

C-Kurs

Rekursive Funktionen & Bibliotheken

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 21/22

Funktionen

Wiederholung: Funktionen

- Funktionen bilden das Grundgerüst jedes Programms
- Sie dienen zur
 - Modularisierung
 - Vermeidung von komplexen Kontrollstrukturen
 - Kapselung
 - Dokumentation

Wiederholung Funktionsaufrufe

- Jede Funktion kann von jeder Funktion aufgerufen werden
- Beispiele:
 - `max()` von `main()` aus
 - `max()` von `printf()` aus
- Insbesondere kann eine Funktion auch sich selbst aufrufen! → **Rekursion**

Rekursion

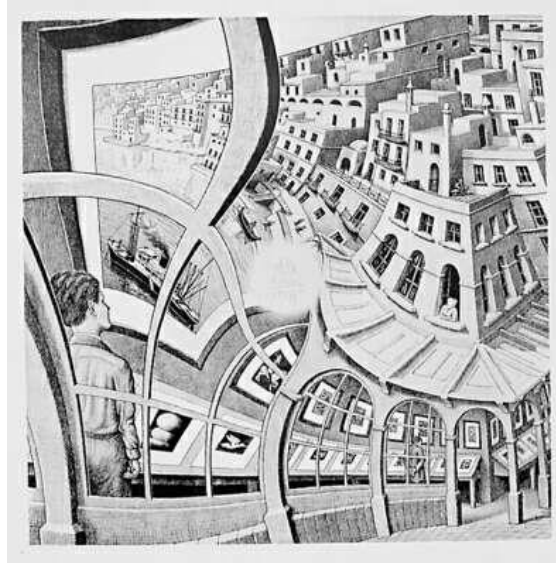
- Spielzeug (Matroschka)



Quelle: German Wikipedia

Rekursion

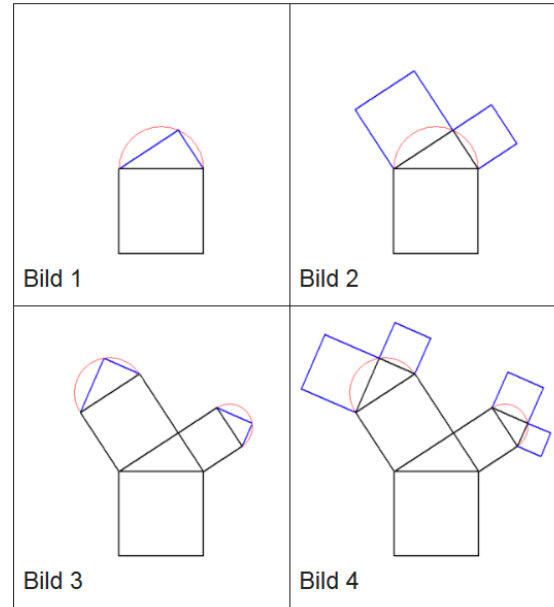
- Kunst



M.C. Escher; Bildgalerie

Rekursion

- Fraktale

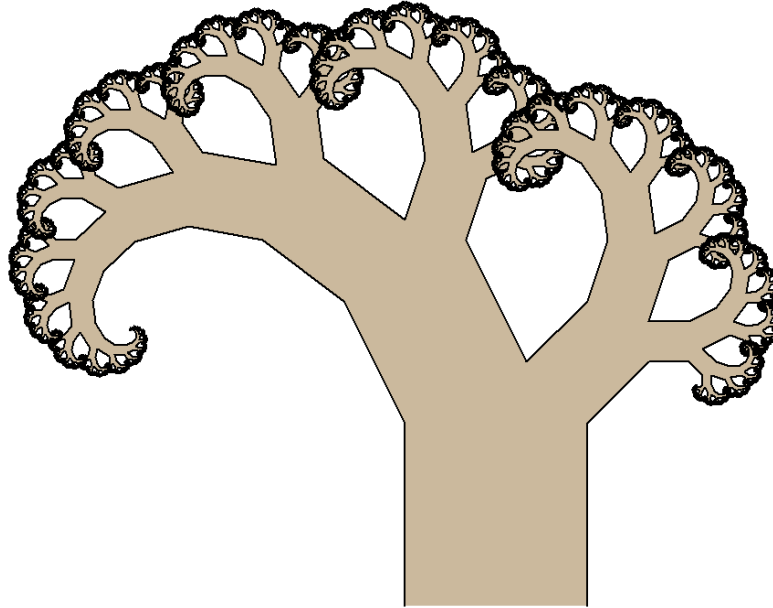


Quelle: German Wikipedia

Pythagoras-Baum

Rekursion

- Fraktale

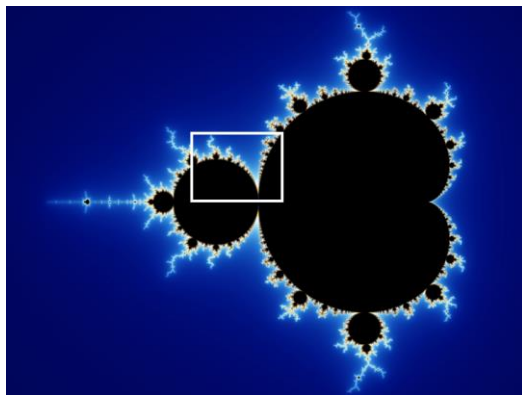


Quelle: German Wikipedia

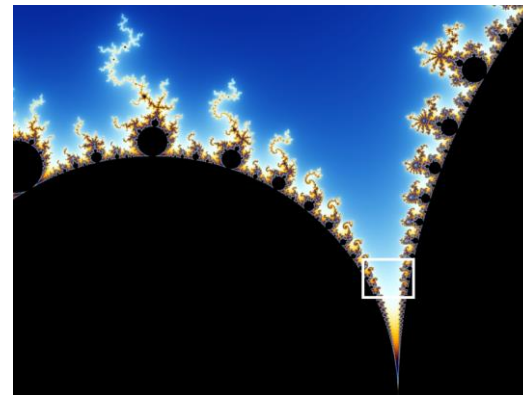
Pythagoras-Baum

Rekursion

- Fraktale



Mandelbrotmenge



Quelle: German Wikipedia, Wolfgang Beyer

<https://math.hws.edu/eck/js/mandelbrot/MB.html>

Mathematische Rekursion

- Viele mathematische Funktionen sind einfach rekursiv definierbar.
- D.h. die Funktion erscheint in ihrer eigenen Definition.
- Beispiel: Fakultätsfunktion

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

Rekursive Funktionen

- Die Funktion ruft sich selbst auf (**Kernkonzept!**)
- Beispiel: (noch falsch...)

```
int recursion(int a) {  
    return recursion(a+1); // rekursiver Aufruf  
}
```

Aufruf: `printf("Recursion: %d\n", recursion(0));`

Ausgabe: ???

Rekursive Funktionen

- Die Funktion ruft sich selbst auf (**Kernkonzept!**)
- Beispiel: (richtig...)

```
int recursion(int a) {  
    if (a > 41) { // Abbruchbedingung  
        return a;  
    }  
    return recursion(a+1); // rekursiver Aufruf  
}
```

Aufruf: `printf("Recursion: %d\n", recursion(0));`

Ausgabe: ???

Rekursive Funktionen

- Die Funktion ruft sich selbst auf (**Kernkonzept!**)
- Zu beachten:
 - Terminierung, d.h. Abbruchbedingung, ist notwendig!
 - Sonst Endlosprogramm
- Typischer Ablauf:

```
int recursion(int a) {  
    if (a > 41) { return a; } // Abbruchbedingung  
    return recursion(a+1);    // rekursiver Aufruf  
}
```

Rekursive Funktionen

Beispiel Fakultät

- Fakultät:

```
int fak(int n) {  
    if (n <= 1) { // Abbruchbedingung  
        return 1;  
    } else {  
        return n * fak(n - 1); // rekursiver Aufruf  
    }  
}
```

Rekursive Funktionen

Beispiel Fakultät

- Fakultät (alternative syntaktische Darstellung):

```
int fak(int n) {  
    if (n <= 1) return 1; // Abbruchbedingung  
    return n * fak(n - 1); // rekursiver Aufruf  
}
```

Rekursive Funktionen

Beispiel Fakultät

- Fakultät (Alternative mit größerem Wertebereich):

```
long fak(int n) {  
    if (n <= 1) return 1; // Abbruchbedingung  
    return n * fak(n - 1); // rekursiver Aufruf  
}
```

- Die Werte werden sehr schnell sehr groß
- Wertebereich long: von -2^{63} bis $2^{63} - 1$ (für 64-bit Architekturen)
- printf** Format für **long** ist **%lu**

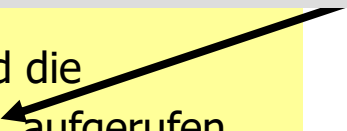
Rekursive Funktionen

- Unendliche Rekursion
 - Ist so leicht zu erzeugen, wie eine unendliche Schleife
 - Hinweis: **Nie Abbruchbedingung vergessen!**
- Wir brauchen Fortschritt, d.h. das Problem, das mit dem rekursiven Aufruf gelöst werden soll, muss „einfacher“ bzw. „kleiner“ werden, z.B:

fak (n) :

Terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit einem Argument **< n** aufgerufen.

“n wird mit jedem Aufruf kleiner.”

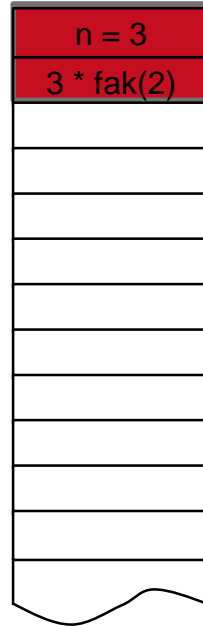


Auswertung rekursiver Funktionsaufrufe: fak(3)

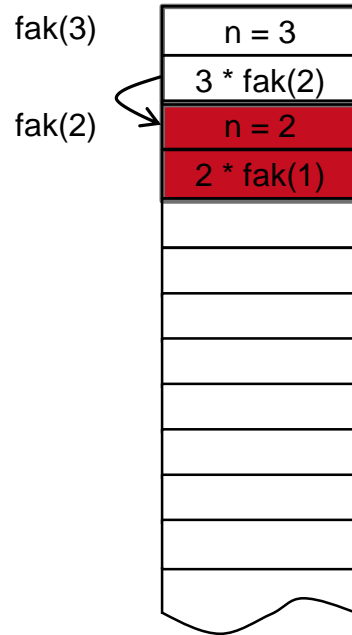
```
// return value is n!  
int fak(int n) {  
    if (n <= 1) return 1;  
    return n * fak(n-1); // n > 1  
}
```

Beispiel: fak(3)

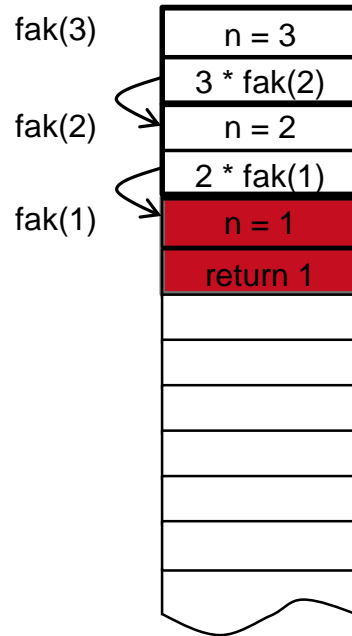
fak(3)



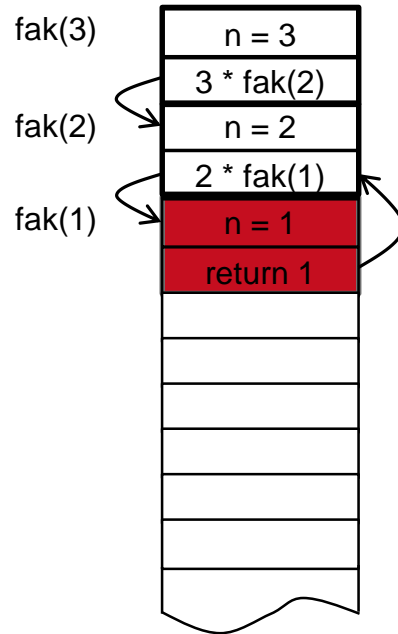
Beispiel: fak(3)



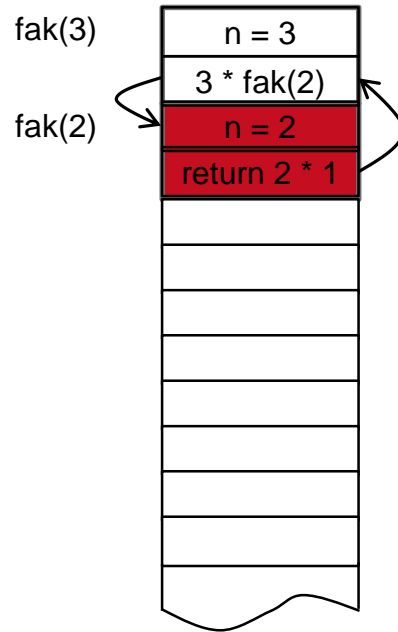
Beispiel: fak(3)



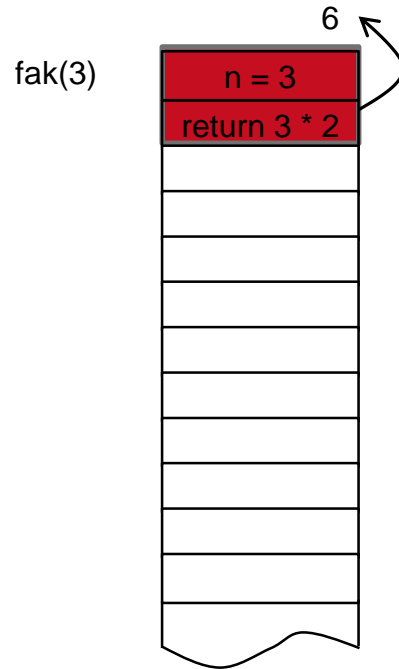
Beispiel: fak(3)



Beispiel: fak(3)



Beispiel: fak(3)



Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 3
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!  
int fak(int n) {  
    // n = 3  
    if (n <= 1) return 1;  
    return n * fak(n-1); // n > 1  
}
```

Ausführen des Funktionsrumpfs:
Auswertung des Rückgabeausdrucks

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!  
int fak(int n) {  
    // n = 3  
    if (n <= 1) return 1;  
    return n * fak(n-1); // n > 1  
}
```

Ausführen des Funktionsrumpfs: Rekursiver Aufruf von fak mit Aufrufargument $n-1 == 2$

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 2
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 2
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Es gibt jetzt zwei $n!$ Das von fak(3), und das von fak(2)

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 2
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Wir nehmen das Argument des *aktuellen* Aufrufs, fak(2)

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 1
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 1
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Es gibt jetzt drei $n!$ Das von fak(3), fak(2) und fak(1)

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!
```

```
int fak(int n) {
```

```
    // n = 1
```

```
    if (n <= 1) return 1;
```

```
    return n * fak(n-1); // n > 1
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Wir nehmen das Argument des *aktuellen* Aufrufs, fak(1)

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!  
int fak(int n) {  
    // n = 1  
    if (n <= 1) return 1; // n == 1, d.h.  
        // Abbruch und Rückgabe des Wertes 1  
        // Kein rekursiver Aufruf von fak() mehr!  
    return n * fak(n-1); // n > 1  
}
```

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!  
int fak(int n) {  
    // n = 2  
    if (n <= 1) return 1;  
    return n * fak(n-1); // n > 1  
                        // d.h. return 2 * 1;  
}
```

Auswertung rekursiver Funktionsaufrufe: fak(3)

```
// return value is n!  
int fak(int n) {  
    // n = 3  
    if (n <= 1) return 1;  
    return n * fak(n-1); // n > 1  
                        // d.h. return 3 * 2 * 1;  
}
```

Bibliotheken

Modularisierung

- C-Programme bestehen aus einer Menge von Funktionen
- Funktionen können in verschiedene Module (Dateien) getrennt werden
- Warum?
 - Übersichtlichkeit / Lesbarkeit
 - Erweiterbarkeit
 - Wiederverwendbarkeit
 - Wartbarkeit

Die `main()` –Funktion

- Eine Funktion ist ausgezeichnet: Die `main`-Funktion.
- Jedes C-Programm braucht eine `main`-Funktion.
- Sie ist die erste Funktion, die von der Shell aus aufgerufen wird.
- Sie bekommt als Parameter die Argumente mit denen das Programm aufgerufen wird.
- Von ihr aus werden alle weiteren Funktionen aufgerufen.
- Sie sollte sich nicht selbst rekursiv aufrufen.

fak.c

```
int fak(int n) {  
    if (n <= 1) return 1;  
    return n * fak(n-1);  
}  
  
int main() {  
    return fak(3);  
}
```


Modularisierung am Bsp. Fakultät

fak-main.c

```
int main() {  
    return fak(3);  
}
```

fak-function.c

```
int fak(int n) {  
    if (n <= 1) return 1;  
    return n * fak(n-1);  
}
```

Modularisierung am Bsp. Fakultät

fak-header.h

```
int fak(int);
```

fak-main.c

```
int main() {  
    return fak(3);  
}
```

fak-function.c

```
int fak(int n) {  
    if (n <= 1) return 1;  
    return n * fak(n-1);  
}
```

Modularisierung am Bsp. Fakultät

fak-header.h

```
int fak(int);
```

fak-main.c

```
#include "fak-header.h"
int main() {
    return fak(3);
}
```

fak-function.c

```
int fak(int n) {
    if (n <= 1) return 1;
    return n * fak(n-1);
}
```

Modularisierung am Bsp. Fakultät

fak-header.h

```
int fak(int);
```

fak-main.c

```
#include "fak-header.h"
int main() {
    return fak(3);
}
```

fak-function.c

```
int fak(int n) {
    if (n <= 1) return 1;
    return n * fak(n-1);
}
```

= Implementierung von Fakultät

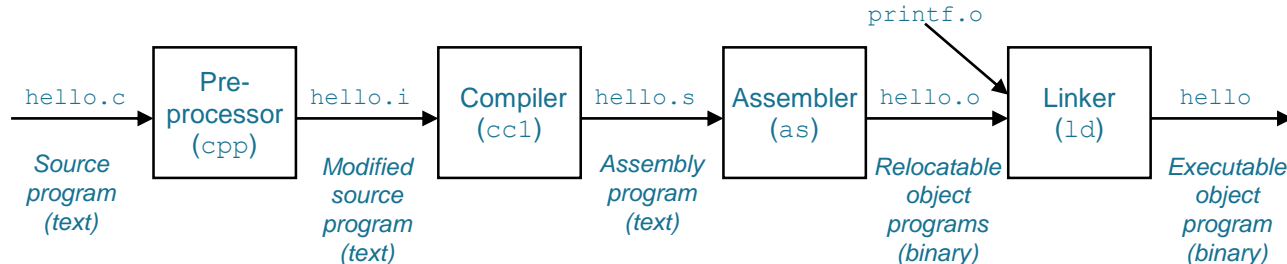
Wiederholung: C-Compiler

- Beispiel: GCC – GNU Compiler Collection

```
unix> gcc -Wall -std=c11 -o hello hello.c
```

4 Phasen:

- Preprocessor Aufbereitung
- Compiler Übersetzt C in Assemblercode
- Assembler Übersetzt Assemblercode in Maschinensprache
- Linker Nachbearbeitung / Kombination verschiedener Module



Wiederholung: C-Compiler

- Beispiel: Clang - a C language family frontend for LLVM

```
unix> clang -Wall -std=c11 -o hello hello.c
```

5+ Phasen:

- Preprocessor
- Compiler
- Optimizers
- Backend
- Assembler
- Linker

Aufbereitung

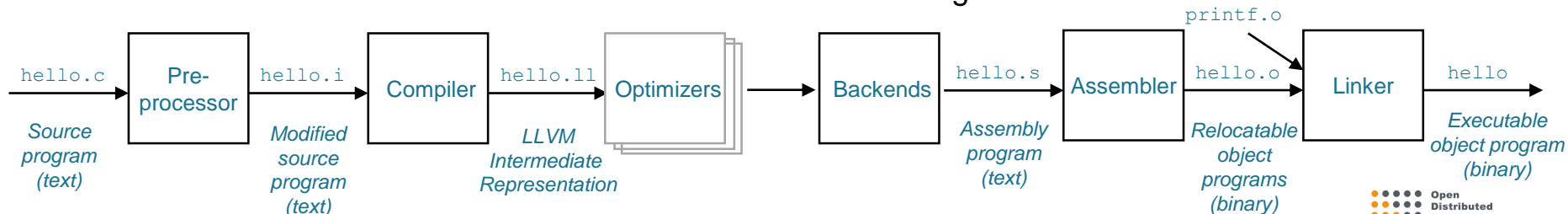
Übersetzt C in LLVM Intermediate Representation (IL)

Optimieren den Sprachunabhängige LLVM IL

Übersetzt LLVM IL in Assemblercode

Übersetzt Assemblercode in Maschinensprache

Nachbearbeitung / Kombination verschiedener Module



C-Module: Übersetzung

- Module können einzeln übersetzt werden

```
clang -c modul.c
```

Dieser Aufruf generiert Maschinencode im File: **modul.o**

- **Problem:** Module benutzen externe Funktionen
- **Lösung:** Header-Dateien, (Endung: **.h**), die
 - Funktionsprototypen (Signatur der Funktion)
 - Enthalten Deklarationen
- Beispiele: **string.h**, **stdio.h**, **math.h**, ...
- Header-Dateien werden mittels **#include** eingebunden

C-Module: Übersetzung

- Module werden mit Hilfe des Linkers verknüpft
`clang -o fak fak-main.o fak-funktion.o`
- Gemischte Übersetzung/Bindung ist möglich
`clang -o fak fak-main.c fak-funktion.o`
`clang -o fak fak-main.c fak-funktion.c`
- Headerdateien enthalten keine Anweisungen und können daher einzeln nicht in Maschinencode übersetzt werden.

Präprozessor

- Der Präprozessor bearbeitet die sogenannten Direktiven.
- Es geht hierbei um textuelle Ersetzungen.
- Beispiele: `#define`, `#include`
(Syntax: `#directive dir_parameters`)
- Beispiel: `#define MAX_LEN 10`
- Ersetzt im Code das Symbol `MAX_LEN` durch `10`
- Sinnvoll für Konstanten

Präprozessor: `#include`

- Include-Direktive:
`#include <StandardHeader>`
`#include "test.h"`
- Ersetzt die Include-Zeile durch den Inhalt des Header-Files.
- `<>` sucht Dateien im Standardsuchpfad.
- `" "` sucht Dateien im Verzeichnis der `.c`-Datei.
- Mit `-I` kann man weitere Suchpfade angeben.

Nutzung einer Bibliothek

- Nutzung einer Bibliotheksfunktion
 - Im C-Code
`#include <glib.h>`
 - Beim Compilieren/Linken
`clang -Wall -std=c11 -o fak fak.c -lglib`
 - Der Linker sucht dann automatisch in den vorgesehenen Directories.
- Um weitere Directories hinzuzufügen:
 - Explizit mittels `-L` für Bibliotheken und `-I` für Header-Dateien

Hinweise zur nächsten C-Kursaufgabe

- Include-Datei:

```
#include "input.h"
```

- Beim Compilieren/Linken:

```
clang -Wall -std=c11 -o datei datei.c input.c
```

- Das wird dann einzeln compiliert und zusammen gelinkt.

Hinweise zur nächsten C-Kursaufgabe

- Die Dateien input.h und input.c aus dem ISIS Kurs herunterladen
- Diese Dateien
 - In Ihr Verzeichnis kopieren in dem Sie arbeiten.
-> Dateien müssen zum Compilieren im selben Verzeichnis liegen
 - Nicht modifizieren
 - Nicht abgeben (wir haben lokale Versionen auf dem Testserver)

[Abgabebranch: ckurs-b04-a01]

Hinweise zu Codevorgaben

- Wenn Vorgaben gemacht werden, dann bitte einhalten:
 - Variablennamen
 - Konstantennamen
 - Ausgabeformate
 - Codesegmente
- Warum:
 - Wegen der automatisierten Tests.
 - Um Ihnen Feedback geben zu können.