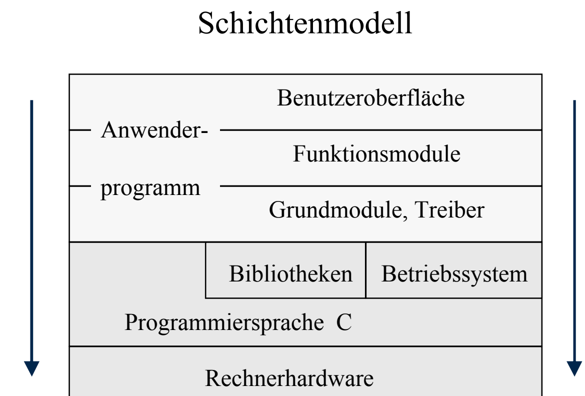
Funktionen

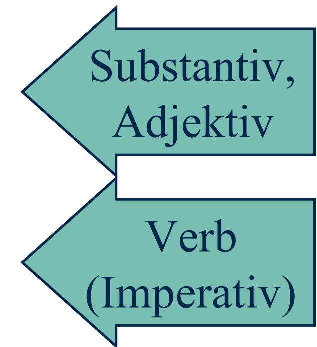
Zweck und Anwendung

- Modularisierung
  - Aufteilung des Gesamtprogramms in überschaubare Einheiten
    - klein bei komplexer Struktur, größer bei einfacher Struktur
  - Austauschbarkeit - Schichtenmodell der Software (prä-OO)
    - einfache, klare Schnittstellen
  - Arbeitsteilung zwischen Programmierern
    - robuste, fehlertolerante Funktion
- Abstraktion
  - Schrittweise Verfeinerung
  - Verbergen von Details der Implementierung
- Mehrfachverwendung
  - in einem Programm
  - in verschiedenen Programmen mittels Bibliotheken



Begriffe: Funktion und Prozedur

- Vergleich mit Pascal:
  - Funktion liefert einen Wert, entspricht dem mathem. Begriff;
    Funktionsaufruf ist ein Ausdruck: x = sin (y);
  - Prozedur liefert keinen Wert, bildet Arbeitsanweisung nach;
    Prozeduraufruf ist eine Anweisung: GibAus (daten);



- in C++ begrifflich nicht unterschieden
  - Prozedur ist Funktion mit (Pseudo-)Ergebnistyp void
  - Funktionswert kann weiterverwendet werden, muß aber nicht

    ok = read_file (datei, puffer); ☺

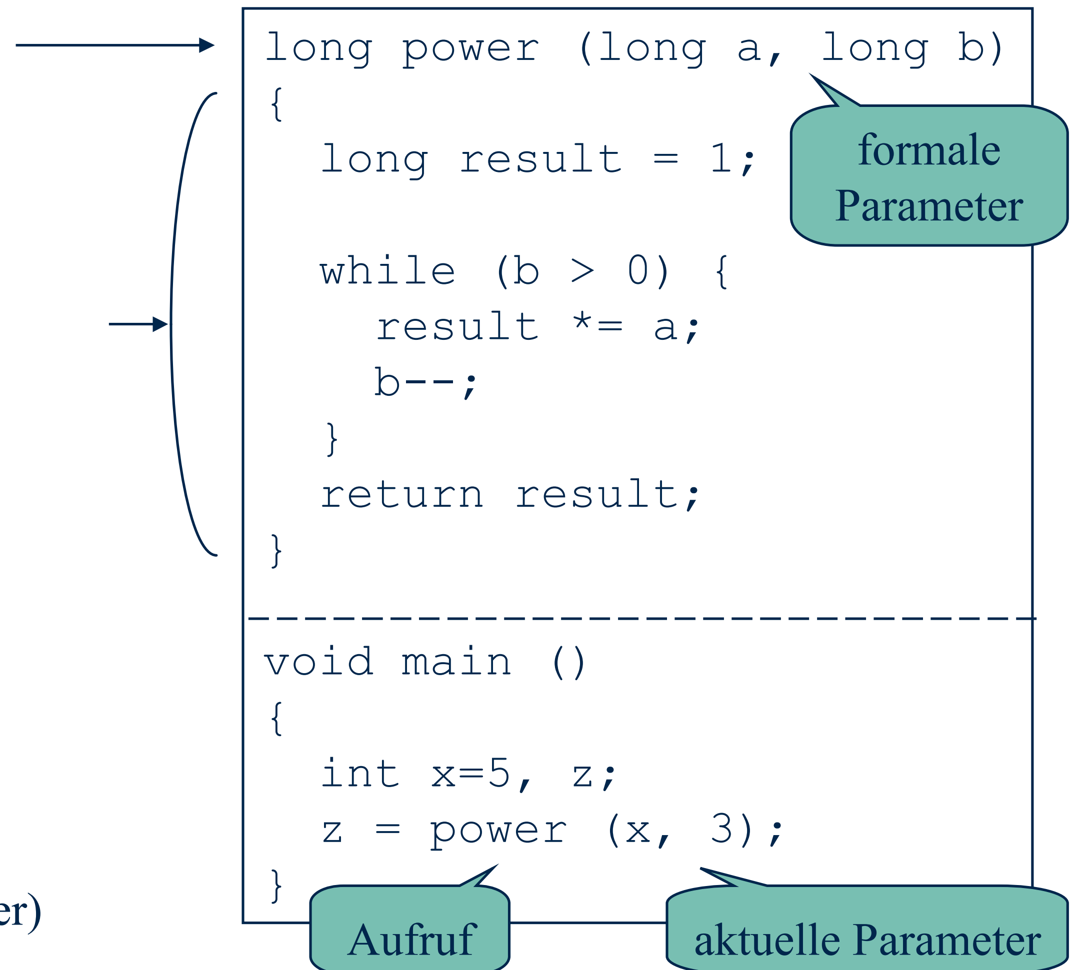
    read_file (datei, puffer); ☹

    "Eigentliche" Aufgabe ist Zugriff auf eine Datei
    - Ergebniswert ist "nur" die Erfolgs-/Mißerfolgsmeldung

Deklaration und Definition allgemeiner Fall

- Deklaration des Prototypen
  - Ergebnistyp
  - Name
  - Formale Parameter mit Typen
- Definition des Funktionskörpers
  - lokale Variablen
  - Anweisungsliste
  - ggfs. return Anweisung
- nur auf Modulebene, nicht geschachtelt

- Prototyp ohne Funktionskörper:
  long power (long a, long b);
  - Vorwärtsreferenzen
  - externe Funktionen (kommt später)



Sonderfälle

- keine Parameter
  - anstelle der leeren Parameterliste darf man auch `void` einsetzen

```
double pi ()
{
  return 3.1415927;
}
```

- kein Ergebnis ("Prozedur")
  - `void` als Ergebnistyp
  - `return` Anweisung entfällt oder ist ohne Wert

```
void GibAus (int i)
{
  cout << setw(5) << i;
}
```

- weder Ergebnis noch Parameter

```
void playCD ()
{
  // z.B. Zugriff auf Hardware
}
```

Aufruf Normalfall: "call by value"

- Übergabe der aktuellen Parameter
  - formale Parameter repräsentieren lokale Variable
  - diese werden beim Aufruf mit den aktuellen Parameterwerten initialisiert
- Ausführung des Funktionskörpers
  - endet mit return oder }
- Substitution des Aufrufs durch seinen Ergebniswert
  - aus return Ausdruck entnommen

```
long power  (a, b)
{
   long a=a, b=b;
   // ...
   return result;
}
```

```
long power(long a,long b);

int  x, z; long y;
z = power (x, y);    /* ≡ */
z = (int)power((long)x,y);
```

automatisch

- ggfs. automatische Typkonversionen
  - aktuelle Parameter ➔ formale Parameter
  - return Ausdruck ➔ Ergebnistyp ➔ Umgebungtyp

Referenzparameter "call by reference"

- Anwendungen
  - Rückgabe mehrerer Ergebnisse über Parameter
  - spart Kopieraufwand bei größeren Objekte;
    Schreibzugriffe vermeiden !
- Referenz ist "zweiter Name" für bereits existierende Variable
  - "Alias" bezeichnet denselben Speicher
  - akt. Param. muß L-Wert sein
- & modifiziert Typdeklaration
  - auf beliebige Typen und Klassen anwendbar

```
void swap (int &a, int &b)
{
  int temp = a;
  a = b;
  b = temp;
}

int  x=3,  y=5;

swap (x, y);

cout << x << " " << y;
                 // gibt aus:  5 3
```

Schrecklicher Sonderfall: Arrays als Parameter

- Array als Ergebnis ist illegal
- Array als Parameter möglich
  - es wird nicht das Array, sondern seine Anfangsadresse übergeben
  - die Funktion arbeitet nicht mit einer lokalen Kopie, sondern mit dem Original
  - die Funktion kennt die Größe des Arrays nicht
- d.h. Arrays sind immer Referenzparameter
- Grund (Ausrede): Effizienz, Vermeidung großer Kopien

Erbe aus C

```
void
kumuliere (int v[5],
           int laenge)
{
  int i;
  // sizeof(v)   liefert 4 !!!
  for (i=1; i<laenge; i++)
    v[i] += v[i-1];
  // return v; wäre illegal
}

int main ()
{
  int x[5] = {1,2,3,4,5};
  kumuliere (x, 3);
  // x enthält jetzt 1,3,6,4,5
}
```

ReadOnly-Parameter const

- **wenn** man Arrayparameter nicht verändern will oder
- **wenn** man Referenzparameter nur aus Effizienzgründen verwendet
  - und nicht zwecks Rückgabe von Werten
- **dann** sollte man solche Parameter `const` deklarieren
  - dies ist die Zusicherung an den Compiler, daß der Funktionskörper diese Parameter nicht ändert
  - der Compiler kontrolliert das und entdeckt versehentliche Änderungen

zugleich Dokumentation

```
Sprite (const Spielwiese& WelcheWiese,
        const char* Bildname);
```

```
int Summe (const int v [10],
           int laenge)
{
  int sum = 0;
    for (int i=0; i<laenge; i++)
      sum += v[i];
  return sum;
}
```

Default-Parameter

- Zuordnung von Standardwerten zu formalen Parametern
  - Prototypdeklaration:
    ```
    void InitBildschirm (int Breite=80, int Hoehe=24);
    ```
  - mögliche Aufrufe:
    ```
    InitBildschirm (132, 25);
    InitBildschirm (132);
    InitBildschirm ();
    ```

kann auch Ausdruck sein

- die letzten Parameter dürfen fehlen
- Standardwerte im Funktionsprototyp definieren
  - dürfen nicht redefiniert werden
  - nur zusätzliche Default-Parameter in späterer Deklaration zulässig

Triviale Funktionen inline

- bei trivialen Funktionen kann der Aufwand für die Parameterübergabe größer sein als die "eigentliche Funktion"
  - trotzdem Funktion sinnvoll wegen Abstraktion, Zugriffsschutz, ...
  - gute Praxis bei Klassen: Zugriff auf Attribute nur über Funktionen
- es soll keinen Grund geben, auf Einsatz einer Funktion zu verzichten

- **inline** bewirkt unmittelbare Substitution des Körpers der Funktion an der Aufrufstelle
  - nach Ermessen des Compilers
- Einschränkungen:
  - nicht exportierbar, daher am besten in .h definieren
  - nicht rekursiv

```
class CDisplay  {
int m_Breite;
public: //=============================
  int Breite () { return m_Breite; };
};
```

vorzugsweise æ; notfalls auch ç

```
inline int CDisplay::Breite ()
{                // reine Abfragefunktion
  return m_Breite;
}
```

Seiteneffekte grundsätzlich vermeiden

und globale Variable auch ...

- Seiteneffekt ist das
  Schreiben in globale Variable
- gefährlich:
  - Wirkung der Funktion spiegelt sich nicht nur im Ergebniswert wider
  - nicht an der Aufrufstelle erkennbar
- Auswertereihenfolge in Ausdrücken und Parameterlisten ist undefiniert !
  diff = pop() - pop();
  (nicht verwechseln mit Operator-Präzedenzen)
- "Funktionale Programmierung" begegnet dem SW-Chaos durch Verbot von Seiteneffekten

Eine Klasse Stack ist hier die bessere Lösung

```
int stack[10];
int index=0;

void push (int wert)
{
  stack [index] = wert;
  index ++;
}

int pop (void)
{
  index--;
  return stack [index];
}
```

```
push (wert);
ergebnis = pop();
```

Rekursion

```
Direkte Rekursion:

int fakultaet (int x)
{
  if (x<=1)
    return 1;
  else
    return x*fakultaet (x-1);
}

ergebnis = fakultaet (wert);
```

- zur Abarbeitung rekursiv definierter Datenstrukturen oder Grammatiken
- beansprucht den Stack stark
- jede Aufgabe ist auch iterativ lösbar

```
Indirekte Rekursion:

void proc1 (int i1);

void proc2 (int i2);
{ ...
  proc1 (...);
  ...
}

void proc1 (int i1);
{ ...
  proc2 (...);
  ...
}
```

Vergleich Iteration / Rekursion

aus Mathebuch: für x>0 gilt:
GGT(x,0) = x
GGT(x,y) = GGT(y,x%y) für y>0

```
int GGT (int x, int y)
{
  while (y > 0) {
    int temp = x % y;
    x = y;
    y = temp;
  }
  return x;
}
```

```
int GGT (int x, int y)
{
  if (!(x>0 && y>=0))
    throw "Param. fehlerhaft";
  if (y==0)
    return x;
  else
    return GGT (y, x % y);
}
```

Zahlenbeispiel GGT (15, 123):

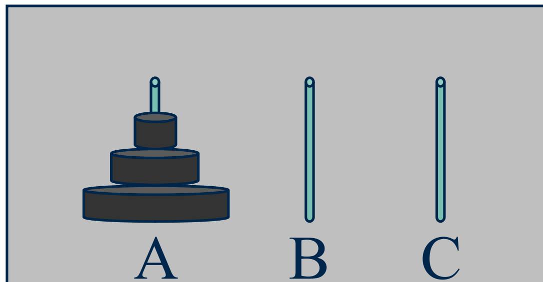
x	y	Ergebnis
15	123	GGT (123, 15)
123	15	GGT (15, 3)
15	3	GGT (3, 0)
3	0	3

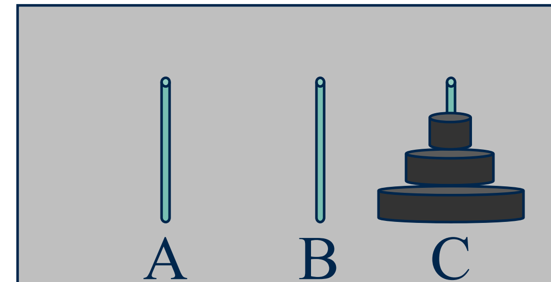
GGT berechnet den Größten Gemein-samen Teiler zweier natürlicher Zahlen

PRE

entspricht den Zwischenergebnissen nach jedem Schleifendurchlauf

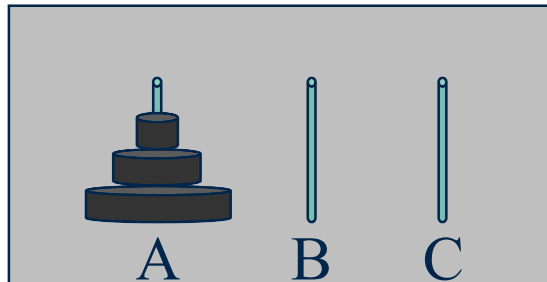
Die Türme von Hanoi

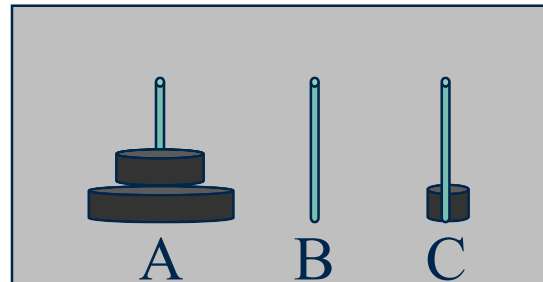


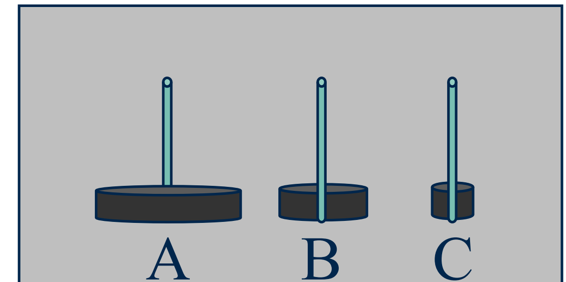


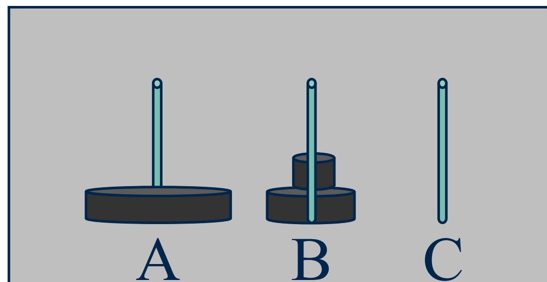
- Ziel des Spiels: Bewege die n – Scheiben des Turms von A nach C
- Bedingung:
  - Eine kleine Scheibe darf nie unter einer grossen Scheibe liegen

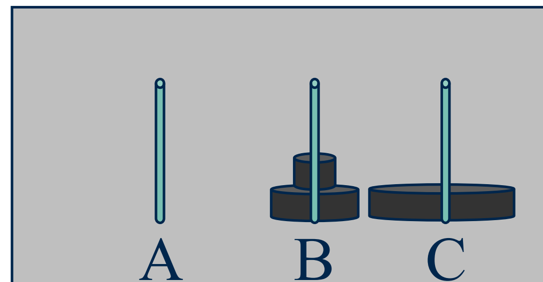
Die Türme von Hanoi – die Lösung (3 Scheiben)

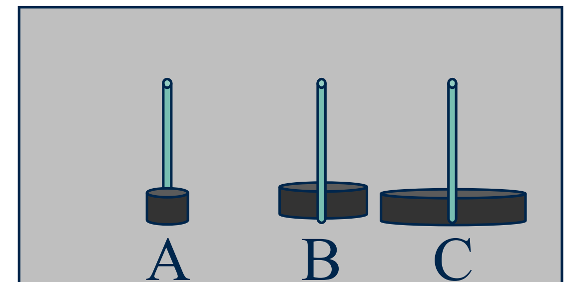


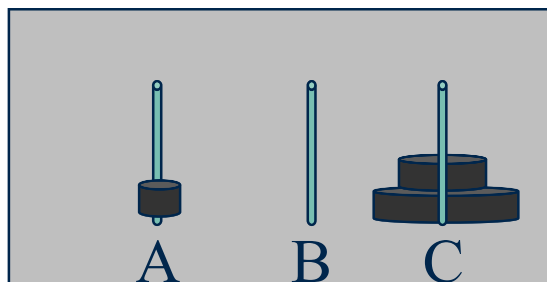


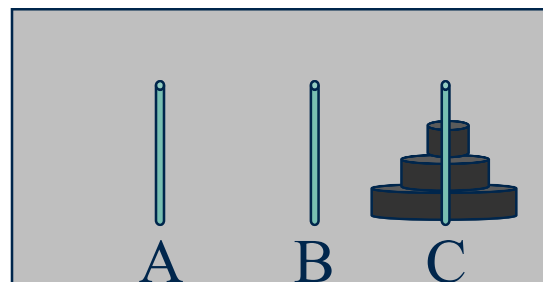






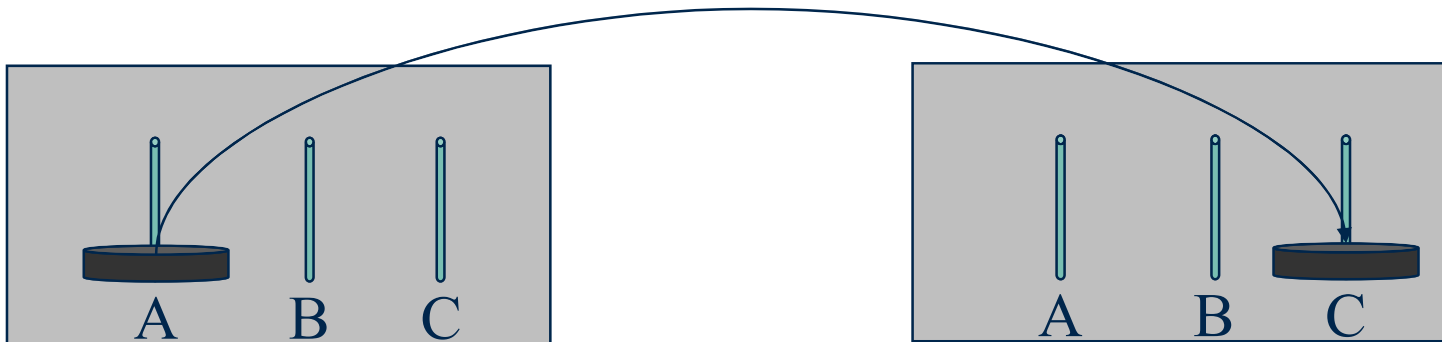






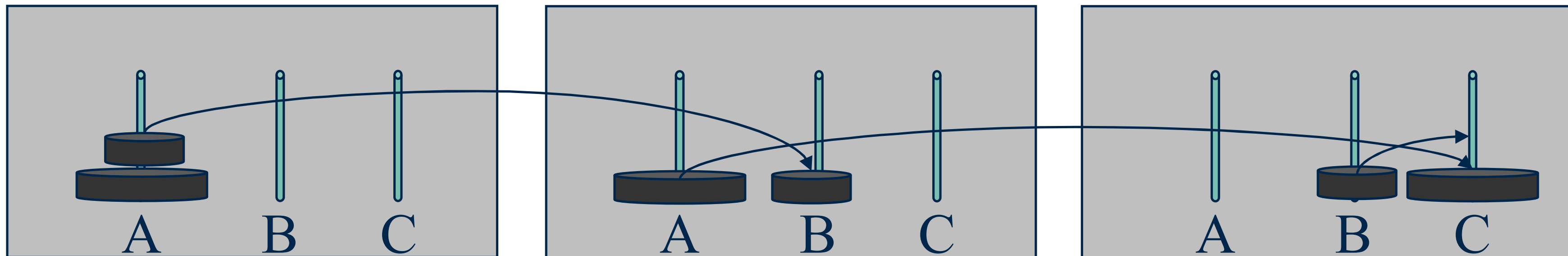
in 7 Zügen

Die triviale Lösung – ein Turm mit Höhe 1



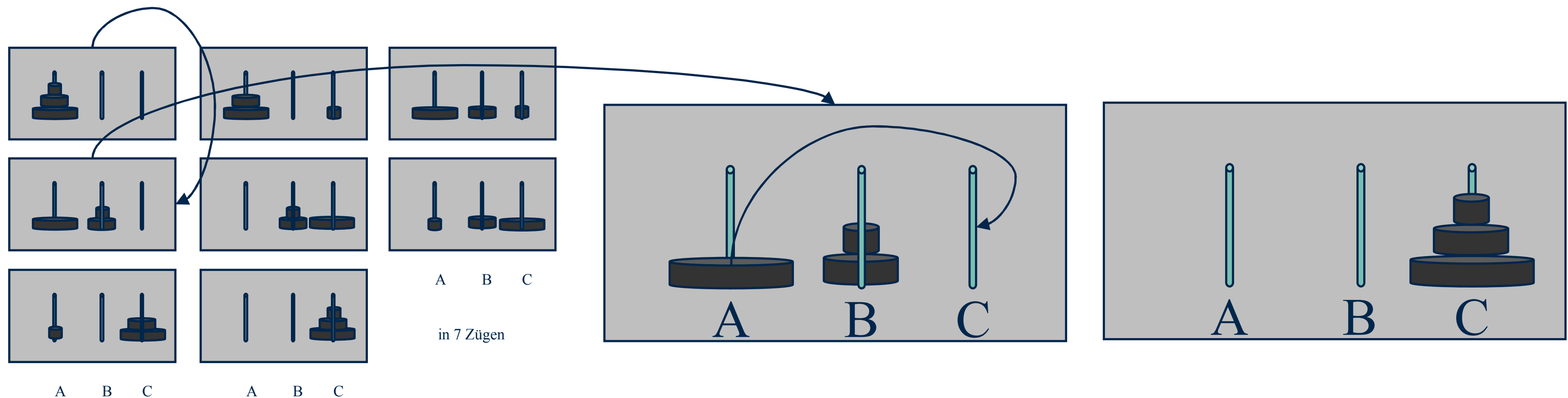
- Lösung:
  - Bewege die Scheibe von A nach C

Rekursiver Ansatz – ein Turm mit Höhe 2



1. Scheibe von A auf B
2. Trivale Lösung von A nach C
3. Dann Scheibe von B nach C

Die Türme von Hanoi - ein Turm mit Höhe 3



1. Zuerst wird der Turm mit Höhe 2 von A via C nach B gebracht. Dies entspricht Zug 3 s.o.
2. Wir haben die triviale Lösung. von A nach C.
3. Dann wird der Turm mit Höhe 2 von B via A nach C gebracht

Die Türme von Hanoi - allgemein

- Um einen Turm der Höhe N von A via B nach C zu bringen muss man:
  1. Einen Turm der Höhe N-1 von A via C nach B
  2. die triviale Lösung von A nach C
  3. Einen Turm der Höhe N-1 von B via A nach C

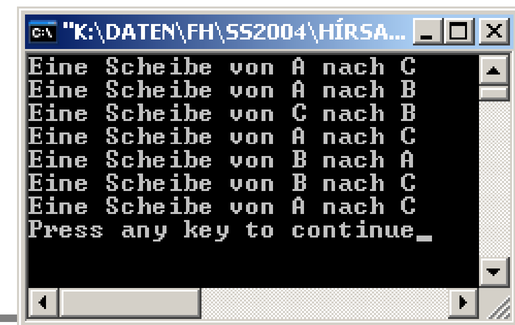
➢ D.h. Rekursiv definiert die Funktion rekhanoi

```
void rekhanoi(int hoehe, char a,
              char b, char c) {

if (hoehe == 1) {
   cout << "Eine Scheibe von "
        << a << " nach " << c;
   return; }

rekhanoi (hoehe-1, a,c,b)
rekhanoi (1,a,b,c);
rekhanoi (hoehe-1, b,a,c);

}
```



Dateistruktur eines "komplexen" Programms

bisher eine Dateien main.cpp

```
….
void swap (int &x, int &y);

void main () {
  int a=5, b = 6;
  swap (a,b);
  cout << a  << b;
}

void swap (int &x ,  int & y){
  int temp = y;
  y = x;
  x = temp;
}
```

Prototyp

Funktion

swap.h

```
void swap (int &x, int &y);
```

main.cpp

```
….
#include "swap.h"

void main () {
  int a=5, b = 6;
  swap (a,b);
  cout << a  << b;
}
```

swap.cpp

```
void swap (int &x ,  int & y){
  int temp = y;
  y = x;
  x = temp;
}
```

Wichtige Grundregeln für den Programmierstil

- Wer Funktionen erstellt muss dem "Verwender" mitteilen was die Funktion kann
  - rein technisch durch den Prototyp
    d.h. in der Header Datei
- ….und welche Voraussetzungen und Bedingungen die Funktion hat.
  - Dokumentation oder/und
  - Kommentare im Quelltext (Header und in der eigenen Funktionsdatei)

```
int fakultaet ( intx) ;
// berechnet die Fakultät von x, d.h. x!
// PRE:  x > 0   &&
//          x klein genug, damit das Ergebnis
//          noch in int passt
// POST: Ergebnis ist die Fakultaet von x
//          &&  Ergebnis >= 1
```

```
/**********************************************
**  void swap ( int &x, int &y)
**  Change the values of the two input parameters
**  Input:              int &y, int &y.
**                      Caution referenz
**  Return:             nothing
**  Precondition:       none
**  Side effects :
**  Version :           1.0
**  Author:             your name
**********************************************/
void swap (int &x, int &y);
```