Beispiel: Vom Problem zum Code

- High-Level Pseudo-Code:

```
Eingabe:
• Anzahl Scheiben: n (jede mit aufgedruckter Nummer)
• Quell-Stange, Ziel-Stange, Zwischenspeicher-Stange

Vorbedingungen:
• Auf Quell-Stange befinden sich n (oder mehr) Scheiben in
      aufsteigender Sortierung
• Auf der Zwischenspeicher- und der Ziel-Stange befinden
      sich nur Scheiben mit größerem Radius (oder gar keine)

1. verschiebe obere n-1 Scheiben von Quell-Stange auf
      Zwischenspeicher-Stange
2. bewege Scheibe mit Nummer n (jetzt oben liegend) von
      Quell-Stange nach Ziel-Stange
3. verschiebe obere n-1 Scheiben von Zwischenspeicher-Stange
      auf Ziel-Stange
```


Beispiel: Vom Problem zum Code

- Verfeinerter Pseudo-Code:

```
Funktion hanoi( n, q, z, t )

Parameter
• Integer n = Anzahl Scheiben, die von Quell-Stange nach
  Ziel-Stange bewegt werden sollen (evtl. sind auf der Quell-
  Stange noch mehr Scheiben)
• Zeichen q = Quell-Stange,
  Zeichen z = Ziel-Stange,
  Zeichen t = Zwischenspeicher-Stange

falls n>1: verschiebe mittels hanoi( n-1, q, t, z )
bewege Scheibe mit Nummer n von Stange q nach Stange z
falls n>1: verschiebe mittels hanoi( n-1, t, z, q )
```


Beispiel: Vom Problem zum Code

- C++ Code:

```
// n = Anzahl Scheiben, die von Quell-Stange nach Ziel-Stange
//     bewegt werden sollen (evtl. sind auf der Quell-Stange
//     noch mehr Scheiben)
// q = Quell-Stange,
// z = Ziel-Stange,
// t = Zwischenspeicher-Stange

void hanoi( unsigned int n, char q, char z, char t )
{
    if ( n > 1 )
        hanoi( n-1, q, t, z );
    printf( "bewege Scheibe %d von Stange %c nach Stange %c",
            n, q, z );
    if ( n > 1 )
        hanoi( n-1, t, z, q );
}
```


Rekursion vs. Iteration

Rekursion	Iteration
Implizite Wiederholung durch Funktionsaufruf	Explizite Wiederholung durch Schleifenkonstrukt
Abbruch wenn Basisfall erreicht	Abbruch wenn Schleifenbedingung false

- Unendliche Schleifen bei beiden möglich
- Jede Rekursion kann man umformen in eine äquivalente Iteration (auf jeden Fall immer durch Verwaltung eines eigenen Stacks)

Tendentiell eleganter	Selten performanter

Türme von Hanoi - Iterative Lösung

- Bei den Türmen von Hanoi gibt es eine wenig bekannte einfache iterative Lösung, die von P. Buneman und L. Levy schon im Jahre 1980 gefunden wurde
- Pseudo-Code:
  - impliziert eine kreisförmige Anordnung der Stäbe

```
while(1)
{
  Bewege die kleinste Scheibe von ihrem aktuellen
       Standort auf den nächsten Stab in Uhrzeigerrichtung;
  if (alle Scheiben sind auf einem Stab)
  {
    break;
  }
  Führe den einzig möglichen Zug aus, der nicht
       die kleinste Scheibe bewegt;
}
```


Tail Recursion

- Rekursive Funktion, wo rekursiver Aufruf der *letzte* Befehl ist.
- Beispiel:

```
unsigned int add( unsigned int a, unsigned int b )
{
    if ( b == 0 )
        return a;
    else
        return add(a + 1, b - 1);
}
```

- Umformung in Schleife ist trivial
- Wird vom Compiler erkannt und automatisch als Schleife implementiert (falls er was taugt)

Beispiel Tail Recursion

- Beispiel: Berechne $n!$ rekursiv (für $n \geq 1$):

$$n! = \prod_{j=1}^{n} j$$

```
unsigned int factorial( unsigned int n )
{
    if ( n == 1 )
        return 1;
    else
        return n * factorial(n - 1);
}
```

- Äquivalente *Tail Recursion*:

```
unsigned int factorial_tail( unsigned int i,
                             unsigned int fac )
{
    if ( i == 1 )
        return fac;
    else
        return factorial_tail(i-1, fac * (i-1));
}
unsigned int factorial( unsigned int n )
{
    return factorial_tail(n, n);
}
```

Vorbedingung: $fac = \prod_{j=1}^{n} j$

denn $fac \cdot (i-1) = \prod_{j=1}^{n} j$

Beispiel Tail Recursion

- Wdh.: Tail Recursion:

```
unsigned int factorial_tail( unsigned int i,
                             unsigned int fac )
{
    if ( i == 1 )
        return fac;
    else
        return factorial_tail(i-1, fac * (i-1));
}
unsigned int factorial( unsigned int n )
{
    return factorial_tail(n, n);
}
```

- Äquivalente Schleife:

```
unsigned int i = n;
unsigned int fac = n;
while ( i != 1 )
{
    fac = fac * (i-1);
    i = i-1;
}
return fac;
```


Lifetime, Scope und Linkage

- Variablen haben noch mehr Attribute:
  - Bekannt: Typ, Name, Speicheradresse, Wert
- *Lifetime* : Lebensdauer, d.h., wie lange Variable im Speicher existiert
- *Scope* : wo die Variable im Programm bekannt ist (referenziert werden kann)
- *Linkage* : in welchen anderen Files die Variable verwendet werden kann (bei Programmen, die aus vielen Files bestehen)

Lifetime

- Lifetime auf einen Block begrenzt (Storage "Automatic"):
  - gilt per Default für lokale Variablen
  - Variable wird erzeugt bei Eintritt in Block (z.B. Fkt-Body)
  - Variable wird zerstört bei Austritt
- unbegrenzte Lifetime (Storage "Static"):
  - gilt für alle globalen Variablen
  - für lokale Variablen: mit Keyword **`static`**
  - Variable existiert und behält Wert für die gesamte Laufzeit des Programms

Scope

- File-Scope:
  - Globale Variablen
  - Sichtbar ab Deklaration bis File-Ende, auch innerhalb allen Funktionen
- Function-Scope:
  - Lokale Variablen und formale Parameter
  - Sichtbar ab Deklaration bis Ende der Funktion
- Block-Scope:
  - Dito für Blocks innerhalb Funktion

Orthogonalität von Lifetime und Scope

- Def. *orthogonal* :
  2 Features einer Programmiersprache sind orthogonal, wenn sie "nichts miteinander zu tun haben".
  In diesem Fall kann man sie i.a. beliebig kombinieren.
- Achtung: Lifetime und Scope sind "orthogonal"
  - Variable kann **`static`** sein, und trotzdem nicht überall sichtbar!
- Beispiel:

```
unsigned int foo( void )
{
    static unsigned int n = 0;
    n ++ ;
    return n;
}
```

Beispiel

```
#include <stdlib.h>
int globale_var;
int foo( int x )
{
    static bool first_time = true;
    int y = 25;
    if ( first_time )
    {
        first_time = false;
        return y;
    }
    else
    {
        int z = globale_var * 2;
        return x-y-z;
    }
}
int main( void )
{
    globale_var= 17;
   foo(0);
}
```

globale Variable

Scope von `globale_var`

Beispiel

```
#include <stdlib.h>
int globale_var;
int foo( int x )
{
    static bool first_time = true;
    int y = 25;
    if ( first_time )
    {
        first_time = false;
        return y;
    }
    else
    {
        int z = globale_var * 2;
        return x-y-z;
    }
}
int main( void )
{
    globale_var= 17;
   foo(0);
}
```

Wird beim Start des Progr. erzeugt und mit Wert belegt

Scope von `first_time`

Beispiel

```
#include <stdlib.h>
int globale_var;
int foo( int x )
{
    static bool first_time = true;
    int y = 25;
    if ( first_time )
    {
        first_time = false;
        return y;
    }
    else
    {
        int z = globale_var * 2;
        return x-y-z;
    }
}
int main( void )
{
    globale_var= 17;
   foo(0);
}
```

Wird jedesmal beim Eintritt in `foo` erzeugt und mit 25 belegt

Scope von `y`

Beispiel

```
#include <stdlib.h>
int globale_var;
int foo( int x )
{
    static bool first_time = true;
    int y = 25;
    if ( first_time )
    {
        first_time = false;
        return y;
    }
    else
    {
        int z = globale_var * 2;
        return x-y-z;
    }
}
int main( void )
{
    globale_var= 17;
   foo(0);
}
```

Wird nur beim Eintritt in diesen Block erzeugt

Scope von `z`

Verdeckung

- Eine Variable in einem *inneren* Scope *verdeckt* eine andere gleichen Namens in einem *äußeren* Scope
- Bsp.:

```
int i;                                  // file (global) scope
int func()
{
    int i = 50;                         // function scope, hides
                                        // i at global scope
    for ( int i = 0; i < 100; i ++ )    // block scope. Hides
                                        // i at function scope
    {
        int i;                          // this is an error...
        ...
    }
}
```

- Ganz schlechte Programmierpraxis!
- Verwende **`-Wshadow`**

Default-Argumente

- Formale Parameter mit Default-Belegung
- Bsp.:
  `int foo( int a, int x = 0, float y = Pi )`
- Aufruf:
  - Alle "normalen" Parameter müssen vorhanden sein
  - Default-Parameter dürfen fehlen (Compiler setzt Defaults ein)
  - Falls einer fehlt müssen alle rechts davon auch fehlen
- Bsp.: **`foo( i, j );`**
- Defaults können sein:
  - Konstanten
  - Globale Variablen
  - Funktionsaufrufe

Function Overloading

- Erste Begegnung mit Polymorphie (zur Compile-Zeit)
- Mehrere Funktionen gleichen Namens aber verschiedenen Prototyps
- Aber: Funktionen können sich *nicht nur* im Typ des Rückgabewertes unterscheiden

- Beispiel:

```
int    square( int x )      { return x * x; }
float  square( float x )    { return x * x; }
```

- Was macht der Compiler?
  - Heuristik zur Entscheidung, welche Fkt verwendet wird bei Call
  - Achtung: Heuristik ist komplex!
    - Type conversions, Default-Argumente, ...

Heuristik zum Matchen

- Vereinfacht:
  1. Suche exakte Korrespondenz
  2. Standardmäßige Argument-Konvertierung gemäß Promotion-Hierarchie (versuche zunächst int → long, dann int → float)
  3. User-definierte Typ-Konvertierung (my_type → int)
  4. Fehlermeldung
- Programmierer kann mit Cast immer einen bestimmten Match erzwingen

Funktionen richtig dokumentieren

```
/**  Einzeilige Beschreibung
 *
 * @param param1      Beschreibung von param1
 * @param param2      Beschreibung von param2
 *
 * @return
 *   Beschreibung des Return-Wertes.
 *
 * Detaillierte Beschreibung ...
 *
 * @warning
 *   Dinge, die der Aufrufer unbedingt beachten muss...
 *
 * @pre
 *   Annahmnen, die die Funktion macht...
 *
 * @todo
 *   Was noch getan werden muss
 *
 * @bug
 *   Bekannte Bugs dieser Funktion
 *
 * @see
 *   ...
 *
**/
```


Spezielle Funktionen

- **`main`**:
  - Sagt dem Betriebssystem, wo Einsprung ins Programm

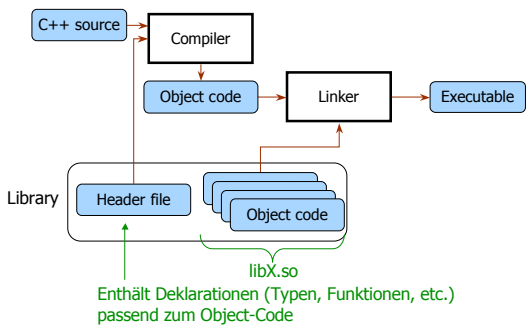
```
int main( int argc, const char *argv[] )
{
    ...
    return x;
}
```

- Exkurs: umgekehrte Fkt dazu ist **`exit`**
  - Bekommt man durch **`#include <stdlib.h>`**
  - Prototyp ist **`void exit( int );`**
  - Parameter ist Rückgabewert für Vater-Prozeß (0 = ok)

Libraries

- Erfinde das Rad nicht neu!
- *Library* :=
  - Menge vor-compilierter Sources (Funktionen, Klassen, Typen, ...)
  - Zusammengefaßt in einen File (Unix: libX.so, Windows: libX.dll)
  - Kein lauffähiges Programm (enthält kein **`main`**)
- *System-Library* :=
  Library, die in bestimmten, vordefinierten Verzeichnissen installiert ist
  - Kommt typischerweise mit dem Betriebsystem
  - Unix: /usr/lib ; Windows: C:\WINDOWS\SYSTEM32
- Ein guter Programmierer kennt viele und die richtigen Libraries ...

Verwendung von Libraries beim Build-Prozeß



- Wie verwendet man System-Libraries ?
  - Im C++ File: Header-File includen
    - Bsp.: `#include <math.h>`
  - Beim Compilieren des Exe's: Lib linken
    - Konvention unter Unix: `-lX` ilnkt Library `libX.so`
    - Beispiel Math-Library linken:

```
g++ -o myprog file1.o file2.o -lm
```

- *Standard-Library* :=
  spezielle System-Library, die zu jedem C-Programm automatisch gelinkt wird

Funktionen der Math-Library

Method	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is 10.0 `ceil( -9.8 )` is -9.0
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is 1.0
`exp( x )`	exponential function *ex*	`exp( 1.0 )` is 2.71828
`fabs( x )`	absolute value of *x*	`fabs( -8.76 )` is 8.76
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is 9.0 `floor( -9.8 )` is -10.0
`fmod( x, y )`	remainder of *x*/*y* as a floating -point number	`fmod( 13.657, 2.333 )` is 1.992
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is 1.0 `log( 7.389056 )` is 2.0
`log10( x )`	logarithm of *x* (base 10)	`log10( 100.0 )` is 2.0
`pow( x, y )`	*x* raised to power *y* (*xy*)	`pow( 2, 7 )` is 128 `pow( 9, .5 )` is 3
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is 0
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is 30.0
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is 0

Fig. 3.2 Math library functions.