

Das erste Windows-Programm

Ziel, Inhalt

- Wir schreiben heute ein erstes Programm, das rein mit der Windows-API ein Fenster erzeugt. Wir lernen dabei einige neue API-Funktionen kennen und sehen wie das Betriebssystem Nachrichten an Applikationen sendet

Das erste Windows-Programm	1
Ziel, Inhalt	1
Ein erstes Windows-Programm	2
Erstellen des Projektes	2
Das Win-32 Projekt	2
Die WinMain-Funktion	2
Windows als Nachrichtenbasiertes System	3
Die Hauptnachrichtenschleife (Message-Loop)	3
Fenster	4
Fensterfunktion	4
Fenster erzeugen	7
Das erste Programm mit eigenem Fenster	7

Ein erstes Windows-Programm

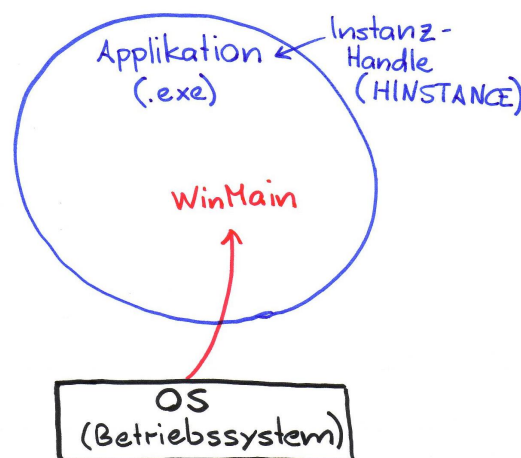
Erstellen des Projektes

Das Win-32 Projekt

Beim erstellen eines neuen Projektes wählen wir im Gegensatz zu den bisherigen Projekten als Projektart die „Win32 Anwendung“ („Win32 Application“). Wir lassen den Wizard ein leeres Projekt erzeugen. Im Gegensatz zu einer Konsolenanwendung wird bei einem solchen Programm nicht eine `main`-Funktion als Eintrittspunkt definiert, sondern wir müssen eine `WinMain`-Funktion bereitstellen.

Die WinMain-Funktion

Durch diese Projektart wird dem Linker mitgeteilt, den Eintrittspunkt der Anwendung, also die Funktion, die vom Betriebssystem aufgerufen wird, wenn das Programm geladen wird auf die Funktion `WinMain` zu setzen.



Die Funktion ist eine globale Funktion, die folgende Argumente vom Betriebssystem erhält:

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
```

- `HINSTANCE hInstance`, jede laufende Applikation wird durch ein Instanz-Handle identifiziert.
- `HINSTANCE notUsed`, das zweite Argument wird nicht mehr benutzt (Win3.1)
- `LPSTR lpCmdLine`, enthält die Kommandozeile, mit der das Programm gestartet wurde.

- `int nCmdShow`, gibt an wie das Hauptfenster unser Applikation aussehen soll (minimiert, maximiert, etc.)

Wir werden in unserem Programm nur das Instanz-Handle unserer Applikation verwenden, denn es wird gebraucht um Fenster zu erzeugen.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE,
                  LPSTR,
                  int)
{
    return 0;
}
```

Das sollte eine Kompilierfähige Applikation ergeben, die selbstverständlich noch nichts macht.

Windows als Nachrichtenbasiertes System

Windows verarbeitet die meisten Ereignisse, die der Benutzer erzeugt als so genannte Windows-Messages. Dabei weiss das Betriebssystem welche Applikation gerade aktiv ist, also den Fokus hat. Drückt nun der Benutzer irgendeine Taste, wird dieser Tastendruck mittels einer Message an die aktive Applikation gesendet.

Ähnliches geschieht mit Mausereignissen. Das Betriebssystem (oder OS für Operating-System) kennt die Position der verschiedenen Fenster der gerade laufenden Applikation. Wird die Maus nun über ein Fenster bewegt oder wird in ein Fenster geklickt, sendet das Betriebssystem ein Ereignis an die Applikation zu der das Fenster gehört.

Die Hauptnachrichtenschleife (Message-Loop)

Eine Windows-Applikation muss sich also die Nachrichten abholen. Im Normalfall geschieht dies in einer Schleife. Um Nachrichten abzuholen gibt es verschiedene API's.

`PeekMessage`, `GetMessage`

Wir verwenden hier *GetMessage*. Diese Funktion hat als Argument einen Zeiger auf eine *Msg* Struktur. Diese Struktur enthält Felder für das Fenster-Handle, für welches die Nachricht bestimmt ist. Ein Feld steht für die Nachricht mit zwei weiteren Feldern, die je nach Nachricht benutzt werden. Die Funktion *GetMessage* gibt als Rückgabewert an, ob die Schleife weiter drehen soll. Falls mit *GetMessage* die Nachricht mit dem Code `WM_QUIT` (in `winuser.h` definiert, oder `windows.h`) geholt wird, gibt *GetMessage* 0 zurück und die Schleife wird verlassen.

```
#include <windows.h>

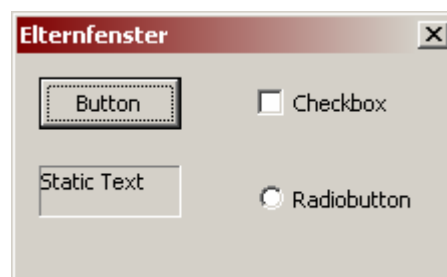
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE,
                  LPSTR,
                  int)
{
    // MessageLoop
    MSG msg;
    while(GetMessage(&msg, // Zeiger auf Message Struktur
                  0, // hier könnte man ein Fenster-Handle angeben
                  0, // den tiefsten Code für empfangene Nachricht
                  0)) // den höchsten Nachrichtencode oder 0 für alle
    {
        // Nachrichten verarbeiten
    }
    return 0;
}
```

Dieses einfache Windows-Programm wird aber nie verlassen, da es keine Fenster hat und dadurch keine Nachrichten erhält.

Fenster

Fensterfunktion

Man unterscheidet verschiedene Fensterarten, aufgrund ihrer Darstellung und ihrer Art auf Ereignisse (Tasten oder Maus) zu reagieren. Hier eine kleine Auswahl vordefinierter Fensterarten:



Solche vordefinierten Fensterarten sind häufig Elemente, die man häufig braucht und immer möglichst gleich aussehen sollen.

Damit eine Fensterart sich immer gleich verhält und aussieht, kümmert sich immer die gleiche Funktion um die Behandlung der Fensternachrichten für eine Fensterart. Diese Fensterarten werden als „Window-Classes“ bezeichnet, aber nicht im Sinn von C++ - Klassen! Um eine solche Fensterfunktion mit einer Fensterart zu verbinden, müssen wir eine „Window-Class“ registrieren. Dabei geben wir auch an, wie die Hintergrundfarbe des Fensters sein soll, oder was für ein Cursor für das Fenster verwendet wird. Um eine Fensterklasse zu registrieren müssen wir eine WNDCLASS Struktur füllen.

```
WNDCLASS wc = {0}; // Struktur
wc.hInstance = hInstance; // Handle der
// Applikation
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1); // Hintergrundfarbe
// für die
// Fensterart
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Maus-Cursor
// gültig für diese
// Fensterart
wc.lpszClassName = "MyFirstWindowCls"; // Name für diese
// Window-Class
wc.lpfnWndProc = WindowProcedure; // Window-
// Bearbeitungsfunktion
```

Das registrieren selbst geschieht mit der Funktion *RegisterClass*.

```
if(RegisterClass(&wc)); // Die Window-Class registrieren
{
    // Registrierung ok
}
```

Die Fensterfunktion selber (hier WindowProcedure genannt) muss so aussehen:

```
LRESULT CALLBACK WindowProcedure(HWND hWnd,
                                  UINT message,
                                  WPARAM wParam,
                                  LPARAM lParam)
{
    switch(message)
    {
        case WM_DESTROY:
            // Das Fenster wird geschlossen
            PostQuitMessage(0); // Die WM_QUIT Nachricht wird
                                //erzeugt (siehe oben bei GetMessage)
            break;
        default:
            // Diese Funktion verarbeitet die Nachricht auf
            // Standard-Art
            return DefWindowProc(hWnd,
                                  message,
                                  wParam,
                                  lParam);
    }

    return 0;
}
```

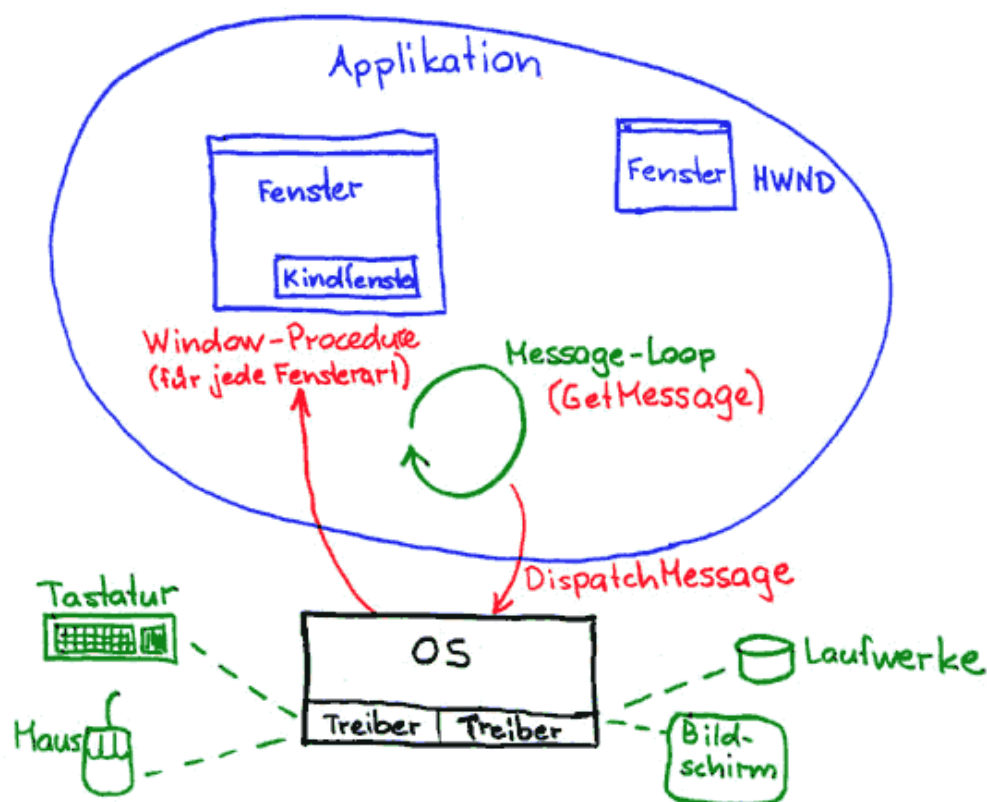
Als Rückgabewert hat sie einen LRESULT also einen *long*. Als Argument erhält sie das Handle des Fensters, für welches die Nachricht ist. Das zweite Argument ist der Message Code von denen sehr viele in windef.h vordefiniert sind. Je nach Nachricht können die beiden Argumente wParam und lParam unterschiedliche Bedeutungen haben.

In unserem Beispiel bearbeiten wir nur die Nachricht WM_DESTROY. Diese Nachricht erhalten wir wenn ein Fenster zerstört werden soll. Da wir im

ersten Anlauf nur ein Fenster erzeugen, nehmen wir das als Anlass *PostQuitMessage* aufzurufen. Diese Funktion erzeugt im Grunde genommen eine *WM_QUIT* Nachricht, die unsere Nachrichtenschleife zum Abbruch veranlasst. Alle anderen Nachrichten geben wir einfach an eine bereits vorhandene Funktion *DefWindowProc* weiter, die für viele Nachrichten die richtige Aktion ausführt.

Damit diese Fensterprozedur überhaupt aufgerufen wird, müssen wir dem Betriebssystem den Auftrag geben die Nachricht, die wir mit *GetMessage* abholen zu verteilen. Die Funktion hierfür heisst *DispatchMessage*. Die Nachrichtenschleife sieht also so aus:

```
// Main message loop:
MSG msg = { 0 };
// GetMessage wartet auf eine Nachricht
// Falls die Nachricht die WM_QUIT-Nachricht ist
// gibt GetMessage 0 zurück und die Schleife wird
// verlassen.
while (GetMessage(&msg, NULL, 0, 0))
{
    // Mit DispatchMessage werden die Nachrichten
    // (Messages) auf die Fensterfunktionen verteilt
    DispatchMessage(&msg);
}
```



Fenster erzeugen

Jetzt haben wir eine Nachrichtenschleife und eine Window-Class mit einer Fensterprozedur. Fehlt nur noch das Fenster selber.

Das Fenster erzeugen wir, nachdem wir die Fensterklasse registriert haben. Die Funktion, die uns von Betriebssystem hierfür zur Verfügung steht heisst *CreateWindow*. Diese Funktion nimmt viele Argumente entgegen und gibt als Identifikation für ein Fenster ein HWND zurück, also ein Fenster-Handle.

```
HWND hwnd = CreateWindow("MyFirstWindowCls", // Window-Class Name
                          "Markus",          // Fenstername
                          WS_VISIBLE | WS_OVERLAPPEDWINDOW, // Fenster-Stil
                          10,                 // x
                          10,                 // y
                          400,                // breite
                          300,                // hoehe
                          NULL,               // Eltern-Fenster (NULL > Toplevel)
                          NULL,               // Handle für ein Menu (NULL keins)
                          hInstance,          // Applikations-Handle
                          NULL);              // Zeiger auf Daten, die der Fenster-
                                              // prozedur übergeben werden
```

Das erste Argument ist der Name der Fensterklasse die vorher registriert werden musste. Das zweite ist der Name des Fensters. Dieser Name wird bei vielen Fenstern zur Anzeige verwendet. Mit dem dritten Argument können wir Window-Stile (WS_XXX) mit bitweisem „Oder“, verknüpfen. Hier machen wir das Fenster sichtbar (WS_VISIBLE) und statten das Fenster mit einigen grundsätzlichen Dingen wie eine Fensterleiste und einen Schliessen-Knopf aus. Die nächsten vier Argumente sind für die Grösse und Position des Fensters. Die anderen Argumente sind kommentiert, oder in der Hilfe viel genauer beschrieben.

Das erste Programm mit eigenem Fenster

Jetzt sollten wir in der Lage sein das Programm, das im Moment nur aus einer .cpp Datei besteht fertig zu stellen. Ein Beispiel kannst du auch von meiner Site [herunterladen](http://www.devmentor.ch/teaching/additional/001/Semester5/Abend12/Code.cpp).

<http://www.devmentor.ch/teaching/additional/001/Semester5/Abend12/Code.cpp>