

C-Kurs

Rekursive Funktionen

& Bibliotheken

Funktionen

Wiederholung: Funktionen

- ❑ Funktionen bilden das Grundgerüst jedes Programms:
 - 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 selber aufrufen! => **Rekursion**

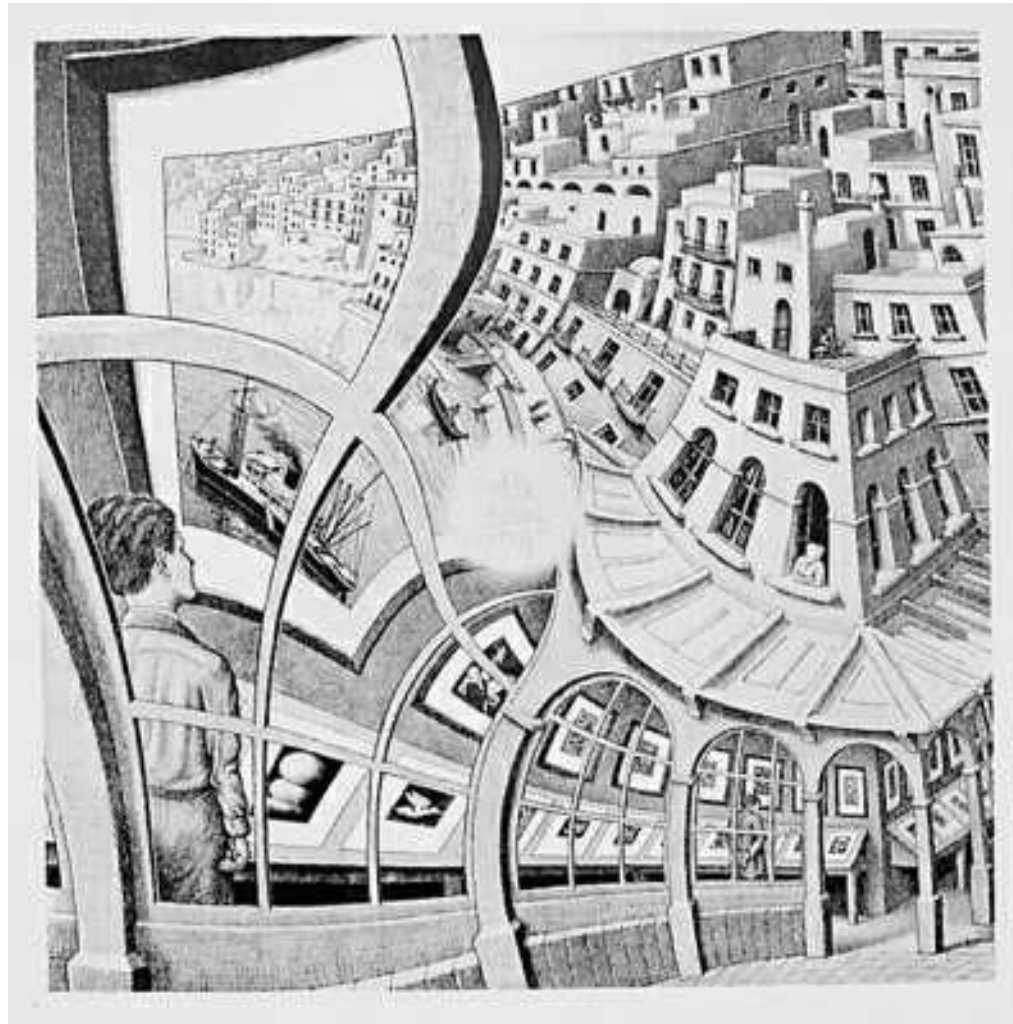
□ Spielzeug:



Matroschka

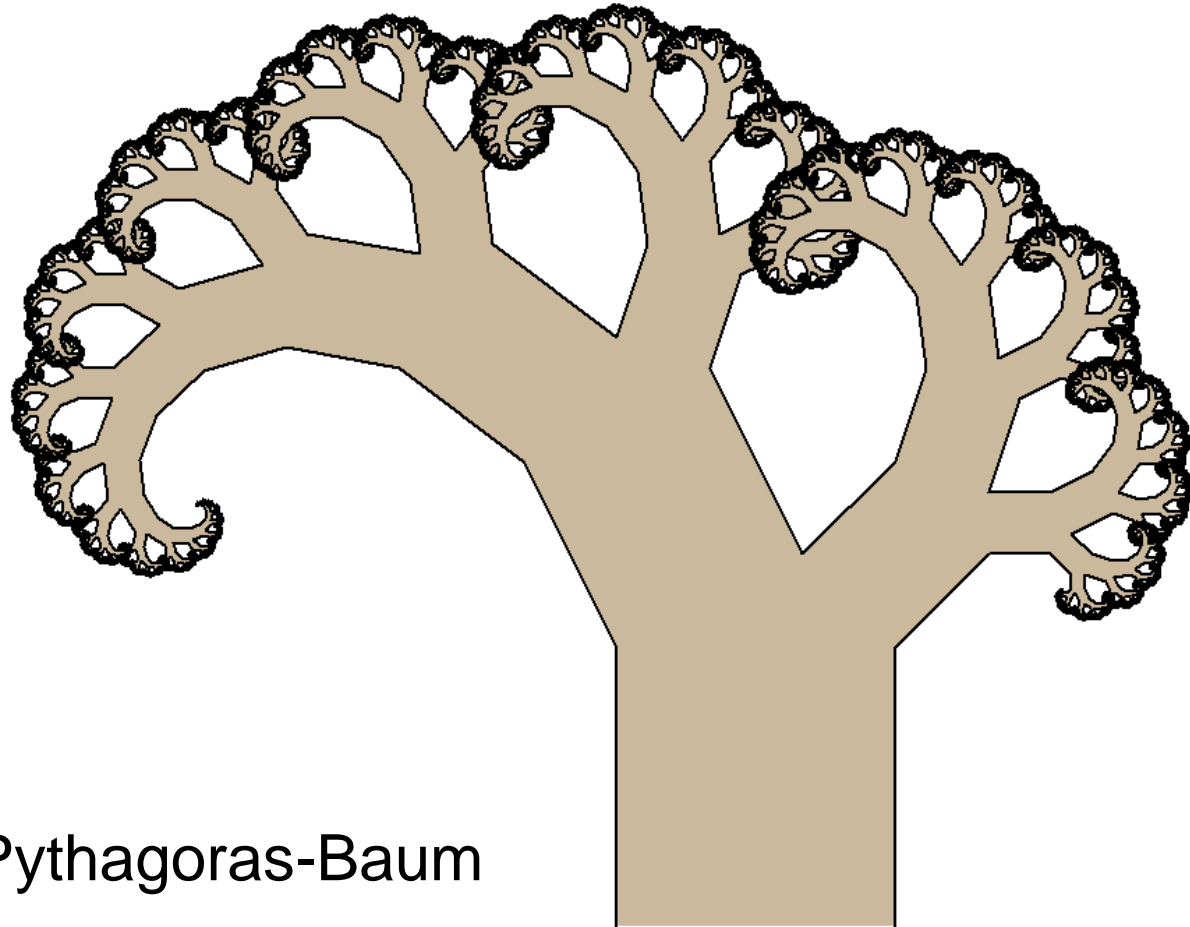
Quelle: German Wikipedia

□ Kunst:



M.C. Escher; Bildgalerie

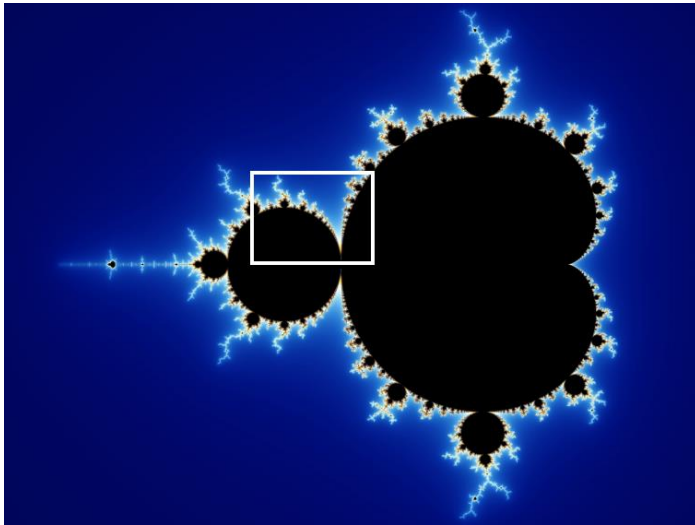
□ Fraktale:



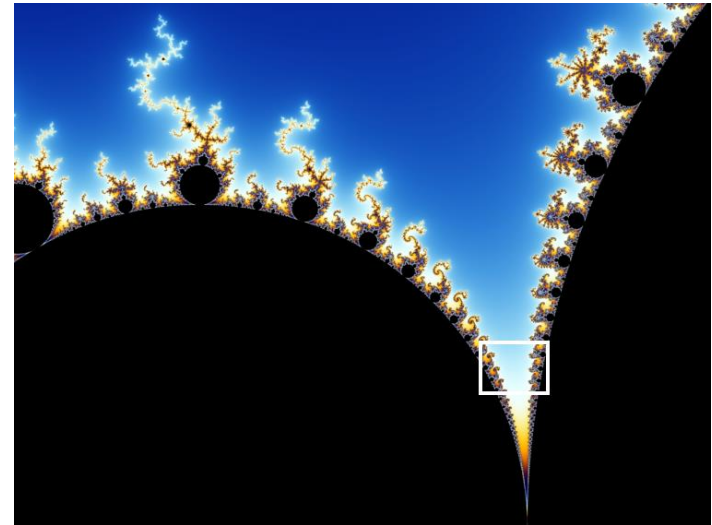
Pythagoras-Baum

Quelle: German Wikipedia

□ Fraktale:



Mandelbrotmenge



Quelle: German Wikipedia, Wolfgang Beyer

<http://www.mathematik.ch/anwendungenmath/fractal/julia/MandelbrotApplet.php>

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 selber 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 selber 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 selber auf **// Kernkonzept!**

❑ Zu beachten:

- Terminierung, d.h. Abbruchbedingung!
- Sonst Endlosprogramm

❑ D.h.: 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 größerer Wertebereich)

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

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

Rekursive Funktionen

□ Unendliche Rekursion

- Ist so leicht zu erzeugen, wie eine unendliche Schleife
- Ist zu vermeiden. Also nie Abbruchbedingung vergessen!

□ Wir brauchen Fortschritt, d.h. das Problem, was mit dem rekursiven Aufruf gelöst wird, muss einfacher werden, z.B:

fak (n) :

Terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit 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  
  
}
```

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 Rückgabedruckes

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
```

```
}
```

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

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
```

```
}
```

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

Initialisierung *des* Arguments mit dem Wert des Aufrufarguments

Bibliotheken

- ❑ 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 selber aufrufen.

fak.c

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

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

Modularisierung am Beispiel Fakultät

fak-main.c

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

fak-funktion.c

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

Modularisierung am Beispiel Fakultät

fak-head.h

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

fak-main.c

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

fak-funktion.c

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

Modularisierung am Beispiel Fakultät

fak-head.h

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

fak-main.c

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

fak-funktion.c

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

Modularisierung am Beispiel Fakultät

fak-head.h

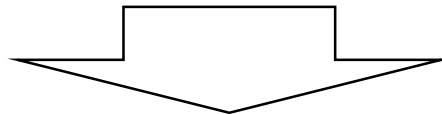
```
int fak(int);
```

fak-main.c

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

fak-funktion.c

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



Implementierung von Fakultät

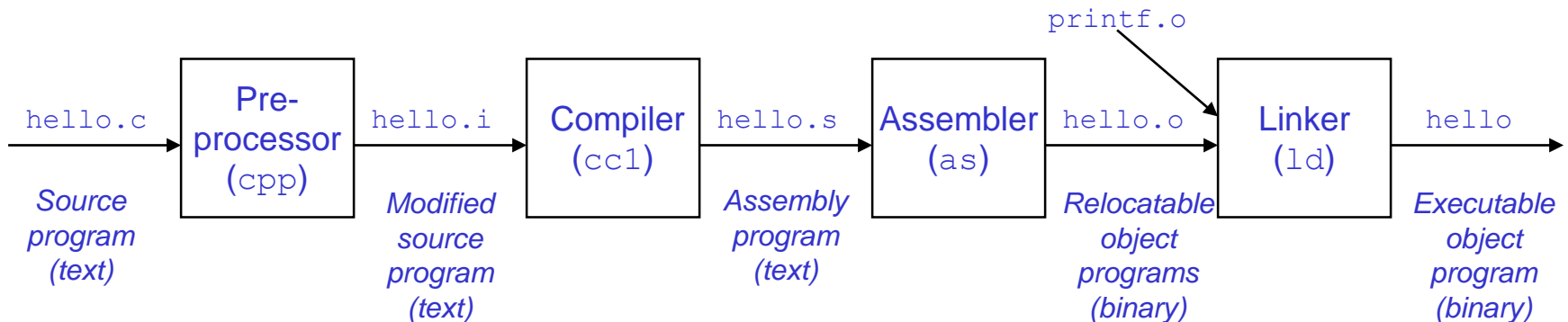
Wiederholung: Ablauf Übersetzung

□ Beispiel: GNU Compiler Collection (gcc)

```
unix> gcc -std=c99 -o hello hello.c
```

➤ 4 Phasen:

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



C-Module: Übersetzung

- ❑ Module können einzeln übersetzt werden
 - `gcc -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)
 - Deklarationenenthalten
- ❑ Beispiele: `string.h`, `stdio.h`, `math.h`,
- ❑ Header-Dateien werden mittels **#include** eingebunden

C-Module: Übersetzung

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

- ❑ Der Präprozessor bearbeitet die Direktiven
- ❑ Beispiele: `#define`, `#include`
- ❑ (Syntax: `#directive dir_parameters`)

- ❑ Beispiel: `#define MAXLEN 10`
- ❑ Ersetzt im Code das Symbol MAXLEN 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

Hinweise zur nächsten C-Kursaufgabe

- ❑ Include-Datei:

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

- ❑ Beim Compilieren/Linken

```
gcc -Wall -std=c99 -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 finden sie im svn im Aufgabenverzeichnis

-ckurs-ws1516/Aufgaben/Blatt04/vorgaben/

- ❑ Diese Dateien

- In Ihr Verzeichnis kopieren in dem Sie arbeiten (mittels cp)

Da die Dateien zum compilieren im selben Directory liegen müssen

- Aber nicht modifizieren!!!

- Aber nicht abgeben
(wir haben lokale Versionen auf dem Testserver)

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