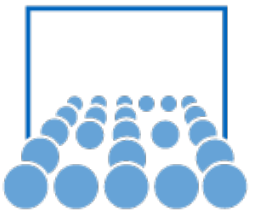
Kompaktkurs
Einführung in die Programmierung

5. Funktionen

Stefan Zimmer

5.3.2010



Warum Funktionen?

- Idee: Programmcode, der an mehreren Stellen des Programms verwendet wird, an einer Stelle aufschreiben (*Funktionsdefinition*); Verwendung durch *Funktionsaufrufe*
- Vorteil gegenüber Copy&Paste:
  - Weniger Schreibarbeit (dank unseres schönen Texteditors ist dieser Vorteil kleiner als man zunächst denkt)
  - Mechanismen zur Parametrisierung des Codes: wir wollen manchmal `sin(x)`, manchmal aber auch `sin(3*y)` berechnen – müssen wir mit „Suchen und Ersetzen“ aus `x` überall `3*y` machen?
  - Wesentlicher Vorteil ist aber, dass man das Programm wesentlich lesbarer (=richtiger!) machen kann

Funktionen wegen Lesbarkeit

- Das Innenleben einer Funktion ist gegenüber außen gut versteckt: wir spezifizieren möglichst nur, *was* sie tut, nicht *wie* sie das tut
- Das ist ein Riesenvorteil, um das große Problem in kleine Happen zu zerteilen: weil nun viel weniger Information relevant ist, fällt der Zusammenbau zur nächstkomplexeren Struktur leichter (naja, zumindest dann, wenn die Funktionen geeignet entworfen worden sind)
- Also lohnt es sich, die Mechanismen von Funktionsdefinition und -aufruf zu studieren
- Das ist aber nur der technische Aspekt der Zerlegung – der inhaltliche Aspekt (wie wähle ich die Teilaufgaben auf), ist nicht mechanisch zu erledigen – hier ist Kreativität (und Übung!) gefragt

Sprachgebrauch

- Der Mechanismus, Programmteile zu verpacken, läuft unter verschiedenen Namen – wir sagen „Funktion“, lesen aber auch mal „Prozedur“ oder vielleicht „Methode“
- Allgemein ist der Sprachgebrauch und die Abgrenzung, falls man mehrere dieser Begriffe verwendet, von Sprache zu Sprache (und manchmal auch von Buch zu Buch) unterschiedlich – an so was müssen wir uns halt gewöhnen
- Interessant ist der Vergleich mit dem Funktionsbegriff aus der Mathematik: der Idealfall einer C-Funktion kommt einer mathematischen Funktion sehr nahe – sie beschreibt eine eindeutige Abbildung der Eingabewerte auf das Funktionsergebnis. Oft geht's aber schmutziger zu (Folie „Seiteneffekte“ weiter hinten)

Funktionen: Definition

- Eine *Funktionsdefinition* sieht so aus:
  Ergebnistyp Name **(***Parameterliste***)** *Rumpf*
- *Ergebnistyp*: der Datentyp des Funktionsergebnisses (z.B. `int` oder `double`). Ausnahme: wenn die Funktion kein Ergebnis hat, steht hier **`void`**
- *Parameterliste*: eine Liste der *Formalparameter* in der Form *Datentyp Parametername*
  Durch Kommas trennen (immer nur ein Parametername):
  `int a, int b, double xyz3`
  Die Liste kann leer sein, die Klammern gehören aber auch dann dazu (man darf dann auch **`(void)`** schreiben)
- *Rumpf*: ein Block – durch geschweifte Klammern eingeschlossene Folge von Variablendefinitionen und Anweisungen (C: in dieser Reihenfolge, C++: auch gemischt)

Funktionen: Beispiele

- Zwei Beispiele: die erste Funktion kennen wir schon, die zweite ist ein Beispiel für eine „unkommunikative“ Funktion, die weder Parameter hat, noch ein Ergebnis berechnet:

```
double hoch(double x, int n) {
    double erg;
    int i;
    erg = 1.0;
    for (i=1; i<=n; i++) { erg = erg*x; }
    return erg;
}
void Muh() {
    std::cout << "Muh\n";
}
```

Funktionsaufruf

- Der Funktionsaufruf hat die Form
  Funktionsname(*Parameterliste*)
- *Parameterliste*:durch Kommas getrennte Folge von Ausdrücken, den *Aktualparametern*.
- Jeder Formalparameter wird auf den Wert des zugehörigen Aktualparameters gesetzt. Wenn's keine Parameter gibt, bleiben die Klammern halt leer.
  Hinsichtlich evtl. nötiger Typumwandlungen gelten in C dieselben Regeln wie bei Zuweisungen (in C++ kommt eine Komplikation dazu, die uns hier nicht interessiert)
- Der Rumpf wird abgearbeitet bis zu einer Anweisung
  `return` *Ausdruck* **`;`**
  Der in der `return`-Anweisung berechnete Wert ist das Funktionsergebnis, mit dem in der aufrufenden Funktion weitergearbeitet wird

Funktionsergebnis; `void`

- Ein Funktionsaufruf kann Bestandteil eines Ausdrucks sein – die Auswertung erfolgt wie eben beschrieben, der Wert dieses Teilausdrucks ist das Funktionsergebnis:
  `(hoch(3.0, 2) + hoch (4.0, 2)) / 5.0`
- Ausnahme: Aufrufe von Funktionen, die als `void` deklariert wurden, dürfen nicht Bestandteile größerer Ausdrücke sein – sie treten nur in der Form auf wie z.B. `Muh();` (für „echte“ Funktionen ist diese Form des Aufrufs natürlich auch OK)
- In diesem Fall entfällt auch der Ausdruck in der `return`-Anweisung (oder auch die ganze Anweisung: wenn die Anweisungen im Block alle abgearbeitet worden sind, ist sowieso Schluss)
- Streng genommen sind die Varianten aus dem vorigen Punkt auch für Funktionen mit Ergebnis erlaubt, allerdings ist dieses Ergebnis dann ein undefinierter Wert

Seiteneffekte

- Funktionen ohne Ergebnis können durchaus sinnvoll sein, weil Funktionen auch Anweisungen enthalten können, die sich auf mehr als auf die Berechnung eines Ergebnisses auswirken: auch Funktionen können *Seiteneffekte* haben
- Bisher kennen wir als Beispiel nur die Ausgabe auf den Bildschirm – Beispiele mit schwereren Konsequenzen werden mit der Zeit noch kommen
- Eine als `void` deklarierte Funktion wird in der Regel Seiteneffekte haben, andere dürfen das aber auch – dabei sollte man aber immer ein wenig schlechtes Gewissen haben, weil der Leser unseres Programms damit kaum rechnet (wir entfernen uns vom Ideal der Funktionen im mathematischen Sinn)

Ein beliebter Fehler

Folgendes wird gerne verwechselt:

- Die Rückgabe eines Funktionsergebnisses

```
int vier() { return 4; }
void viertest() {
    int a = vier();
    // ...
}
```

- Und die Ausgabe auf den Bildschirm (ein Seiteneffekt):

```
void fuenf() { std::cout << 5; }
void fuenftest() {
    fuenf();
}
```

- Einen wesentlichen Unterschied sieht man gleich: mit dem Funktionsergebnis kann ich weiterrechnen, was mit der Bildschirmausgabe natürlich nicht geht.

Automatische Variablen und Blöcke (1)

- Im Folgenden werden wir uns im Detail ansehen, was bei einem Funktionsaufruf passiert – insbesondere, was den Speicherplatz angeht.
- Eine wesentliche Rolle spielen dabei die Blöcke; es gilt: für die in einem Block vereinbarten Variablen wird
  - eine passende Menge Speicherplatz reserviert, wenn wir den Block *betreten* („bei Passieren des `{`“)
  - und wieder freigegeben, wenn wir den Block wieder *verlassen* („bei Passieren des `}`“, ggf. aber auch mittels eines `return`, `break` oder `continue`)
- Das ist sehr effizient zu realisieren, wobei eine simple Tatsache sehr nützlich ist: der zuletzt betretene Block wird als erstes wieder verlassen. So eine „first in, last out“-Struktur nennt man *Kellerspeicher* (Stack)

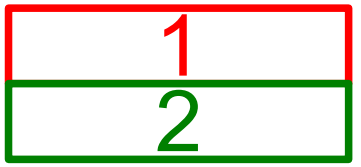
Automatische Variablen und Blöcke (2)

- Ein (sinnfreies) Beispiel und wie es im Speicher an den markierten Punkten der Ausführung aussieht:

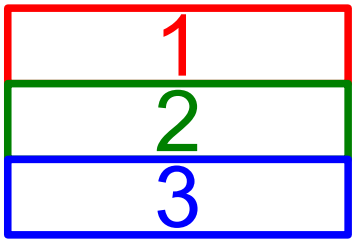
```
{   int x1 = 1;          // A
    { int x2 = 2;        // B
      { int x3 = 3;      // C
      }                  // D
      { int x4 = 4;      // E
      }
    }                    // F
}
```

A:

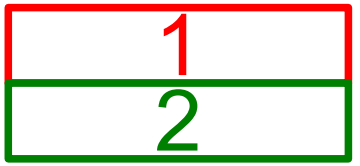
B:



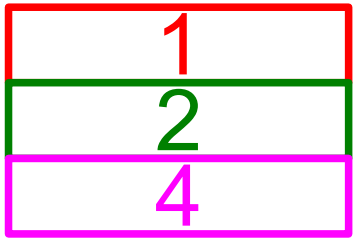
C:



D:



E:



F:

Automatische Variablen und Blöcke (3)

- Da man einen Block auch betreten kann, indem man eine Funktion aufruft, lässt sich die Kellerstruktur im Allgemeinen nicht so leicht aus dem Quelltext ablesen.
- In folgendem Programm passiert bei einem Aufruf der Funktion **`A()`** in unserem Keller praktisch dasselbe wie gerade eben:

```
void E() { int x4 = 4; }
void C() { int x3 = 3; }
void B() { int x2 = 2; C(); E(); }
void A() { int x1 = 1; B(); }
```

- Variablen, wie wir sie bisher kennen gelernt haben, entstehen und verschwinden also genau so, wie man das braucht. Deshalb heißen sie auch *automatische Variablen* (es gibt noch andere Sorten, aber erst später)

Lebensdauer, Gültigkeitsbereich

- Die *Lebensdauer* einer Variablen ist der Zeitabschnitt der Programmausführung, zu dem Speicherplatz für sie reserviert ist: bei einer automatischen Variable also vom Betreten bis zum Verlassen des Blocks, in dem sie definiert ist
- Davon zu trennen ist der *Gültigkeitsbereich* (*Scope*) eines Bezeichners (Variablennamens): der Abschnitt des Quelltextes, innerhalb dessen der Bezeichner zu einer bestimmten Deklaration gehört („an sie gebunden ist“). Im Falle automatischer Variablen ist der Bereich von der Deklaration bis zum Ende des Blocks
- (Die Benennungen sind hier übrigens mal wieder nicht einheitlich – also nicht wundern, wenn man in einem Buch andere Definitionen sieht)

Sichtbarkeit von Bezeichnern

- Ein (insbesondere hier) weniger wichtiger Sachverhalt, der sich aber sonst nirgends gut unterbringen lässt, entsteht dadurch, dass in einem Block Bezeichner vergeben darf, die außerhalb schon mal vergeben worden sind – solange die Definitionen zu verschiedenen Blöcken gehören, ist das kein Fehler
- In diesem Fall „gilt“ die innerste Definition:

```
{   int i = 1;
    { int i = 2; // OK, inneres i zählt
    }            // Äußeres i (1) zählt
    int i = 3;   // Fehler!
}
```

- Der äußere Bezeichner ist weiterhin gültig, aber nicht sichtbar („verschattet“)

Funktionsaufruf im Detail

- Bevor wir uns nun einen Funktionsaufruf im Detail anschauen können, nur noch eine Regel: die Formalparameter einer Funktion werden behandelt wie automatische Variablen des Funktionsrumpfes
  - Lebensdauer: vom Funktionsaufruf bis „fertig“
  - Gültigkeitsbereich: der Funktionsrumpf
  - Sparsame Leute verwenden die Formalparameter als Variablen, das ist erlaubt
  - Auch wenn der Aktualparameter ein Lvalue (z.B. ein Variablenname) ist, wird dem Formalparameter eine Kopie dessen Objektes zugewiesen, nicht der Lvalue selbst (klärendes Beispiel kommt noch)
  - Diese Art der Parameterübergabe durch Kopieren des Wertes heißt „*Werteparameter*“ (call by value)

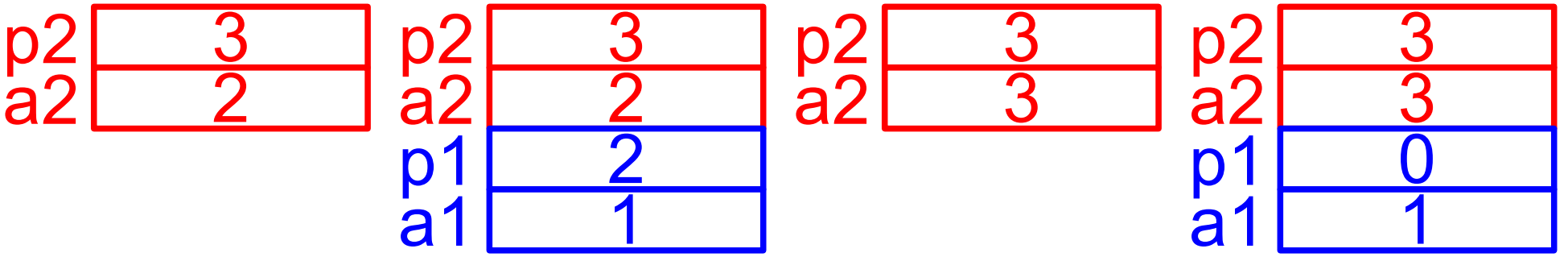
Funktionsaufruf: Beispiel

- Ein (nach wie vor sinnfreies) Beispiel mit Parametern und automatischen Variablen:

```
int f1(int p1) {
   int a1 = 1;
   return a1+p1;
}
```

```
int f2(int p2) {
   int a2 = 2;
   a2 = f1(a2);
   return f1(a2-p2);
}
```

- Einige Momentaufnahmen des Kellerspeichers bei der Auswertung von `f2(3)` (Bezeichner nur für uns zur Orientierung, der Rechner sieht sie nicht)



- (Ergebnisse: f1(2) = 3, f1(0) = 1, also f2(3) = f1(0) =1)

Funktionen und der Rechenknecht (1)

- Der Mechanismus bei Funktionsaufruf macht erfahrungsgemäß vielen Schwierigkeit, daher strapazieren wir nochmal unseren Rechenknecht
- Der sitzt da und ist mit der Auswertung einer Formel beschäftigt, als er feststellt, dass die einen Funktionsaufruf enthält:

1) Als erstes berechnet er die Werte der Aktualparameter
2) Dann holt er eine neue Kopie der entsprechenden Dienstanweisung und legt sie auf die, die er gerade bearbeitet hat, drauf (er notiert vorher noch die genaue Stelle, wo die Bearbeitung unterbrochen wurde)

Funktionen und der Rechenknecht (2)

3) Der (neuen) Dienstanweisung entnimmt er, wie viel Speicher er für die automatischen Variablen (incl. Formalparameter) der Funktion braucht
   Wir nehmen vereinfachend an, dass die Funktion keine Blöcke mit weiteren Variablendefinitionen enthält, alle Variablen seien also am Anfang der Funktion notiert (wie man das auf den allgemeinen Fall erweitert: selber überlegen!)

4) Also reserviert er im Regal entsprechend viele Fächer und beschriftet sie mit den Variablennamen

5) In die Fächer der Formalparameter legt er die in 1) berechneten Werte der Aktualparameter

Funktionen und der Rechenknecht (3)

6) Nun führt er die Dienstanweisung aus, bis er auf ein „Return“ (wie auch immer er dazu sagt) trifft

7) Er wertet den zugehörigen Ausdruck aus, merkt sich das Ergebnis, ...

8) ... wirft die (obenliegende) Dienstanweisung weg, ...

9) ... entfernt die Namensschilder aus 4) vom Regal, ...

10) ... und setzt den eben berechneten Wert an der Stelle ein, die er bei 2) notiert hatte (liegt ja wieder oben), um mit diesem Wert die Formel weiter auszuwerten.

- Beachtenswert ist, dass in den vorigen Punkten oft wiederum ein Funktionsaufruf zu bearbeiten ist – u.U. hat der Rechenknecht dann einen dicken Stapel von Dienstanweisungen vor sich (Beispiele folgen)

Zuweisungen an Parameter?

- Die folgende Funktion soll den ++-Operator nachbilden, also ihr Argument um 1 erhöhen:

```
void plusplus(int x) { x = x+1; }
void test() {
   int a = 5;
   plusplus(a);
   std::cout << a;
}
```

- Das funktioniert so nicht: zwar bekommt `x` den Wert 6, aber es ist ja eine Kopie des Wertes von `a`, das selber davon nicht betroffen ist!
- Viele Sprachen (z.B. C++) bieten daher eine andere Art der Parameterübergabe an („Variablenparameter“) - in C hingegen gilt die strikte Regel „alle Parameter sind Werteparameter“. Später lernen wir aber einen Trick, wie man trotzdem so eine Funktion schreiben kann!

Rekursion

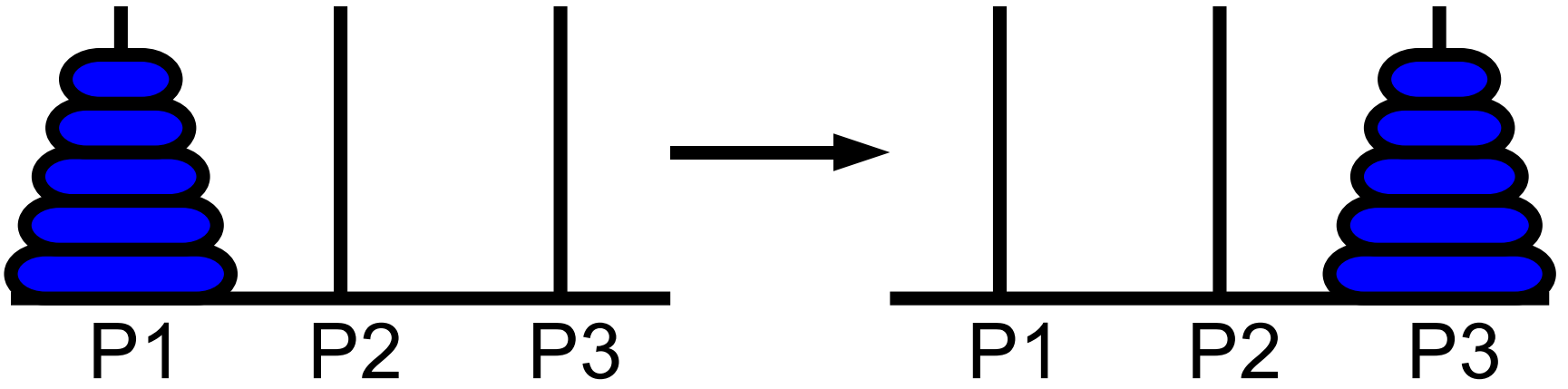
- Eine besondere Betrachtung im Zusammenhang mit der Speicherverwaltung verdienen rekursive Funktionen, etwa die beliebte Fakultätsfunktion:

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

- Ohne Rekursion wäre es möglich, einfach jeder Funktion einen festen Speicherbereich für ihre automatischen Variablen zuzuteilen
- Spätestens, wenn man aber die Auswertung von z.B. `fakultaet(3)` nachvollzieht, sieht man, dass das mit Rekursion im Allgemeinen nicht mehr möglich ist!

Beispiel: Türme von Hanoi

- Ein notorisches Beispiel für eine etwas kompliziertere Rekursion sind die *Türme von Hanoi*
- Drei Pfosten *P1*, *P2*, *P3*; auf *P1* sind *h* Scheiben von paarweise verschiedenem Durchmesser aufgespießt – der Größe nach (die größte liegt unten).
- Dieser Turm soll nun nach P3 transportiert werden
- Es darf immer nur eine Scheibe auf einmal transportiert werden
- eine Scheibe darf nur abgelegt werden entweder
  - als unterste Scheibe eines Pfostens oder
  - auf eine größere Scheibe



Türme von Hanoi: Algorithmus

- Wir teilen den Turm auf in die unterste Scheibe und den Rest.
- Den kleineren Turm auf dem freien Pfosten parken. (Rekursion!)
- Die unterste Scheibe transportieren.
- Den geparkten Turm hinterher.
- Die Rekursionsstruktur wird hier interessant, weil der Transport des Turms der Höhe *h* zwei Transporte von *(h-1)*-Türmen erfordert.
- Die Rekursion bricht ab, weil für Türme der Höhe *h=0* nichts zu tun ist.
- Die Rollen von Anfangs-, Zwischen- und Endpfosten wechseln (z.B. ist P2 das Ziel des ersten *(h-1)*-Transfers).
- Dass wir auch wirklich die Stapelregeln einhalten, kann man sich bei Gelegenheit mal überlegen.

Türme von Hanoi: Ausführung

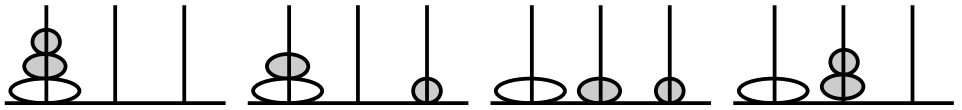
- Mit zwei Prozeduren (die Pfosten durch `int`-Werte 1,2,3 dargestellt)
  - `void bewegeTurm(int A, int Z, int E, int h)`
    zum Bewegen eines Turms der Höhe h von A über Z nach E
  - `void umlegeScheibe(int A, int E)`
    zum Bewegen einer Scheibe von A nach E

ergibt sich für *h=3* folgendes Spiel:

bewegeTurm(1,2,3, 3)

umlegeScheibe(1,3)

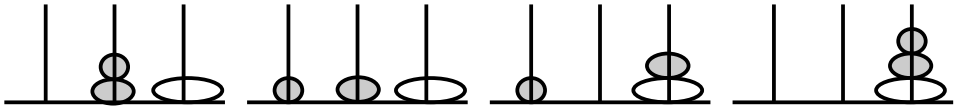
bewegeTurm(1,3,2, 2)



umlegeScheibe(1,2)

bewegeTurm(1,2,3, 1)	bewegeTurm(3,1,2,1)
umlegeScheibe(1,3)	umlegeScheibe(3,2)

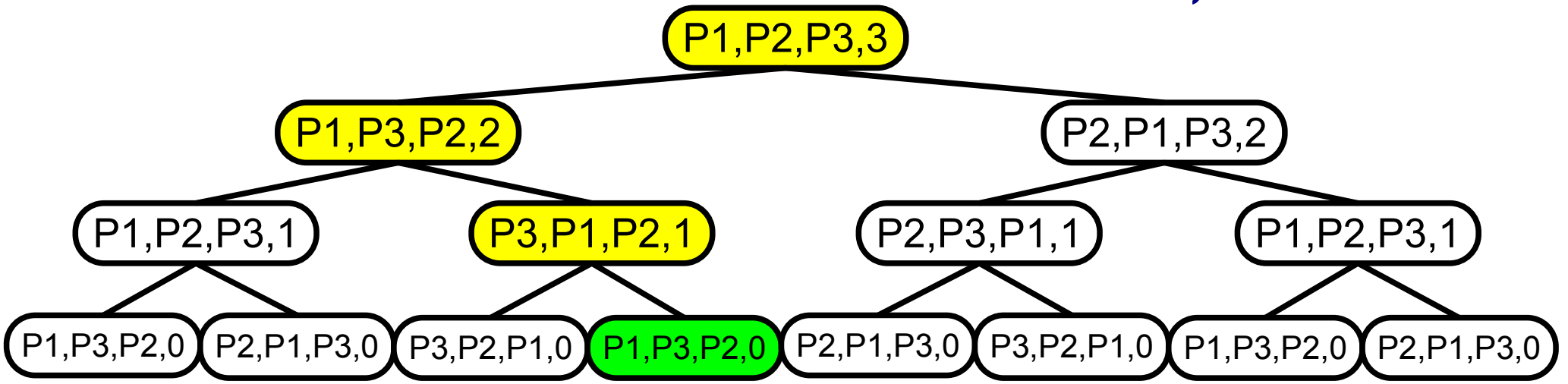
bewegeTurm(2,1,3, 2)



umlegeScheibe(2,3)

bewegeTurm(2,3,1, 1)	bewegeTurm(1,2,3, 1)
umlegeScheibe(2,1)	umlegeScheibe(1,3)

Türme von Hanoi: Aufrufstruktur, Kosten



- Die Aufrufe von bewegeTurm (einschließlich der mit *h=0*, die waren auf dem vorigen Bild weggelassen)
- Während des grün markierten „Transports“ sind die gelben begonnen, aber noch nicht abgeschlossen: Mehrere *Inkarnationen* der Funktion existieren also gleichzeitig
- Der Aufwand zum Bewegen eines Turms hängt offensichtlich (?) nur von dessen Höhe ab. Die Anzahl der Scheibenbewegungen dafür heiße *A(h).*
- Dann gilt *A(0)=0* und *A(h)=2A(h-1)+1* für *h>0*, was auf $A(h)=2^h-1$ führt. Die Zahl der Scheibenbewegungen wächst also exponentiell mit der Turmhöhe