`)

Menus Types

- Menu Bar, a.k.a. top-level menu
  - One per window
  - Only for top-level windows
  - Always visible, drawn at the top of window, below caption bar
  - `CreateMenu` — creates empty menu bar
- Pop-up menu, a.k.a. drop-down menu
  - Not visible until activated
  - For submenus, context menus, system menu
  - `CreatePopupMenu` — creates empty pop-up menu
  - Displayed in system-managed pop-up window
- Regardless of type, menus must be destroyed by `DestroyMenu` when no longer needed
- Menu can contain multiple menu items, each optionally associated with a pop-up submenu.

Menu Bars

`HMENU GetMenu(HWND hWnd)` — Retrieves window's menu bar

`BOOL SetMenu(HWND hWnd, HMENU hMenu)`

- Replaces window's menu bar (previous menu bar needs to be retrieved before call)
- Pass `nullptr` to remove it

`BOOL GetMenuBarInfo(HWND hWnd, LONG obj, LONG item, MENUBARINFO* info)`

- If `item` is 0 retrieves menu bar `info` properties:
  - `OBJID_MENU` — window's menu bar
  - `OBJID_SYSMENU` — window's system menu bar (i.e. pseudo-menu bar with one empty item whose submenu is the window's system menu)
- `item` ≥ 0 — Retrieves properties (size/position, focus) of menu bar items (1-based index)

```
struct MENUBARINFO {
    DWORD cbSize; //sizeof(MENUBARINFO)
    RECT  rcBar;  //menu item size/pos
    HMENU hMenu;  //menu bar handle
    BOOL  fBarFocused : 1;
    BOOL  fFocused : 1;
};
```

`BOOL DrawMenuBar(HWND hWnd)` — Redraws window's menu bar (call after changes to menu bar)

Menu Properties

MENUINFO — Used to retrieve or set menu properties

- fMask — which properties set or retrieve:
  - MIM_STYLE, MIM_MAXHEIGHT, MIM_BACKGROUND, MIM_HELPID, MIM_MENUDATA
  - MIM_APPLYTOSUBMENUS — modifies all submenus
- dwStyle — menu style flags, some options:
  - MNS_NOCHECK — doesn't reserve space for item check-marks
  - MNS_CHECKORBMP — item bitmap drawn in the space of a check-mark
  - MNS_MODELESS — set to prevent thread entering modal loop when menu is active (i.e. messages not retrieved via main message loop; manual menu handling might be required)
  - MNS_NOTIFYBYPOS — sends WM_MENUCOMMAND instead of WM_COMMAND
- cyMax — maximum height before scroll bar appears (0 for screen height)
- hbrBack — menu background brush
- dwContextHelpID — context help id
- dwMenuData — application specific value

```
struct MENUINFO {
    DWORD cbSize; //sizeof(MENUINFO)
    DWORD fMask;
    DWORD dwStyle;
    UINT  cyMax;
    HBRUSH hbrBack;
    DWORD dwContextHelpID;
    ULONG_PTR dwMenuData;
};
```

Menu Functions

- `GetMenuInfo`, `SetMenuInfo` — retrieve, set menu properties
- `BOOL GetSystemMenu(HWND hWnd, BOOL revert)`
  - Retrieves a (copy of) system pop-up menu
  - It will be used as system menu for the window and can be modified
  - Pass `true` as `revert` to return to default system menu
- `BOOL TrackPopupMenuEx(HMENU menu, UINT f, int x, int y, HWND hWnd, TPMPARAMS* p)`
  - Displays and tracks selection in context (pop-up) menu — modal loop
  - `x`, `y` — intended menu location in screen coordinates
  - `hWnd` — owner window
  - `p` — rectangle that menu should not overlap
  - Flags `f` define:
    - position of menu in relation to `x`, `y` (alignment, overflow)
    - animation used to display menu
    - which mouse buttons can select items
    - how selection is reported (message to window, returned value)
- `EndMenu` — ends (deactivates, hides) windows active menu

Menu Item Properties

- Text label
  - Use & to select the following letter as access key (mnemonic)
  - Use `\t` to separate label in two columns (usually command label and corresponding shortcut — although such shortcuts don't work automatically)
- Menu item (command) identifier
- Optional sub-menu (such items usually don't acts as commands, as clicking them just expands sub-menu)
- Current state, whether item is disabled, checked, highlighted, default
  (default item is selected when parent item of its submenu is double-clicked)
- Optional bitmap, displayed next to label where check-mark would appear
  (possible to provide two bitmaps for checked and unchecked state)
- Alternatively menu item can be a separator (only valid in pop-up menus)

Menu Item Properties

`MENUITEMINFOW` — Used to retrieve or set menu properties

- `fMask` — which properties to set or retrieve:
  - `MIIM_FTYPE`, `MIIM_STATE`
  - `MIIM_ID` — command identifier via `wID`
  - `MIIM_SUBMENU` — submenu handle via `hSubMenu`
  - `MIIM_BITMAP` — item bitmap via `hbmpItem`
  - `MIIM_CHECKMARKS` — bitmaps for checked, unchecked state via `hbmpChecked`, `hbmpUnchecked` (`nullptr` resets to default)
  - `MIIM_DATA` — application defined value via `dwItemData`
  - `MIIM_STRING` — label string via `dwTypeData` (`cch` for its length)
- `fType` — menu item type flags, e.g.:
  - `MFT_SEPARATOR` — item is a separator (pop-up menus only)
  - `MFT_RADIOCHECK` — defaults to radio-mark instead of check-mark for checked state
  - `MFT_MENUBREAK`, `MFT_MENUBARBREAK` — new row (menu bar), column (pop-up menu, optional bar)
  - `MFT_OWNERDRAW` — allows custom item drawing by window owning the menu
- `fState` — current state flags `MFS_DISABLED`, `MFS_CHECKED`, `MFS_HILITE`, `MFS_DEFAULT`

```
struct MENUITEMINFOW {
    //sizeof(MENUITEMINFO):
    UINT cbSize;
    UINT fMask;
    UINT fType;
    UINT fState;
    UINT wID;
    HMENU      hSubMenu;
    HBITMAP    hbmpChecked;
    HBITMAP    hbmpUnchecked;
    ULONG_PTR dwItemData;
    LPWSTR     dwTypeData;
    UINT       cch;
    HBITMAP    hbmpItem;
};
```

Menu Item Functions

Accessing items, changing properties:

- `GetMenuItemCount` — retrieve number of items in a menu
- `GetMenuItemInfoW`, `SetMenuItemInfoW` — retrieve or set menu properties
- `CheckMenuRadioItem` — simultaneously check one item and uncheck all others in a range

Modifying menus:

- `InsertMenuItemW` — adds item to menu
- `DeleteMenu`, `RemoveMenu` — removes menu item, former destroys submenu, latter does not — it must be retrieved before the call
- `GetMenuDefaultItem`, `SetMenuDefaultItem` — retrieve or change default item in a menu

Measuring items:

- `MenuItemFromPoint`
- `GetMenuItemRect` (must be visible)
- `GetSystemMetrics` with `SM_CXMENUCHECK`, `SM_CYMENUCHECK` — default check-mark size

Note: Items identified by index or command identifier, the latter often recursively searching through submenus

Menu Item Functions

Legacy functions — superseded, although sometimes might be easier to use:

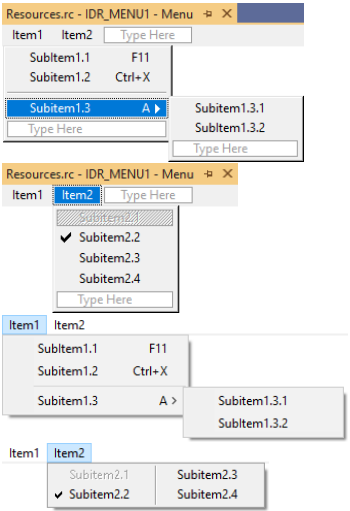
- retrieving properties: `GetMenuItemID`, `GetMenuState`, `GetMenuStringW`, `GetSubMenu`
- changing state: `CheckMenuItem`, `EnableMenuItem`, `HiliteMenuItem`
- adding, modifying items: `AppendMenuW`, `InsertMenuW`, `ModifyMenuW`, `SetMenuItemBitmaps`

Legacy item variant:

- Bitmap instead of text label
- Can be achieved via `SetMenuItemInfoW` with:
  - `fMask`: `MIIM_TYPE`
  - `fType`: `MFT_BITMAP`
  - `dwTypeData`: bitmap handle
- Owner-drawn items should be preferred for custom visual appearance

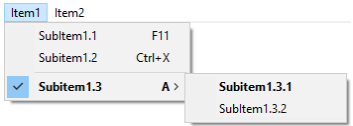
Menu Templates

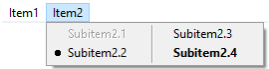
```
//////////////////////////////////////////////////////////////////////////////
//
// Menu
//
IDR_MENU1 MENU
BEGIN
    POPUP "Item1"
    BEGIN
        MENUITEM "SubItem1.1\tF11",      ID_ITEM1_SUBITEM1
        MENUITEM "Subitem1.2\tCtrl+X",   ID_ITEM1_SUBITEM2
        MENUITEM SEPARATOR
        POPUP "Subitem1.3\tA"
        BEGIN
            MENUITEM "Subitem1.3.1",     ID_SUBITEM1_SUBITEM1
            MENUITEM "SubItem1.3.2",     ID_SUBITEM1_SUBITEM2
        END
    END
    POPUP "Item2"
    BEGIN
        MENUITEM "Subitem2.1",  ID_ITEM2_SUBITEM2, GRAYED
        MENUITEM "Subitem2.2",  ID_ITEM2_SUBITEM3, CHECKED
        MENUITEM "Subitem2.3",  ID_ITEM2_SUBITEM4, MENUBARBREAK
        MENUITEM "Subitem2.4",  ID_ITEM2_SUBITEM5
    END
END
```



Menu Templates

```
//                                  Identifier              Type flags          Style Flags
IDR_MENU2 MENUEX
BEGIN
    POPUP "Item1",                  ID_ITEM1
    BEGIN
        MENUITEM "SubItem1.1\tF11",     ID_ITEM1_SUBITEM1
        MENUITEM "Subitem1.2\tCtrl+X",  ID_ITEM1_SUBITEM2
        MENUITEM "",                    ,                   MFT_SEPARATOR
        POPUP "Subitem1.3\tA",          ID_ITEM1_SUBITEM3,  ,               MFS_CHECKED | MFS_DEFAULT
        BEGIN
            MENUITEM "Subitem1.3.1",    ID_SUBITEM1_SUBITEM1,   ,           MFS_DEFAULT
            MENUITEM "SubItem1.3.2",    ID_SUBITEM1_SUBITEM2
        END
    END
    POPUP "Item2",                  ID_ITEM2
    BEGIN
        MENUITEM "Subitem2.1",          ID_ITEM2_SUBITEM2,  ,               MFS_DISABLED
        MENUITEM "Subitem2.2",          ID_ITEM2_SUBITEM3,  MFT_RADIOCHECK, MFS_CHECKED
        MENUITEM "Subitem2.3",          ID_ITEM2_SUBITEM4,  MFT_MENUBARBREAK
        MENUITEM "Subitem2.4",          ID_ITEM2_SUBITEM5,  ,               MFS_DEFAULT
    END
END
```





Menu Templates

```
HMENU LoadMenuW(HINSTANCE hInst, LPCWSTR name)
```

- Loads menu template or extended menu template from resource
- Creates menu bar
- To store context menus as template
  - Define menu bar template with context menu as submenu
  - Load template
  - Use `GetMenuItemInfoW` or `GetSubMenu` to retrieve pop-up menu

Menu Messages

- WM_COMMAND — menu item selected (if menu is not MNS_NOTIFYBYPOS)
  - HIWORD(wParam) — 0 for menus
  - LOWORD(wParam) — selected item's command identifier
- WM_SYSCOMMAND — system menu item selected (as discussed before)
- WM_MENUCOMMAND — menu item selected (if menu is MNS_NOTIFYBYPOS)
  - wParam — item's index
  - lParam — handle of menu that contains the item
- WM_MENURBUTTONUP — menu item right-clicked
  - wParam, lParam as in WM_MENUCOMMAND
  - can be used to display context menu for item (TrackPopupMenuEx with TPM_RECURSE)
- WM_INITMENU — menu bar clicked or Alt pressed (wParam — menu handle)
- WM_INITPOPUPMENU — pop-up menu is becoming active
  - wParam — pop-up menu handle
  - LOWORD(lParam) — index of parent menu item
  - HIWORD(lParam) — TRUE if submenu of window menu bar
- WM_UNINITPOPUPMENU — pop-up menu was destroyed (wParam — pop-up menu handle)

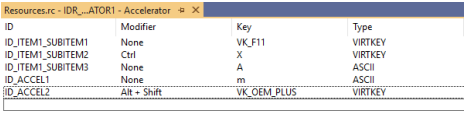
Menu Messages

- WM_CONTEXTMENU
  - Sent by DefWindowProcW on WM_RBUTTONUP, WM_NCRBUTTONUP, ⇧ + F10 or VK_APPS key release
  - lParam — mouse screen coordinates when clicked (use GET_X_LPARAM, GET_Y_LPARAM)
  - wParam — handle of window which received mouse click (or keyboard input)
  - DefWindowProcW propagates message to parent/displays system menu caption bar clicked
  - Can be used to display context menu (TrackPopupMenuEx)
- WM_ENTERMENULOOP, WM_EXITMENULOOP — start/end of menu modal loop
  (wParam TRUE — if context menu)
- WM_ENTERIDLE — periodically sent when modal loop is idle
  - wParam — MSGF_MENU for menu modal loop
  - lParam — handle of window containing the menu
- WM_MENUSELECT — sent when user hovers over/clicks items in a menu
  - LOWORD(wParam) — item index
  - HIWORD(wParam) — item flags (type/state, see docs!)
  - lParam — menu handle

Keyboard Accelerators

- Associate shortcuts (character or virtual key code + modifiers) with command identifiers
- Loading accelerator table from resource:

```
HACCEL LoadAcceleratorsW(HINSTANCE hInst, LPCWSTR name)
```




```
//////////////////////////////////////////////////////////////////////
//
// Accelerator
//
IDR_ACCELERATOR1 ACCELERATORS
BEGIN
    VK_F11,         ID_ITEM1_SUBITEM1,      VIRTKEY, NOINVERT
    "X",            ID_ITEM1_SUBITEM2,      VIRTKEY, CONTROL, NOINVERT
    "A",            ID_ITEM1_SUBITEM3,      ASCII,  NOINVERT
    "m",            ID_ACCEL1,              ASCII,  NOINVERT
    VK_OEM_PLUS,    ID_ACCEL2,              VIRTKEY, SHIFT, ALT, NOINVERT
END
```

- Creating accelerator table dynamically:

```
HACCEL CreateAcceleratorTableW(ACCEL acc[], int count)
```

```
struct ACCEL {
    BYTE fVirt;
    WORD key;
    WORD cmd;
};
```

Keyboard Accelerators

```
int TranslateAcceleratorW(HWND hWnd, HACCEL acc, MSG* msg)
```

- Messages retrieved from queue need to be passed to enable accelerators
- If return value is non-zero, message is handled and should not be processed further
- Commands sent as `WM_COMMAND` or `WM_SYSCOMMAND`
  - `HIWORD(wParam)` — 1 for accelerators (`WM_COMMAND` only)
  - `LOWORD(wParam)` — accelerator command identifier
- Accelerators cooperate with window's menu bar and system menu
  - `WM_SYSCOMMAND` is sent only if accelerator command identifier matches one of system menu's items': `SC_CLOSE`, `SC_MINIMIZE`, …
  - Accelerators matching menu items don't generate messages when item is disabled or window is minimized (latter only applies to menu bar)
  - If accelerator matches menu bar item and window isn't disabled, menu messages are sent: `WM_INITMENU`, `WM_INITPOPUPMENU`, etc. — as if user selected item manually

End of Windows API Lecture 3

Thank you for listening! ☺