

## Kurseinheit 10: Modulare Programmierung

1. Modulare Programmierung
2. Mehrdateienprogramme
3. Parameterübergabe an Main
4. Dynamic Link Library
5. Grundstruktur Solution

## Übersicht KE 10

Lehrveranstaltung Ingenieur-Informatik – 2 SWS/2 Credits: EI1, EI+1, MKA1, MK+1, EI3nat3  
Lehrveranstaltung Programmierung 2 (Teil C) – 2 SWS/2 Credits: AI2

Unterrichtsdauer für diese Kurseinheit: 90 Minuten

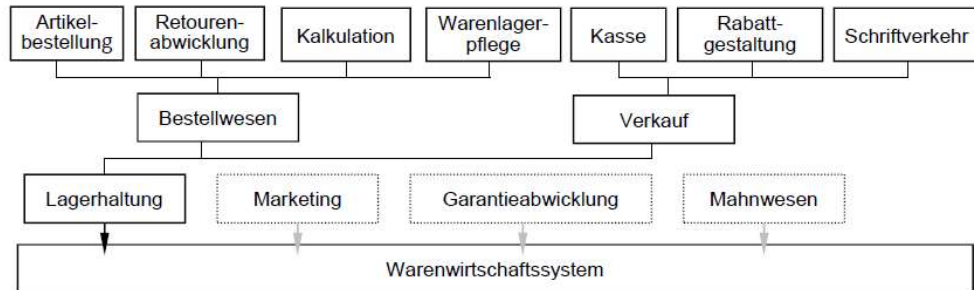
Korrespondierende Kapitel aus *C-Programmierung – Eine Einführung*: Kapitel 11

Zusatzthemen: Dynamic Link Libraries

## 1. Modulare Programmierung

3

### Beispiel Warenwirtschaftssystem



[LUH17]

Prinzip der modularen Programmierung: Aufgabe in Teilprobleme zu zerlegen. Diese lassen sich dann einfacher in einem „Modul“ realisieren. Wichtig sind dabei die Schnittstellen zwischen den Modulen. Diese Schnittstellen müssen einheitlich und gut dokumentiert sein.

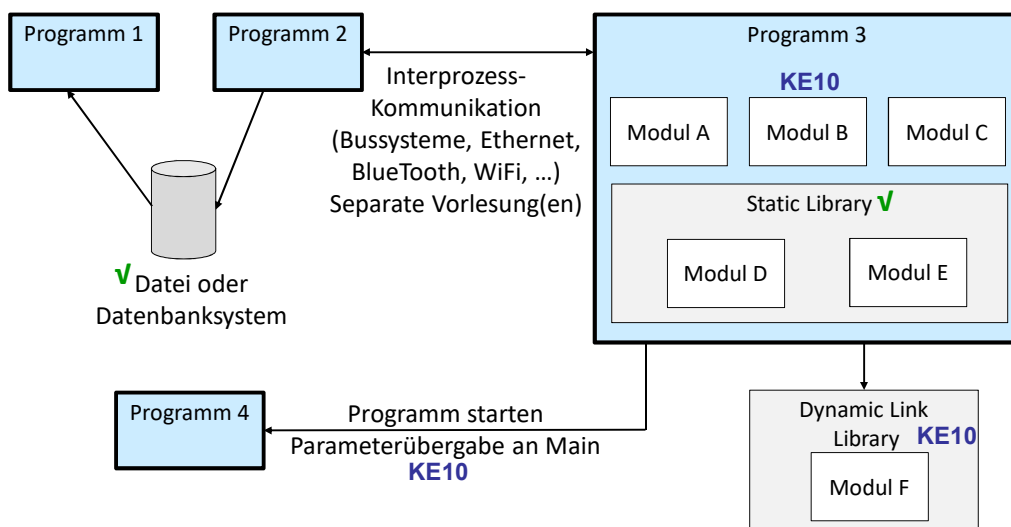
Vorteile der modularen Programmierung siehe [LUH17], S. 150

## 1. Modulare Programmierung

4

### Wie lässt sich modulare Programmierung realisieren?

Programm in Ausführung: Prozess – Prozesse können auf unterschiedlichen Rechnern laufen.



## 1. Modulare Programmierung

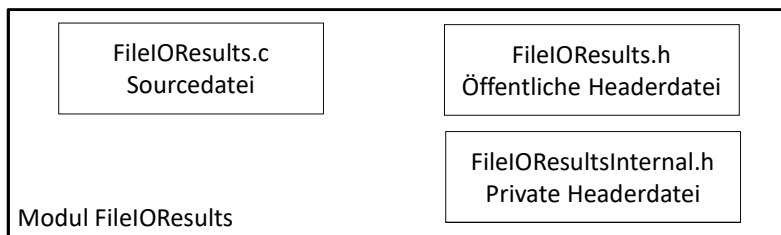
5

### Was ist ein Modul?

Ein **Modul** (neutrum, das Modul) ist im Software Engineering ein Baustein eines Software-systems, der bei der Modularisierung entsteht, eine funktional geschlossene Einheit darstellt und einen bestimmten Dienst bereitstellt.

Quelle: wikipedia.org

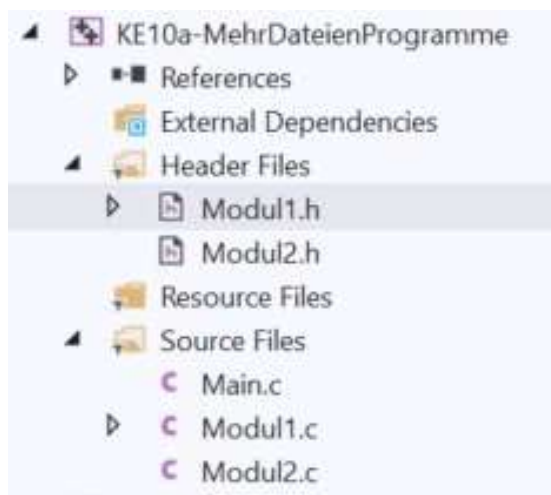
In der Programmiersprache C wird unter einem Modul eine \*.c-Datei mit entsprechender (öffentlicher) \*.h-Datei verstanden. In einer \*.c-Datei sollten sich ein bis mehrere C-Funktionen befinden, die logisch zusammengehören (z.B. Read/Write von Dateien). Im (öffentlichen) Header sollten sich die Deklarationen, #defines und typedefs befinden, die benötigt werden, wenn das Modul verwendet wird. Entgegen [LUH17]: Keine Definitionen in Headerdateien.



## 2. Mehrdateienprogramme

6

### Beispiel: Einfaches Mehrdateienprogramm



Programm mit drei Modulen:

- Main
- Modul1
- Modul2

Fragestellungen:

1. Welches Modul kann welche Funktionen aufrufen?
2. Welche Variablen können gemeinsam genutzt werden?
3. Welche Möglichkeiten bietet C bezüglich „Kapselungsprinzip“, d.h. das Verbergen von Funktionen und Variablen für andere Module?

## 2. Mehrdateienprogramme

7

### Beispiel: Einfaches Mehrdateienprogramm – Modul 1

```
#include "Modul1.h"

static int f3(int iVal);

int f1(int iVal)
{
    return (3 * iVal);
}

int f2(int iVal)
{
    return (f3(iVal) + f1(iVal));
}

static int f3(int iVal)
{
    return (iVal - 1);
}
```

**Modul1.c**

```
#pragma once

extern int f1(int iVal);
extern int f2(int iVal);
```

**Modul1.h**

Funktionen, die mit **static** deklariert sind, können nur innerhalb des Moduls aufgerufen werden (Kapselungsprinzip). Funktionen, die mit **extern** deklariert sind, können auch von außerhalb aufgerufen werden (extern ist dabei default).

**Modul1**

## 2. Mehrdateienprogramme

8

### Beispiel: Einfaches Mehrdateienprogramm – Modul 2

```
#include "Modul2.h"

static int iModulGlobal = 5;
int iGlobal2 = 3;

int f4(int iVal)
{
    int iRet;
    iRet = (iGlobal1 * iVal) +
           (iGlobal2 * iVal) +
           iModulGlobal;
    return iRet;
}
```

**Modul2.c**

```
#pragma once

extern int iGlobal1;

int f4(int iVal);
```

**Modul2.h**

Variablen, die außerhalb einer Funktion mit **static** definiert wurden, sind modulglobal (Zugriff nur von der \*.c-Datei aus - Kapselungsprinzip). Variablen, die außerhalb einer Funktion ohne static deklariert sind, erlauben auch einen Zugriff von außerhalb, diese sind dann global. Wird von außerhalb darauf zugegriffen, so sind die globalen Variablen dort als **extern** zu deklarieren.

**Modul2**

iGlobal2 ist im Modul2 definiert. iGlobal1 muss in einem anderen Modul definiert sein.

## 2. Mehrdateienprogramme

9

### Beispiel: Einfaches Mehrdateienprogramm – Main

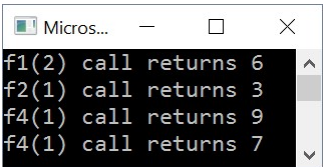
```
#include <stdio.h>
#include "Modul1.h"
#include "Modul2.h"

extern int iGlobal2;
int iGlobal1 = 1;

int main(void)
{
    printf("f1(2) call returns %d\n", f1(2));
    printf("f2(1) call returns %d\n", f2(1));
    printf("f4(1) call returns %d\n", f4(1));
    iGlobal2 = 1;
    printf("f4(1) call returns %d\n", f4(1));

    return 0;
}
```

Main.c



```
f1(2) call returns 6
f2(1) call returns 3
f4(1) call returns 9
f4(1) call returns 7
```

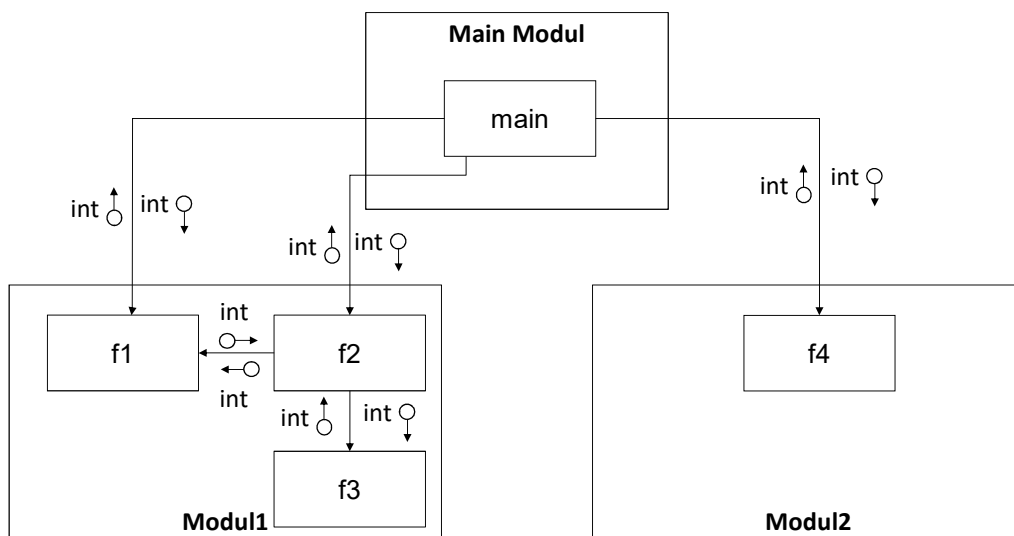
Der Aufruf von f3 würde zu einem Compilerfehler führen, da die Funktion in Modul1 static ist.

Main

## 2. Mehrdateienprogramme

10

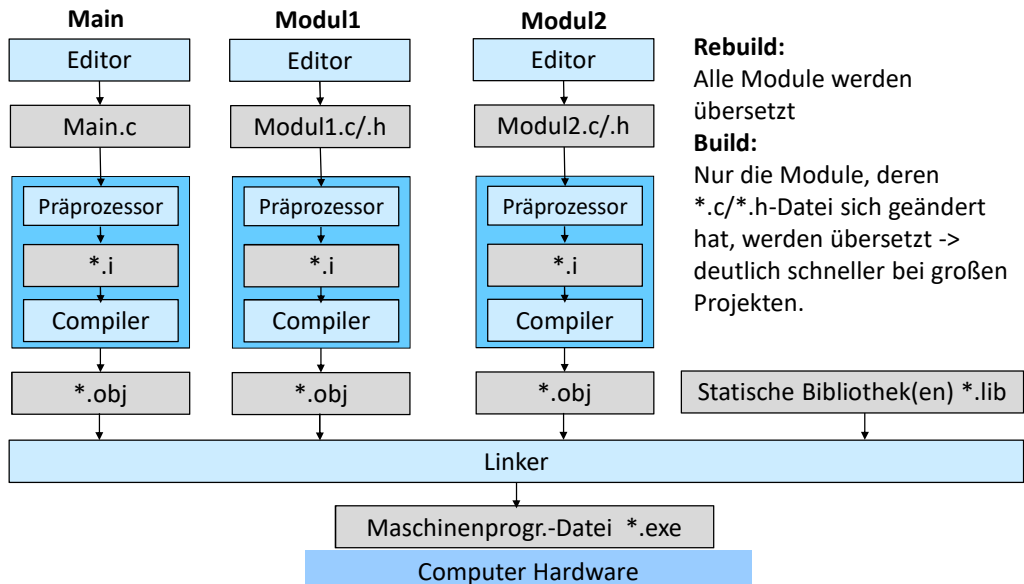
### Beispiel: Einfaches Mehrdateienprogramm – Strukturdiagramm



## 2. Mehrdateienprogramme

11

### Beispiel: Einfaches Mehrdateienprogramm – Generierung Maschinenprogr.



## 2. Mehrdateienprogramme

12

### Sichtbarkeit, Speicher und Initialisierung von Variablen

Beispiel	Sichtbarkeit	Speicher	Default-Initialisierung
<code>int iGlobal;</code> //außerhalb Funktion	Im ganzen Programm! extern nutzen	Daten-segment	0
<code>static int iModulGlobal;</code> //außerhalb einer Funktion	Nur im Modul	Daten-segment	0
<code>static int iVal1;</code> //innerhalb einer Funktion	Nur innerhalb der Funktion	Daten-segment	0
<code>int f(int iVal2)</code> //Übergabeparameter	Nur innerhalb der Funktion	Stack-segment	Durch Aufrufer
<code>auto int iVal3;</code> //Lokale Variable	Nur innerhalb der Funktion	Stack-segment	zufällig
<code>{</code> <code>int iVal4; //Blockvariable</code> <code>}</code>	Nur innerhalb des Blocks – C99 Feature	Stack-segment	zufällig

Die Sichtbarkeit wird durch den Compiler überwacht.

## 2. Mehrdateienprogramme

13

### Sichtbarkeit, Speicher und Initialisierung von Konstanten

Wiederholung: Es gibt drei Arten von Konstanten

- Literale Konstanten (3.1415)
- Konstante Variablen (const float cfPI = 3.1415f)
- Symbolische Konstanten (#define PI 3.1415 oder #define PI cfPI)

Der Präprozessor ersetzt die symbolische Konstante durch literale Konstanten oder konstante Variablen.

Literale Konstanten werden im Codesegment abgelegt oder direkt in Maschinencode umgesetzt.

Für konstante Variablen gelten die gleichen Sichtbarkeiten wie für Variablen. Diesen muss immer ein Wert bei der Definition zugewiesen werden (sonst Compilerfehler). Ebenso überwacht der Compiler, dass der Wert nicht mehr geändert werden kann. Auch die konstanten Variablen liegen im Codesegment.

## 3. Parameterübergabe an main

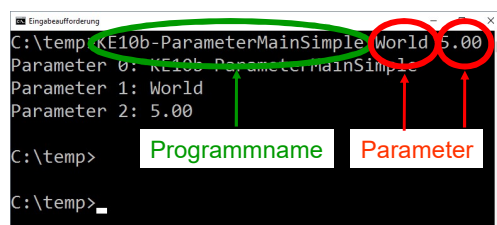
14

### Prinzip

Ein Programm kann vom Anwender oder von einem anderen Programm gestartet werden. Bisher wurden die Programme über die IDE gestartet.

Diese Möglichkeit wird benutzt, wenn ein Programm kein Benutzerinterface hat oder wenn ein Programm sowohl die Steuerung mit Benutzerinterface und über Parameterübergabe anbieten soll.

#### Anwender startet Programm



```
C:\temp> KE10b-ParameterMainSimple World 5.00
Parameter 0: KE10b-ParameterMainSimple
Parameter 1: World
Parameter 2: 5.00
C:\temp>
```

Starten der Eingabeaufforderung. Ggf. Navigation ins Verzeichnis der Anwendung. Anwendung starten mit Programmnamen (.exe nicht notwendig) und Parametern getrennt durch Leerzeichen.

#### Programm startet Programm

Eine Windowsanwendung kann über die Windows-API-Funktion **CreateProcess** andere Programme starten.

In dieser LV wird hierzu die IDE verwendet, die dann intern die Programme startet. Um Parameter beim Aufruf zu übergeben: Project Properties -> Configuration Properties -> Debugging -> Command Arguments



```
Command Arguments: World 5.00
```

### 3. Parameterübergabe an main

15

#### Notwendige Anpassung von main

Bisher mussten alle Funktion außer main deklariert werden. Die Funktion main dient als Einsprung und muss diesen Namen haben. Intern im Compiler ist die Funktion main in **zwei Varianten** schon deklariert! (Intrinsic Function)

##### Variante 1

```
int main(void);
```

Diese Variante wurde bisher benutzt.

Oft finden sich in der Praxis C-Programme, die Variante 2 verwenden, ohne die Parameter zu nutzen. Dies liegt daran, dass die IDE beim Anlegen eines Projektes immer im voraussetzenden Gehorsam Variante 2 anlegt. In diesem Kurs wurde immer „Empty Project“ als Vorlage ausgewählt. Dort unterbleibt dann die automatische Generierung von Variante 2.

char\*\* argv

##### Variante 2

```
int main(int argc, char* argv[]);
```

Diese Variante ist notwendig, wenn Parameterübergabe an main erfolgen soll. argc (**argument count**) enthält die Anzahl der Parameter und die argv (**argument values**) die Argumente (char\* Zeiger). argv könnte alternativ auch als char\*\* argv deklariert werden (Zeiger auf Zeiger)

argv[argc-1]	char*	→ Parameter 3
...	char*	→ Parameter 2
argv[1]	char*	→ Parameter 1
argv[0]	char*	→ Parameter 0

### 3. Parameterübergabe an main

16

#### Einfaches Beispiel

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int iCounter;

    for (iCounter = 0; iCounter < argc; iCounter++)
    {
        printf("Parameter %d: %s\n", iCounter, argv[iCounter]);
    }

    return 0;
}
```

argv[0], argv[1], ... sind char-Zeiger (char\*)  
argv[0] ist die Zeichenkette (char-Zeiger) des Programmaufruf, ggf. mit Pfadangabe  
**argv[argc] ist undefined!**

```
Eingabeaufforderung
C:\temp>KE10b-ParameterMainSimple World 5.00
Parameter 0: KE10b-ParameterMainSimple
Parameter 1: World
Parameter 2: 5.00
C:\temp>
C:\temp>
```



### 3. Parameterübergabe an main

17

#### Komplexeres Beispiel (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NAMES_LENGTH 40

int main(int argc, char* argv[])
{
    char acLastName[NAMES_LENGTH];
    char acFirstName[NAMES_LENGTH];
    int iMatrNo;
    double dMarksAverage;

    // Sanity checks and error message
    // See next slide

    return 0;
}
```

Dieses Beispiel erwartet wahrscheinlich 1 + 4 Übergabeparameter (Programmname (immer vorhanden) sowie Nachname, Vorname, Matrikelnummer und Notendurchschnitt.

Die fünf Parameter sind somit in argv[0] bis argv[4] zu erwarten.

Es handelt sich dabei um char-Zeiger (char\*). Für acLastName und acFirstName kann ein einfaches Kopieren verwendet werden (Äquivalenz zwischen char[] und char\*). Für iMatrNo und dMarksAverage sind Konvertierungsfunktionen notwendig, die von char\* in den Zieltyp int oder double konvertieren. Details zu Konvertierungsfunktionen werden in KE11 behandelt.

### 3. Parameterübergabe an main

18

#### Komplexeres Beispiel (2)

Zuerst ist zu überprüfen, ob überhaupt 5 Parameter übergeben wurden.

- Sind mehr übergeben worden, so werden die letzten nicht ausgewertet -> Nicht korrekt!
- Sind weniger übergeben worden, so würde bei 4 Parametern der Zugriff auf argv[4] eine Exception auslösen oder undefiniertes Verhalten zeigen (usw.)

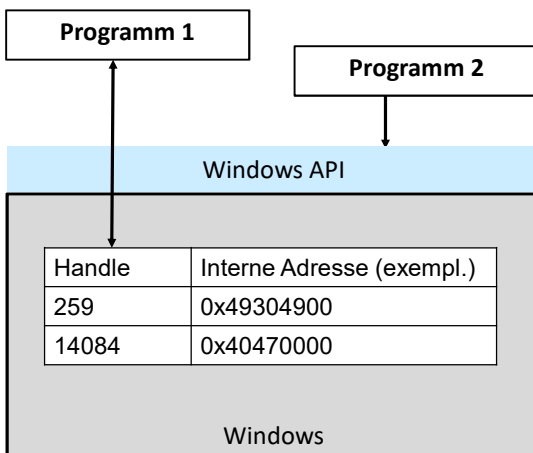
```
// Sanity checks and error message
if (argc == 5)
{
    //if correct Number of passed parameter assign or convert them
    //argv[0] Program name + optional path not used
    strncpy_s(acFirstName, NAMES_LENGTH, argv[1], NAMES_LENGTH-1);
    strncpy_s(acLastName, NAMES_LENGTH, argv[2], NAMES_LENGTH-1);
    iMatrNo = atoi(argv[3]);
    dMarksAverage = atof(argv[4]);
    printf("%s %s %i %f\n", acLastName, acFirstName, iMatrNo, dMarksAverage);
}
else
{
    printf("Number of passed parameters is wrong!\n");
}
```

## 4. Dynamic Link Library (DLL)

19

### Grundlagenwissen: Windows API und Handles

Bei der Erstellung von DLLs (und auch später bei der Programmierung von Threads) ist etwas Grundlagenwissen über die **Windows API** und **Handles** notwendig. Die Windows API (Application Programming Interface) ist eine Schnittstelle, um auf Systemressourcen (Fenster, Dateien, ...) des Betriebssystems zuzugreifen. Dazu ist nur <windows.h> zu inkludieren.



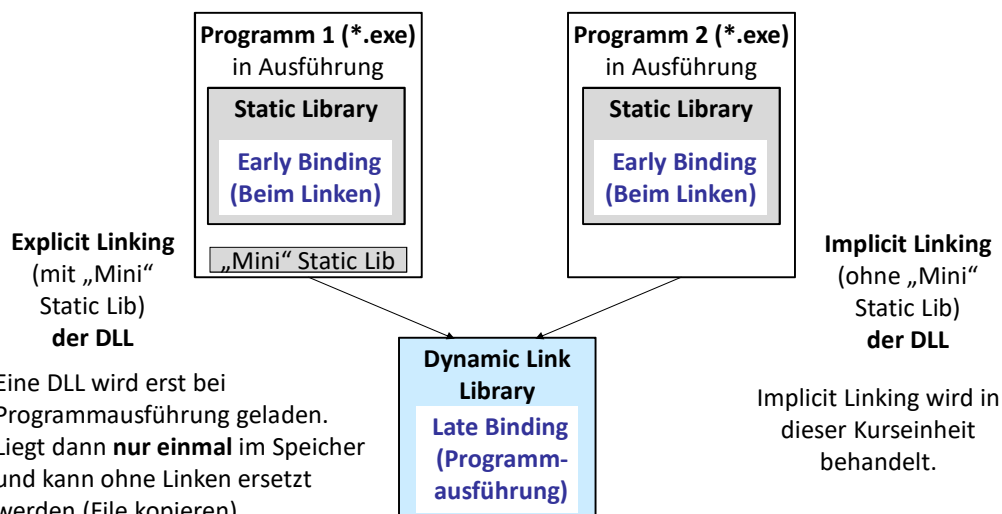
Der Zugriff auf die Systemressourcen mit Zeigern wäre zu fehleranfällig, daher arbeitet ein Betriebssystem mit Handles, was intern nur eindeutige Nummern (Zahlen) in Tabellen sind. Dahinter verbirgt sich dann die konkrete Adresse der Systemressource. Das Betriebssystem kann anhand der Handlenummer überprüfen, ob dies eine korrekte Nummer ist. Der typedef von Handels beginnt in Windows immer mit einem „H“, z.B. HMODULE. Nur das Betriebssystem kann auf die Tabellen zugreifen.

## 4. Dynamic Link Library (DLL)

20

### Prinzip

Zwei Programme nutzen die gleiche Static Lib, diese ist zweimal im Speicher vorhanden und ist Bestandteil der \*.exe. Ändert sich die Static Lib ist erneut zu Linken.



## 4. Dynamic Link Library (DLL)

21

### DLL-Erstellung

Eine DLL kann in MSVS auf unterschiedliche Weise angelegt werden. Hier eine Möglichkeit:

1. Leeres Projekt anlegen (Empty Project)
2. Properties -> Configuration Properties -> General -> Configuration Type -> Dynamic Library (.dll) anwählen und mit OK bestätigen.
3. Jetzt eine Header und eine C-Datei dem Projekt hinzufügen.

```
#pragma once
__declspec(dllexport) void PrintFromDLL(char* pcText);
```

KE10d-DLL.h

```
#include <stdio.h>
#include "KE10d-DLL.h"
__declspec(dllexport) void PrintFromDLL(char* pcText)
{
    //Sanity check
    if (pcText != NULL)
    {
        printf("%s\n", pcText);
    }
}
```

DLLMain.c

Nur Funktionen die mit  
\_\_declspec(dllexport)  
gekennzeichnet sind, können von  
Anwendungen aufgerufen werden.

4. Rebuild/Build: -> Allerdings kann eine DLL nicht direkt ausgeführt werden.

## 4. Dynamic Link Library (DLL)

22

### Anwendung, die eine DLL nutzt – Implicit Linking

Eine DLL kann nur mit einer Anwendung getestet werden. Beim Implicit Linking muss die Anwendung die folgenden Details der DLL kennen:

1. Verzeichnis der DLL
2. Name der DLL
3. Deklaration der exportierten Funktion(en), die genutzt werden. Eine DLL kann selbstverständlich auch interne Funktionen haben, die von außen nicht aufgerufen werden können (Kapselungsprinzip).

Per Default ist der Searchmode der DLL auf SafeDllSearchMode gestellt, d.h. die DLL wird in den folgenden Verzeichnissen nach dieser Reihenfolge gesucht.

1. Verzeichnis wo sich die Exe befindet
  2. Systemverzeichnis
  3. 16-Bit Systemverzeichnis
  4. Windowsverzeichnis
  5. Aktuelles Verzeichnis
  6. Verzeichnisse, die in der Umgebungsvariablen aufgeführt sind.
- Beindet sich das DLL- und das Anwendungsprojekt in einer Solution, werden \*.dll- und \*.exe-Dateien in das gleiche Verzeichnis kopiert.

## 4. Dynamic Link Library (DLL)

23

### Anwendung, die eine DLL nutzt – Implicit Linking

```
#include <windows.h>
```

```
int main(void)
```

```
{
```

```
    HMODULE hModule;
```

```
    void (*fpDLLFunc)(char*);
```

```
    hModule = LoadLibrary(TEXT("KE10d-DLL.dll"));
```

```
    if (hModule != 0)
```

```
    {
```

```
        fpDLLFunc = (void (*)(char*))GetProcAddress(hModule, "PrintFromDLL");
```

```
        if (fpDLLFunc != NULL)
```

```
        {
```

```
            fpDLLFunc("My First Call to a DLL");
```

```
        }
```

```
        FreeLibrary(hModule);
```

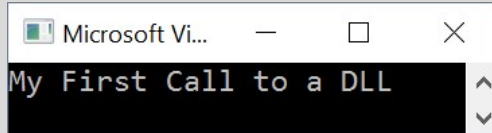
```
    }
```

```
    return 0;
```

```
}
```

Es müssen Windows-API Funktionen genutzt werden:

- LoadLibrary und FreeLibrary
- GetProcAddress



## 4. Dynamic Link Library (DLL)

24

### Load Library und FreeLibrary – Implicit Linking

```
#include <windows.h>
```

```
int main(void)
```

```
{
```

```
    HMODULE hModule;
```

```
    void (*fpDLLFunc)(char*);
```

```
    hModule = LoadLibrary(TEXT("KE10d-DLL.dll"));
```

```
    if (hModule != 0) ← LoadLibrary erfolgreich
```

```
    {
```

```
        fpDLLFunc = (void (*)(char*))GetProcAddress(hModule, "PrintFromDLL");
```

```
        if (fpDLLFunc != NULL)
```

```
        {
```

```
            fpDLLFunc("My First Call to a DLL");
```

```
        }
```

```
        FreeLibrary(hModule);
```

```
    }
```

```
    return 0;
```

```
}
```

Cast: #define aus windows.h

Searchmode muss berücksichtigt werden.

Fall 1: DLL und Exe im gleichen Verzeichnis

Wird DLL nicht mehr benötigt, kann diese freigegeben werden. Wird dann aus dem Speicher entfernt, falls keine andere Anwendung diese DLL gleichzeitig nutzt.

## 4. Dynamic Link Library (DLL)

25

### GetProcAddress

```
#include <windows.h>
```

```
int main(void)
```

```
{
```

```
    HMODULE hModule;
```

```
    void (*fpDLLFunc)(char*);
```

Es muss ein Funktionszeiger deklariert werden, der kompatibel mit der aufzurufenden Funktion ist (Dokumentation oder Headerdatei liefert die Information)

```
    hModule = LoadLibrary(TEXT("KE10d-DLL.dll"));
```

```
    if (hModule != 0)
```

```
    {
```

```
        fpDLLFunc = (void (*)(char*))GetProcAddress(hModule, "PrintFromDLL");
```

Cast auf obigen Funktionszeiger

```
        if (fpDLLFunc != NULL)
```

GetProcAddress erfolgreich

```
        {
```

```
            fpDLLFunc("My First Call to a DLL");
```

Aufruf über Funktionszeiger

```
        }
```

```
        FreeLibrary(hModule);
```

```
    }
```

```
    return 0;
```

```
}
```

## Zusammenfassung KE 10

26

### Behandelte Schlüsselwörter in KE 10

Schlüsselwörter C89:

~~auto~~ ✓

do ✓

~~goto~~ ✓

signed ✓

unsigned ✓

break ✓

double ✓

if ✓

sizeof ✓

void ✓✓

case ✓

else ✓

int ✓

static ✓✓

volatile ✓

char ✓

enum ✓

long ✓

struct ✓

while ✓

const ✓

extern ✓✓

register ✓

switch ✓

~~continue~~ ✓

float ✓

return ✓

typedef ✓

default ✓

for ✓

short ✓

union ✓

Schlüsselwörter ab C99:

\_Bool ✓

\_Complex ✓

\_Imaginary ✓

inline ✓

restrict ✓

Schlüsselwörter ab C11:

\_Alignas

\_Alignof ✓

\_Atomic

\_Generic

\_Noreturn

\_Static\_assert

\_Thread\_local