

# Programmierkurs: Rekursive Funktionen & Modularisierung

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 25/26

---

# Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine

VL 1 „Hello World“: „Lebenswichtiges“, Programtablauf, Programmierablauf, Kompilierung und Ausführung von Programmen

VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen

VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit

**VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung**

VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition

VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“

VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer

VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen

VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git

VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

# Funktionen

# Wiederholung: Funktionen

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

# Wiederholung: Funktionen

- Definition

```
// function to calculate
// the maximum of a and b
int max (int a, int b) {
    if (a > b) {
        return a; // a is max
    } else {
        return b; // b is max
    }
}
```

- Aufruf

```
int main() {
    int n = 10;
    int m = 11;

    printf("max of n, m: %d\n",
        max(n, m));
}
```

# Funktionsaufruf

```
... // Definition of max
int main() {
    int r1, r2;
    int n = 10;
    int m = 11;

    r1 = max(10, 11); // Aufruf mit festen Werten
    r2 = max(n, m);   // Aufruf mit Variablen

    printf("1: max of 10, 11: %d\n", r1);
    printf("2: max of n, m: %d\n", r2);
    // Aufruf innerhalb eines anderen Aufrufs
    printf("3: max of n, m: %d\n", max(n, m));
}
```

# 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)

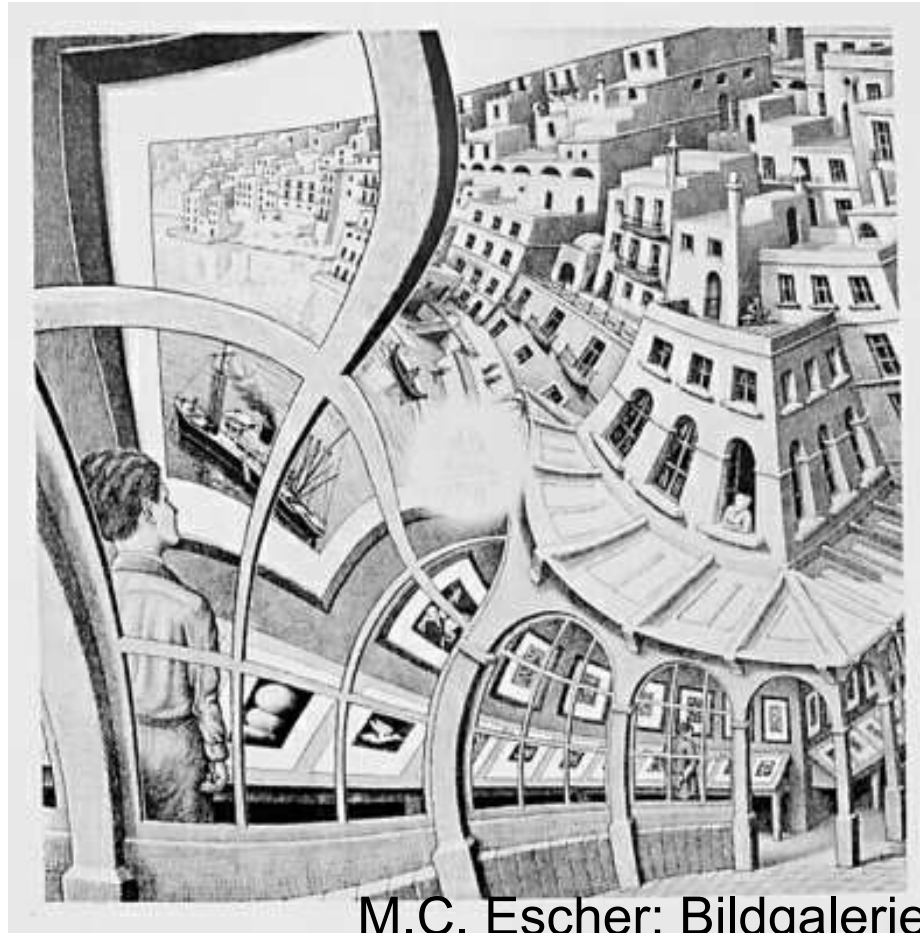


Quelle: German Wikipedia



# Rekursion

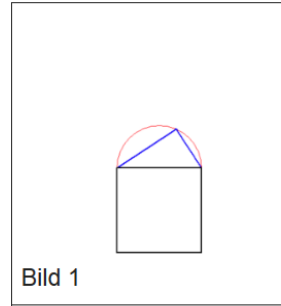
- Kunst



M.C. Escher; Bildgalerie

# Rekursion

- Fraktale

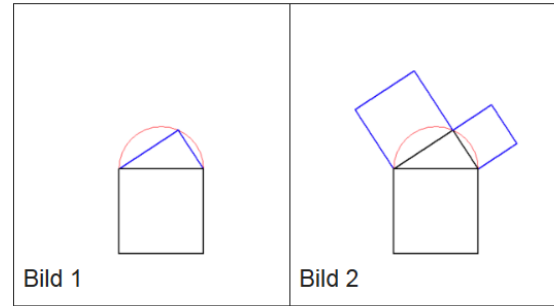


## Pythagoras-Baum

Quelle: [German Wikipedia](#)

# Rekursion

- Fraktale

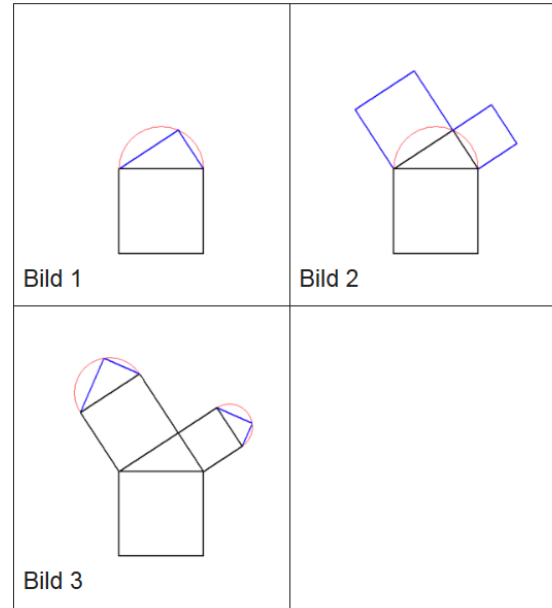


## Pythagoras-Baum

Quelle: [German Wikipedia](#)

# Rekursion

- Fraktale

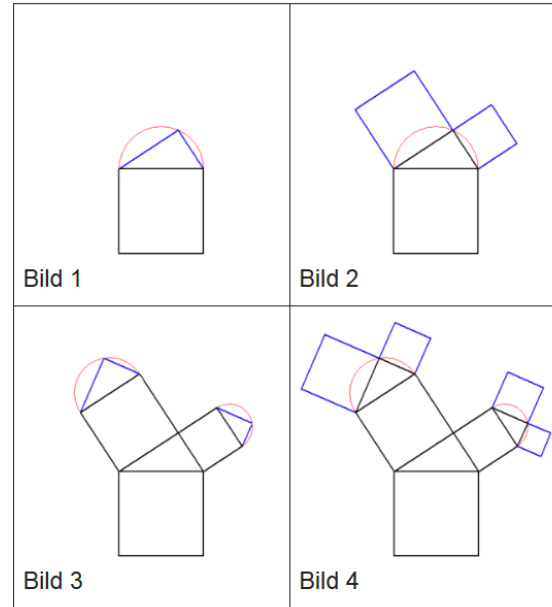


## Pythagoras-Baum

Quelle: [German Wikipedia](#)

# Rekursion

- Fraktale

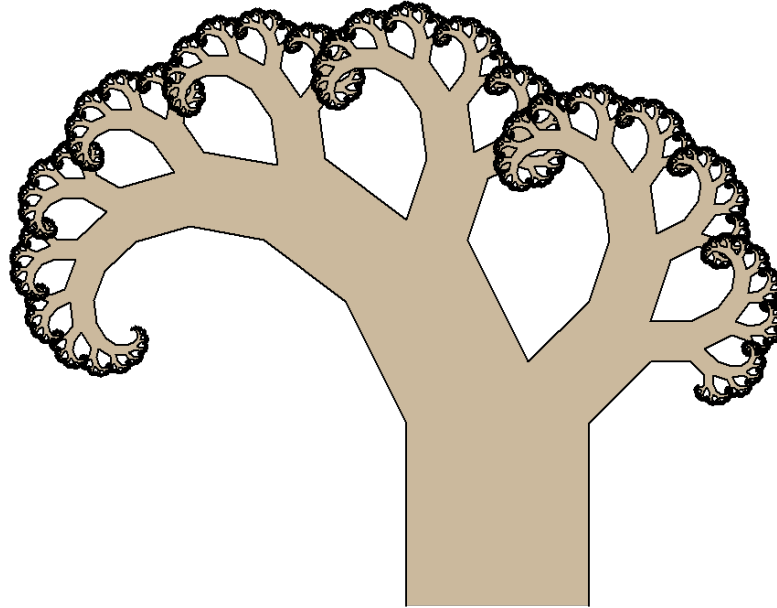


## Pythagoras-Baum

Quelle: [German Wikipedia](#)

# Rekursion

- Fraktale



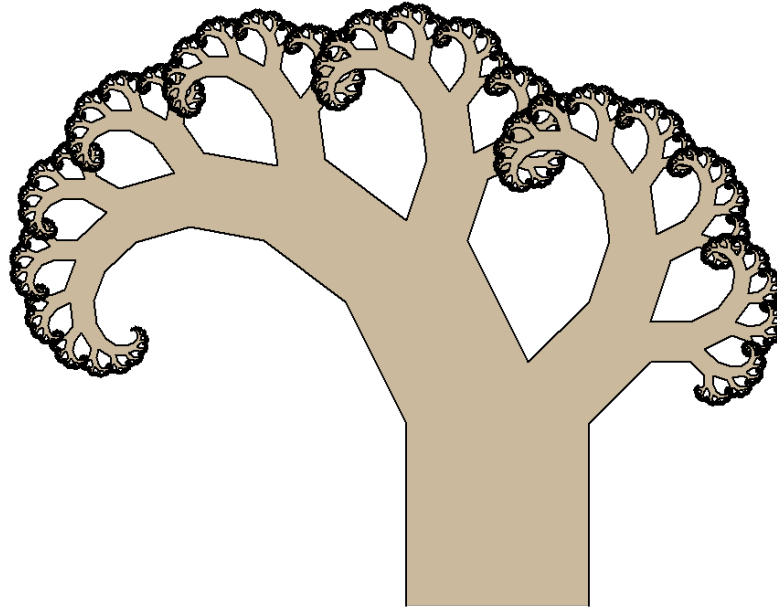
Pythagoras-Baum

Quelle: [German Wikipedia](#)

# Rekursion

- Fraktale

... aus der Natur: Romanesco

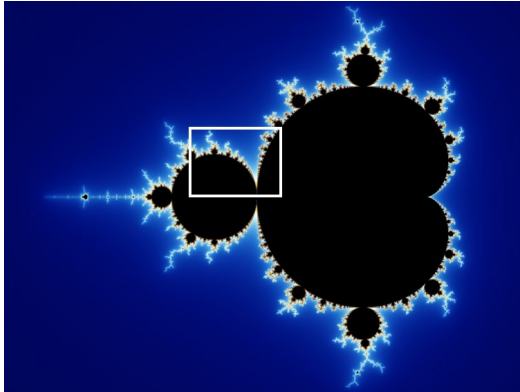


Pythagoras-Baum

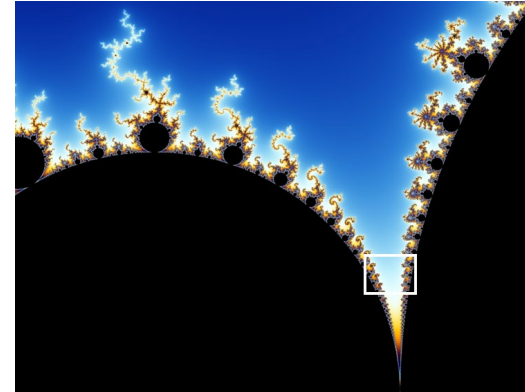
Quelle: [German Wikipedia](#)

# 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: Potenzen  $2^n$  für  $n \geq 0$  berechnen

$$2^n = \begin{cases} 1, & \text{falls } n = 0 \\ 2 \cdot 2^{(n-1)} & \text{falls } n > 1 \end{cases}$$

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

# Rekursive Funktionen: Beispiel $2^n$

- Zweierpotenzen:

```
int pot_2(int n) {  
    if (n == 0) { // Abbruchbedingung  
        return 1;  
    }  
    else {  
        return 2 * pot_2(n - 1); // rekursiver Aufruf  
    }  
}
```

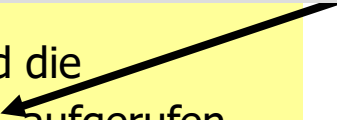
# Rekursive Funktionen

- 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:

`pot_2(n):`

Terminiert sofort für  $n = 0$ , andernfalls wird die Funktion rekursiv mit einem Argument  $< n$  aufgerufen.

“n wird mit jedem Aufruf kleiner.”

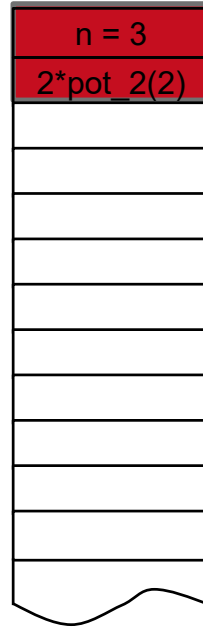


# Auswertung rekursiver Funktionsaufrufe: `pot_2(3)`

```
// return value is 2^n  
int pot_2(int n) {  
    if (n == 0) return 1;  
    return 2 * pot_2(n - 1); // n > 0  
}
```

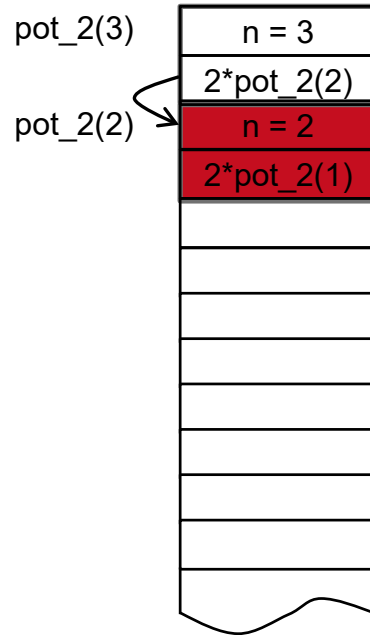
# Beispiel: `pot_2(3)`

`pot_2(3)`

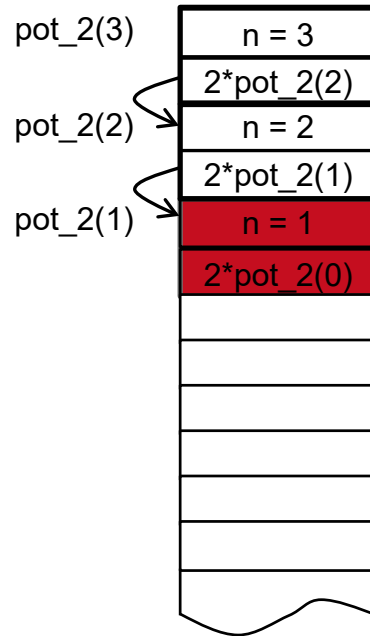




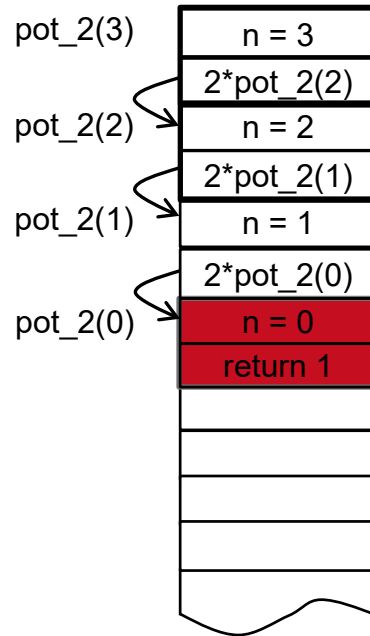
# Beispiel: `pot_2(3)`



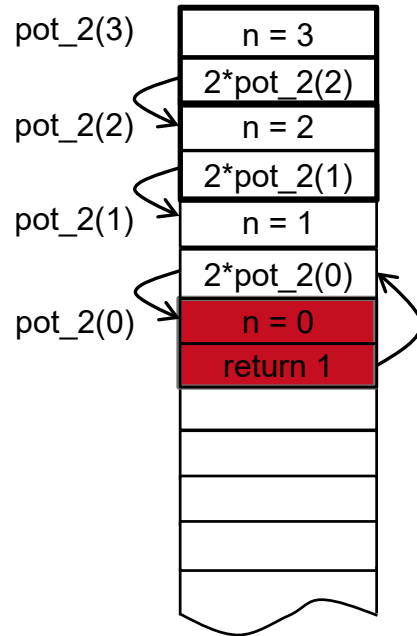
# Beispiel: `pot_2(3)`



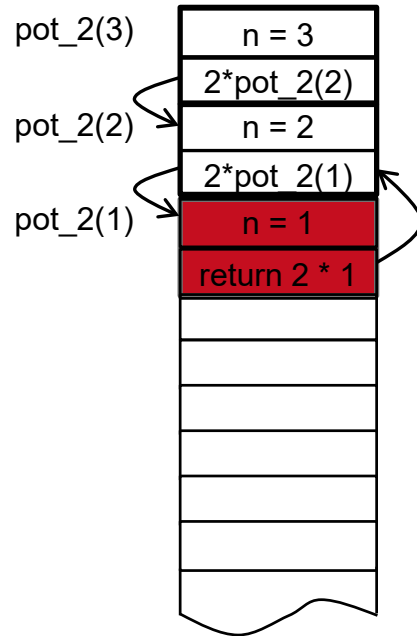
# Beispiel: `pot_2(3)`



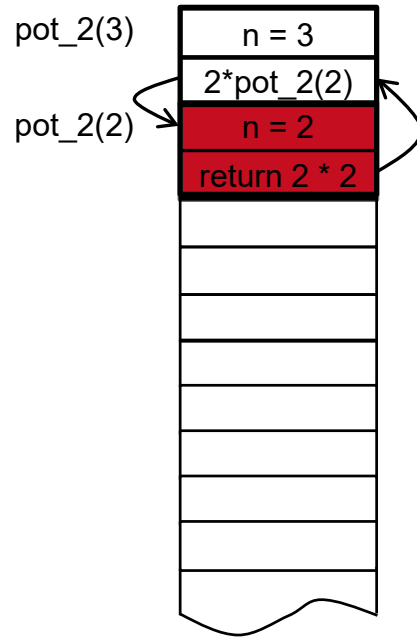
# Beispiel: `pot_2(3)`



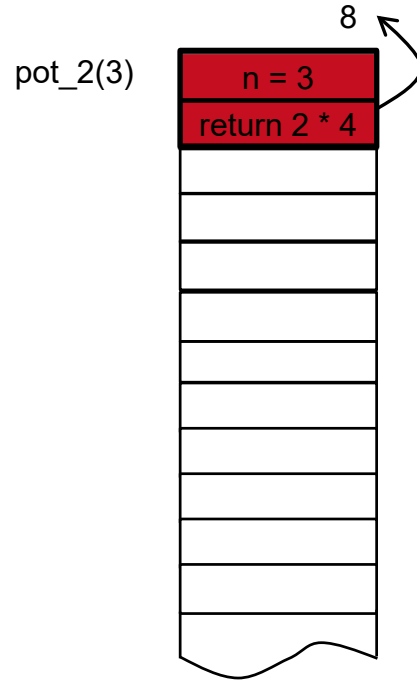
# Beispiel: `pot_2(3)`



# Beispiel: `pot_2(3)`



# Beispiel: `pot_2(3)`



# Auswertung rekursiver Funktionsaufrufe: fak (5)



Video



# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 3
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
int pot_2(int n) {
    // n = 3
    if (n == 0) return 1;
    return 2 * pot_2(n-1); // n > 0
}
```

Ausführen des Funktionsrumpfs:  
Auswertung des Rückgabeausdrucks

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
int pot_2(int n) {
    // n = 3
    if (n == 0) return 1;
    return 2 * pot_2(n-1); // n > 0
}
```

Ausführen des Funktionsrumpfs: Rekursiver  
Aufruf von `pot_2` mit Aufrufargument `n-1 == 2`

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 2
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 2
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Es gibt jetzt zwei `pot_2`: Das von `pot_2 (3)`, und das von `pot_2 (2)`

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 2
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Wir nehmen das Argument des *aktuellen* Aufrufs, `pot_2 (1)`

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 1
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 1
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Es gibt jetzt drei `pot_2`: Das von `pot_2 (3)`, das von `pot_2 (2)` und das von `pot_2 (1)`



# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 1
```

```
    if (n == 0) return 1;
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

Initialisierung des Arguments mit dem Wert des Aufrufarguments

Wir nehmen das Argument des *aktuellen* Aufrufs, `pot_2 (0)`

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
```

```
int pot_2(int n) {
```

```
    // n = 0
```

```
    if (n == 0) return 1; // n == 0, d.h.
```

```
        // Abbruch und Rückgabe des Wertes 1
```

```
        // Kein rekursiver Aufruf von pot_2 mehr!
```

```
    return 2 * pot_2(n-1); // n > 0
```

```
}
```

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
int pot_2(int n) {
    // n = 1
    if (n == 0) return 1;
    return 2 * pot_2(n-1); // n > 1
                          // d.h. return 2 * 1;
}
```

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
int pot_2(int n) {
    // n = 2
    if (n == 0) return 1;
    return 2 * pot_2(n-1); // n > 1
    // d.h. return 2 * 2 * 1;
}
```

# Auswertung rekursiver Funktionsaufrufe: `pot_2 (3)`

```
// return value is 2^n
int pot_2(int n) {
    // n = 3
    if (n == 0) return 1;
    return 2 * pot_2(n-1); // n > 1
    // d.h. return 2 * 2 * 2 * 1;
}
```

# Modularisierung

# Die `main()` -Funktion

- Eine Funktion ist ausgezeichnet: `main()`
- Jedes C-Programm braucht eine `main()` -Funktion.
- Sie ist die erste Funktion, die 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.

# Modularisierung

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



# Bisher (ohne Modularisierung)

pot\_2-main.c (alles in 1 Datei)

```
int pot_2(int n) {  
    if (n == 0) return 1;  
    return 2 * pot_2(n-1);  
}
```

Implementierung  
der Funktion pot\_2

```
int main() {  
    return pot_2(3);  
}
```

Verwendung / Aufruf  
der Funktion pot\_2

# Modularisierung an einem Beispiel

wird durch den Inhalt der  
Datei pot\_2-header.h  
erstellt

```
int pot_2(int n) {
    if (n == 0) return 1;
    return 2 * pot_2(n-1);
}

#include "pot_2-header.h"

int main() {
    return pot_2(3);
}
```

Verwendung / Aufruf  
der Funktion pot\_2

pot\_2-header.h

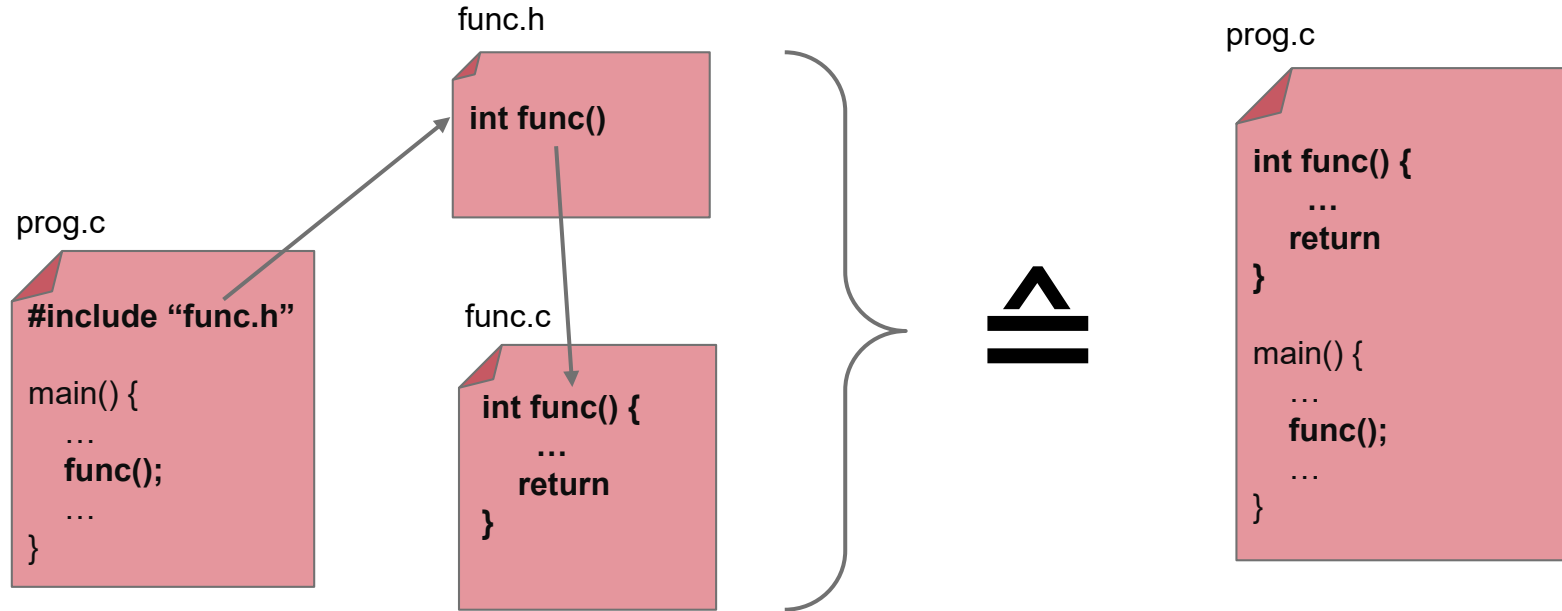
```
int pot_2(int);
```

Definition der  
Funktion pot\_2

pot\_2-function.c

Implementierung  
der Funktion pot\_2

# Modularisierung: Muster



# Modularisierung: Muster

func.h

```
int func()
```

func.c

```
int func() {  
    ...  
    return  
}
```

prog1.c

```
#include "func.h"  
  
main() {  
    ...  
    func();  
    ...  
}
```

prog2.c

```
#include "func.h"  
  
main() {  
    ...  
    func();  
    ...  
}
```

prog3.c

```
#include "func.h"  
  
main() {  
    ...  
    func();  
    ...  
}
```

# Compiler

- Alles in 1 Datei (nicht modularisiert)

```
clang -Wall -std=c11 -o prog prog.c
```

- Modularisiert

```
clang -Wall -std=c11 -o prog prog.c func.c
```

# Hilfreich: #define

- Beispiel: `#define MAX_LEN 10`
- Ersetzt im Code **MAX\_LEN** durch 10
- Sinnvoll für Konstanten
  - Sprechender Name
  - Muss nur an 1 Stelle geändert werden

# Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine

VL 1 „Hello World“: „Lebenswichtiges“, Programtablauf, Programmierablauf, Kompilierung und Ausführung von Programmen

VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen

VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit

**VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung**

VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition

VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“

VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer

VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen

VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git

VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

# Slides für Interessierte



# Rekursive Funktionen: Beispiel $2^n$

- Zweierpotenzen (alternative Schreibweise):

```
int pot_2(int n) {  
    if (n == 0) return 1;           // Abbruchbedingung  
    return 2 * pot_2(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**

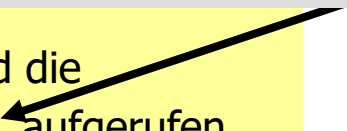
# Rekursive Funktionen

- 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.”



## 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

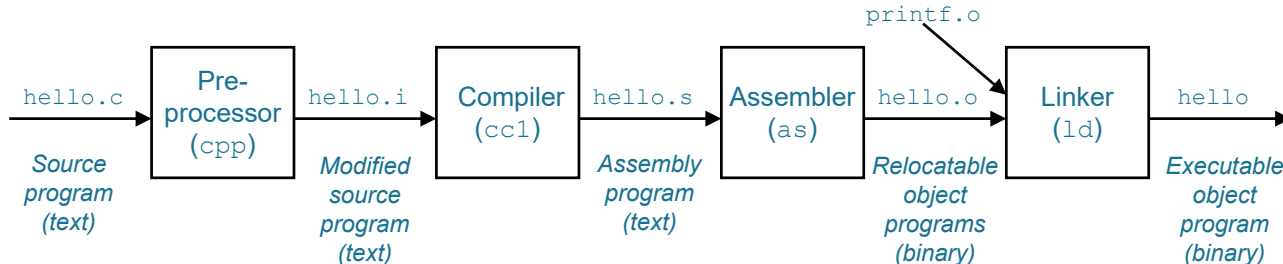
# Wiederholung: C-Compiler

- 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

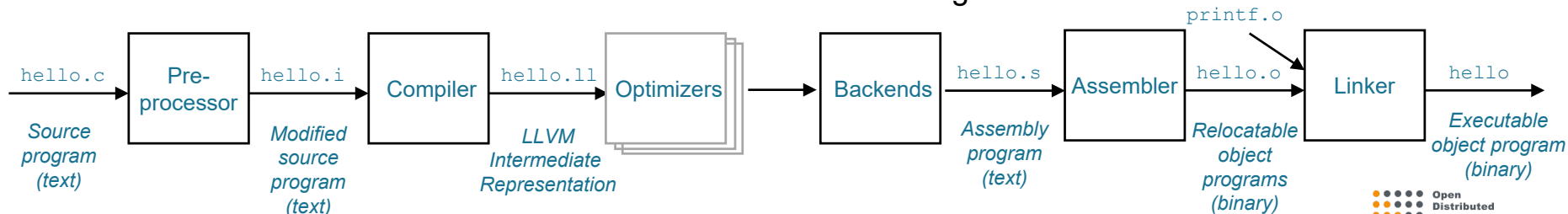
Übersetzt C in LLVM Intermediate Representation (IL)

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