Natürlich ist die Lösung mit der Maximallänge nicht elegant:

- Es wird möglicherweise nicht benötigter Speicherplatz reserviert.
- Es besteht die Gefahr dass der Indexbereich überschritten wird. Man kann zwar (und soll auch!) auf `n > nMax` testen und damit ungültige Speicher-zugriffe vermeiden. Aber es besteht keine Möglichkeit, den Array nachträglich zu redimensionieren.

Bessere Lösungen ermöglicht die später behandelte *dynamische Speicherverwaltung*.

Variablen mit mehreren Indices sind ebenfalls möglich. In der Mathematik wären dies Matrizen oder Tensoren.

Eine Rechteckmatrix mit 2 Zeilen und 3 Spalten wird deklariert als

```
double a[2][3];
```

Eine Initialisierung ist beispielsweise

```
double a[2][3] = { {11,12,13},
                   {21,22,23} };
```

und entspricht den Zuweisungen

```
a[0][0] = 11; a[0][1] = 12; a[0][2] = 13;
a[1][0] = 21; a[1][1] = 22; a[1][2] = 23;
```

Beispiel Matrix-Multiplikation. Wenn A eine $N_i \times N_k$ Matrix und B eine $N_k \times N_j$ Matrix ist, dann ist das Produkt C=AB eine $N_i \times N_j$ Matrix mit Koeffizienten

$$c_{ij} = \sum_{k=0}^{N_k} a_{ik} b_{kj}$$

```
for (int i = 0; i < Ni; i++) {
  for (int j = 0; j < Nj; j++) {
    c[i][j] = 0;
    for (int k = 0; k < Nk; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

Das Programm zur Matrix-Multiplikation kann auf Boole'sche Matrizen angewandt werden, wobei dann **+** durch **||** und ***** durch **&&** ersetzt werden muss.

Mit einer quadratischen Boole'schen Matrix

```
bool R[N][N];
```

kann eine *Relation* ausgedrückt werden.

Beispielsweise kann R(i,j) bedeuten: "Es gibt eine direkte Zugverbindung zwischen i und j ".

Das Quadrat R·R bedeutet dann: "Es gibt eine Zugverbindung mit einmaligem Umsteigen zwischen i und j ".

Eindimensionale Arrays über dem Datentyp **char** ergeben die *Strings* der Sprache C, die aber auch in C++ verwendbar sind.

C-Strings enden mit einem Nullzeichen, darum sind folgende zwei Initialisierungen äquivalent:

```
char str[10] = "Summe";
char str[10] = {'S','u','m','m','e',0};
```

Mit dem Nullzeichen kann der String begrenzt oder "geleert" weden. Beispiele:

```
str[3] = 0; cout << str; // Druckt: Sum
str[0] = 0; cout << str; // leerer String
```

Funktionen für Länge, lexikographisches Vergleichen, Konkatenation (Zusammenfügen) etc. werden mit

```
#include <cstring>
```

deklariert.

Alternativ dazu verfügt C++ über Strings, die nicht als Arrays sondern als Objekte einer Klasse **string** realisiert sind. Beispiel:

```
string s = "Hello"; cout << s;
```

Die C++ Strings erfordern

```
#include <string>
```

Es existieren ähnliche Funktionen wie für C-Strings aber auch überladene Operatoren, wie etwa **+** für Konkatenation oder **<** für lexikographischen Vergleich.

Verbund-Datentypen

Der Array ist eine Reihung von Elementen gleichen Typs. Der Verbund *(structure*, Pascal: *record)* ist dagegen eine Zusammenfassung von Elementen (auch: Komponenten) unterschiedlicher Typen.

Wie schon bei den Aufzähltypen gesehen, geht der Variablendeklaration eine Datentypdefinition voraus.

Die Syntax der Typdefinition ist:

```
struct Verbundname  {
    Element-Deklarationen
};
```

Beispiel:

```
struct Datum { int tag, monat, jahr; };

enum Geschlecht { maennlich, weiblich };

struct Person {
  char name[80];
  Datum geburtstag;
  Geschlecht geschlecht;
};
```

Wichtig:

- Die `struct` Deklaration endet immer auf `};`
- In einer `struct` Deklaration darf keine Initialisierung verwendet werden.

Beispiele einer Variablendeklaration:

```
Person a, studierende[278];
```

Beispiele für Zugriffe auf Elemente:

```
strcpy(a.name, "Ada Byron Lovelace");
a.geburtstag.tag   =   10;
a.geburtstag.monat =   12;
a.geburtstag.jahr  = 1815;
a.geschlecht = weiblich;
```

Variablen von Verbundtypen können initialisiert werden:

```
Person b = { "Charles Babbage",
             26, 12, 1791, maennlich };
```

struct.cpp

Welche Vorteile hat das Zusammenfassen zu einem Verbund?

- Der wichtigste Vorteil wird später klar: Zwei Unterprogramme können via eines einzigen Parameters die gesamte im Verbund gespeicherte Information austauschen.
- Der Verbund kann als Ganzes an eine andere Variable zugewiesen werden. Dabei werden alle Elemente kopiert, selbst wenn es sich um weitere Verbundtypen oder sogar um Arrays handelt.

Der Fall der Arrays ist deshalb bemerkenswert, weil Arrays selber nicht zuweisbar sind.

Vereinigungs-Datentypen

Vereinigungstypen *(unions)* haben eine ähnliche Syntax wie die Verbundtypen:

```
union Name {
   Element-Deklarationen
};
```

Im Unterschied zum Verbund werden die Elemente bei der Vereinigung aber im Speicher "übereinander-gelegt". Das heisst es wird pro Variable nur soviel Speicherplatz alloziert (reserviert) wie die grösste der Elemente benötigt.

In Pascal-Terminologie wird von einem "Record mit Varianten" gesprochen.

Vereinigungstypen werden hauptsächlich als Elemente eines Verbundes verwendet. Damit lassen sich Datenstrukturen mit Varianten definieren.

Beispiel:

```
struct Ortsangabe {
  enum typ = {Ortsname, PLZ, Koordinaten};
  union wert {
    char name[16]; int plz; float xy[2];
  };
};
Ortsangabe ort[1000];
```

Beispiel für Zugriff:

```
if (ort[k].typ == Ortsname){
  cout << "in der Ortschaft "
       << ort[k].wert.name << endl;
}
```

Eine weitere Anwendung von Vereinigungstypen besteht darin, den Inhalt einer Speicherzelle auf zwei verschiedene Arten zu interpretieren.

Dies wird selten gebraucht und ist nicht zu empfehlen wegen Maschinenabhängigkeiten.

Beispiel 1: Ein Buchstaben-Code der auch als Zahl benutzt werden kann.

```
union id { char alfa[4]; int numeric; };
```

Die Reihenfolge der Bytes in Ganzzahlen ist aber maschinenabhängig *(big endian* vs. *little endian)*.

Beispiel 2 (reine Spielerei):

- Gesucht: ganzer Teil des 2er-Logarithmus.
- Lösung: eine **`float`**-Zahl hat den Zweier-Exponenten als Teil des Bitmusters. Also kann man einen Typ

```
union Hack { float f; unsigned int u; };
```

verwenden, eine Variable

```
Hack x;
```

deklarieren, dann die Zahl an **`x.f`** zuweisen und schliesslich die Bits mittels der Bit-Operatoren **`&`** und **`>>`** aus **`x.u`** extrahieren. log2.cpp

Nebenbei: Für die Extraktion von Mantisse und Exponent aus Gleitkommazahlen gäbe es die Funktion **`frexp()`** in der **`<cmath>`** Bibliothek.

Unterprogramme

Unterprogramme sind das wichtigste Strukturierungsmittel beim Programmieren. Sie erlauben

- Teilaufgaben zu definieren und diese in je einem Unterprogramm zu behandeln,
- das Programm auf verschiedene Dateien aufzuteilen, die dann separat kompilierbare *Module* bilden,
- Programmbibliotheken zu erstellen (aus einem oder mehreren Modulen),
- Unterprogramme als *„black box“* zu betrachten, die man als korrekt voraussetzt, und von denen man nur die Schnittstelle kennen muss.

Einige Programmiersprachen unterscheiden zwei Arten von Unterprogrammen:

- *Funktionen*, die einen Wert zurückgeben, und
- *Prozeduren* (Pascal) resp. *Subroutinen* (Fortran), die eine Aktion ausführen.

In C und C++ gibt es dagegen nur die Funktionen. Es ist aber zugelassen, dass eine Funktion keinen Wert zurückgibt. Eine solche Funktion hat formal den leeren Rückgabetyp **`void`**.

Standardfunktionen

C++ besitzt eine Bibliothek von mathematischen Funktionen. Mit

```
#include <cmath>
```

integriert man deren Deklarationen ins Programm.

Funktionsdeklarationen haben die Form

```
Typ Funktionsname ( Parameterliste );
```

Dabei ist `Typ` der Datentyp des Rückgabewertes und `Parameterliste` die Liste der Funktionsargumente. Eine leere Liste `()` ist zulässig, mehrere Parameter werden durch Kommata getrennt.

Die Parameterliste kann zwei Formen haben:

- Sie kann nur Datentypen enthalten wie im Beispiel (des Potenzierens):

```
float pow(float, float);
```

  In diesem Fall spricht man von einem *Prototypen* der Funktion. Der Teil `pow(float, float)` heisst die *Signatur* der Funktion.

- Sie kann Datentypen und *formale Parameter* enthalten wie im Beispiel

```
float pow(float basis, float exponent);
```

In C++ darf, anders als in C, der gleiche Funktionsname für mehrere Funktionen gebraucht werden, sofern sich deren Signaturen unterscheiden. So kennt z.B. die Standard-Mathematikbibliothek zusätzlich eine Funktion

```
float pow(float, int);
```

die auch für negative Basen funktioniert.

Die folgende Liste gibt die Prototypen der Funktionen von **<cmath>** wieder. Es existieren aber zusätzlich zu den **float**-Versionen noch entsprechende Versionen für **double** (und möglicherweise für **long double**).

```
float acos (float);           // Arcus cosinus
float asin (float);           // Arcus sinus
float atan (float);           // Arcus tangens
float atan2(float, float);    // atan(y/x) (x bel.)
float ceil (float);           // aufrunden
float cos  (float);           // Cosinus
float cosh (float);           // Cosinus hyperbolicus
float exp  (float);           // Exponentialfunktion
float fabs (float);           // Absolutbetrag
float floor(float);           // abrunden
float frexp(float, int*);     // Mantisse und Exponent
float fmod (float, float);    // Divisionsrest
float log  (float);           // Logarithmus naturalis
float log10(float);           // Zehner-Logarithmus
float pow  (float, float);    // x hoch y (x >= 0)
float pow  (float, int);      // x hoch y (x bel.)
float sin  (float);           // Sinus
float sinh (float);           // Sinus hyperbolicus
float sqrt (float);           // Quadratwurzel
float tan  (float);           // Tangens
float tanh (float);           // Tangens hyperbolicus
```

Funktionen werden benützt indem man sie vom Hauptprogramm oder einem anderen Unterprogramm aus *aufruft*.

Der Funktionsaufruf bildet einen *Ausdruck*. Damit kann er z.B. auf der rechten Seite einer Zuweisung stehen, oder aber Teil eines grösseren Ausdrucks sein.

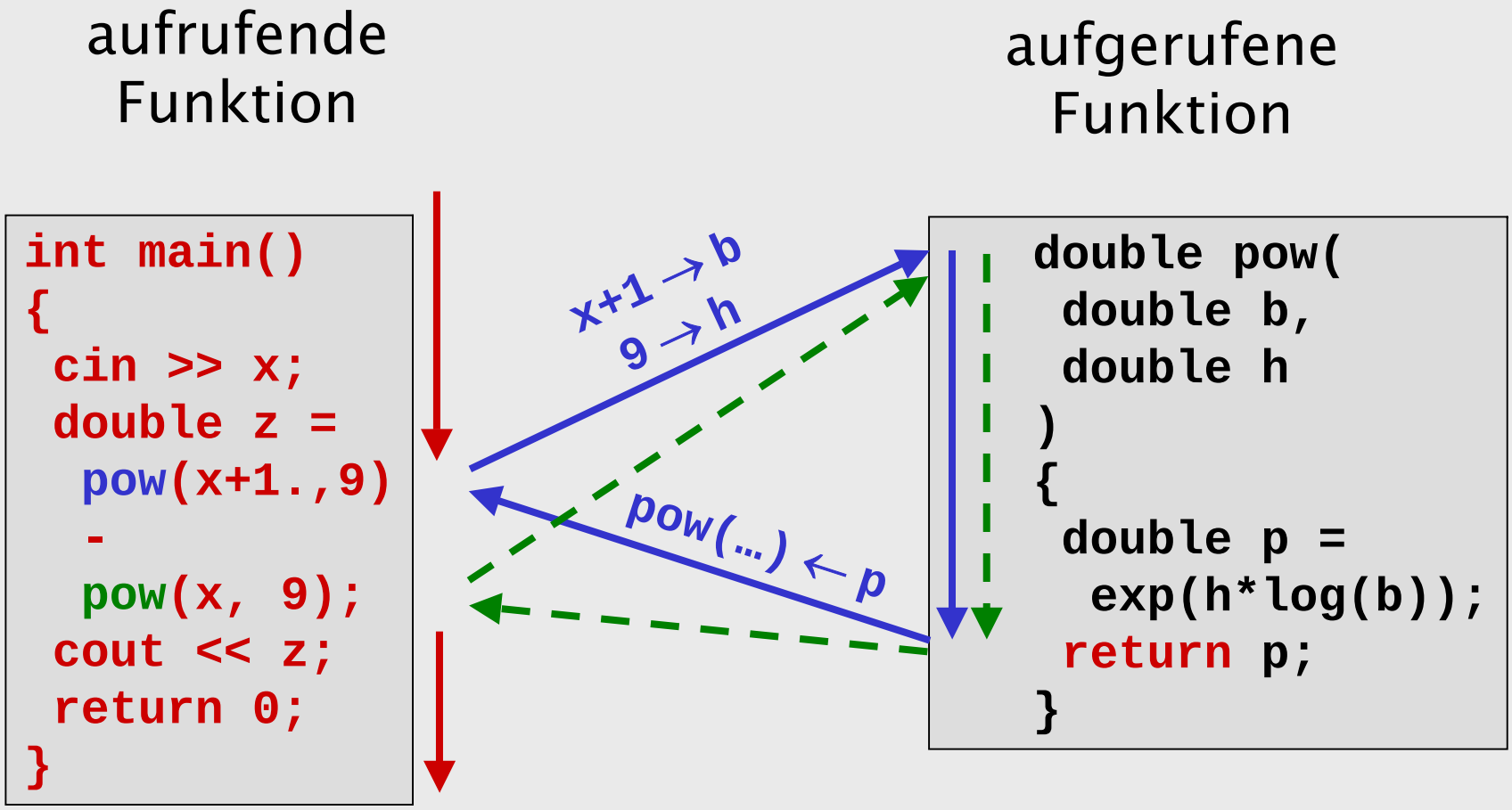
Beim Funktionsaufruf werden sog. *aktuelle Parameter* in die Parameterliste eingesetzt, das sind Ausdrücke deren Datentyp mit dem der formalen Parameter resp. der Signatur übereinstimmt.

Beispiel: `double z = pow(x+1., 9) - pow(x, 9);`

Kann es passieren, dass das Unterprogramm Variablen überschreibt?

Da nur die ausgewerteten aktuellen Parameter an das Unterprogramm weitergegeben werden, besteht diese Gefahr nicht. Das Unterprogramm müsste dazu die Adresse der Variablen kennen.

Beispiel: Der Wert der Variablen `x` kann durch den Aufruf `pow(x+1., 9)` nicht verändert werden, selbst dann nicht wenn die Funktion `pow` fehlerhaft programmiert wäre. Auch mit dem Aufruf `pow(x,9)` kann dies nicht passieren.


aufrufende Funktion
aufgerufene Funktion
int main()
{
cin >> x;
double z =
pow(x+1.,9)
-
pow(x, 9);
cout << z;
return 0;
}
x+1 → b
9 → h
pow(…) ← p
double pow(
double b,
double h
)
{
double p =
exp(h*log(b));
return p;
}

Selber definierte Funktionen

Die *Deklaration* einer Funktion ist die Angabe des Prototyps. Damit wird die Schnittstelle formal beschrieben („wieviele Argumente gibt es und von welchem Typ sind sie? Was ist der Typ des Rückgabewerts?“).

Die *Definition* ist dagegen die vollständige Beschreibung der Funktion. Die Syntax der Funktionsdefinition ist

```
Typ Funktionsname ( Parameterliste )
  Block
```

wobei in der Parameterliste diesmal nicht nur Datentypen sondern auch dazugehörige formale Parameter stehen müssen.

Es gilt zu beachten:

- Funktionen müssen in jedem Modul (d.h. in jeder Datei) in dem sie aufgerufen werden, *vor dem ersten Aufruf* deklariert werden.
- Befindet sich die Definition in einem anderen Modul, so muss der Deklaration das Schlüsselwort **`extern`** voran-gestellt werden.
- Die Definition gilt auch als Deklaration.
- Funktionen dürfen nicht mehrfach definiert werden (auch nicht in verschiedenen Modulen).

Weil eine überflüssige **`extern`**-Deklaration nicht stört, ist es zweckmässig, Funktionsdeklarationen in Header-Dateien auszulagern und diese dann in jedem Modul zu benützen.

Die Anweisung

```
return Ausdruck ;
```

bewirkt das Verlassen der Funktion und die Rückgabe von `Ausdruck` .

Die Anweisung

```
return;
```

bewirkt das Verlassen einer als `void` deklarierten Funktion.

Funktionsdefinitionen können durchaus mehrere `return`-Anweisungen enthalten. Der Block muss mit einer solchen enden (Ausnahme: bei einer Schleife der Art `while(true) { … }`).

Beispiel: Die Berechnung des grössten gemeinsamen Teilers kann jetzt als Funktion geschrieben werden:

```
int ggT(int a, int b)
{
   while (b > 0) {
     int c = a%b;  a = b;  b = c;
   }
   return a;
}
```

Wie man sieht, dürfen die formalen Parameter **a** und **b** verändert werden. Sie verhalten sich wie lokale Variablen, die beim Aufruf einen Wert zugewiesen erhalten.

Die Variablen der aufrufenden Funktion werden nicht tangiert:

```
int m = 14, n = 35;
cout << ggT(m, n) << " ist ggT von ";
cout << m << " und " << n << endl;
```

Variablenparameter

Die bis jetzt behandelten Parameter waren sogenannte *Wertparameter*. Dabei erhält das aufgerufene Unterprogramm eine *Kopie* des aktuellen Parameters.

Bei *Variablenparametern* erhält das Unterprogramm dagegen die *Speicheradresse.* Das Unterprogramm kann deshalb die Variable nicht nur lesen, sondern auch verändern.

Variablenparameter kennzeichnet man durch ein **&** (den *Referenzoperator*) vor dem formalen Parameter.

Eine typische Anwendung von Variablenparameter sind Funktionen die mehrere Werte berechnen sollen.

Beispiel: kartesische in Polarkoordinaten umrechnen:

```
void polar(double x, double y,
           double& rho, double& phi)
{
    rho = sqrt(x*x + y*y);
    phi = atan2(x, y); return;
}
```

Aufruf:

```
double x = 4., y = 3., laenge, winkel;
polar(x, y, laenge, winkel);
cout << "  Laenge:" << laenge <<
        ", Winkel:" << winkel << endl;
```

Weitere Beispiele von Funktionen mit Variablenparametern gibt es in der Bibliothek **`iostream`**.
Eine solche Funktion ist **`cin.get(char& c)`**, die das nächste Zeichen einschliesslich *whitespace* liest.
(Funktionsnamen wie **`cin.get`** beschreiben Elementfunktionen, die erst später behandelt werden).

Die Sprache C kennt keine Variablenparameter.
Darum sind z.B. die Funktionen in der Bibliothek **`cstring`** mit Zeigern programmiert (später behandelt).

Weiteres Beispiel: Eine Funktion **swap** welche die Werte zweier Variablen vertauscht.

Die Parameter sind hier sowohl Eingabe- als auch Ausgabeparameter.

```
void swap(int& i, int& j)
{
    int h = i; i = j; j = h;
}
```

Aufruf:

```
int sechs = 7, sieben = 6;
swap(sechs, sieben);
cout << "sechs:"  << sechs << endl;
cout << "sieben:" << sieben << endl;
```

Arrays als Parameter

Die Regel dass bei Wertparametern Werte kopiert und bei Variablenparametern Adressen übergeben werden, gilt auch für zusammengesetzte Datentypen, z.B. den Verbund.

Der Array macht hier eine scheinbare Ausnahme: Arrays werden bei der Übergabe nicht kopiert.

Der Grund ist: In C/C++ wird ein Array intern so behandelt wie eine Speicheradresse (des nullten Elementes). Beim Unterprogrammaufruf wird deshalb nur diese Adresse kopiert, nicht aber der ganze Array.

Dass Arrays allein durch ihre Anfangsadresse repräsentiert werden, bedeutet auch dass die Arraylänge nirgends gespeichert ist.

Entsprechend ist es möglich, Funktionen zu schreiben, die beliebige Arraylängen zulassen, wie etwa:

```
float median(float folge[]);
```

Die Dimensionierungs-Klammer `[]` wird also einfach leer gelassen. Dadurch sind Aufrufe mit verschieden langen `float`-Arrays möglich.

Die Länge muss der Funktion aber mitgeteilt werden, wozu es mehrere Möglichkeiten gibt :

- fix vereinbart
- separater Parameter
- spezieller Wert als Schlussmarkierung.

Letztere Technik wird von den Funktionen in **<cstring>** verwendet, weil ja bei C-Strings eine solche Schluss-markierung existiert (das Zeichen mit ASCII-Code 0).

Beispiel: Die Funktion **strcpy** zum Kopieren von C-Strings. Sie ist deklariert als

```
char* strcpy(char* dest, const char* src);
```

was (wie wir später sehen werden) gleichwertig ist mit

```
char* strcpy(char dest[], const char src[]);
```

Beim (meist ignorierten) Rückgabewert gibt es keine Alternative zur Zeiger-Schreibweise.

Das Hauptprogramm

Das Hauptprogramm **main** ist eine selber definierbare Funktion. Es sind zwei Prototypen deklariert:

Neben dem Prototyp

```
int main();
```

existiert der Prototyp

```
int main(int argc, char **argv);
```

manchmal auch geschrieben als

```
int main(int argc, char *argv[]);
```

Die Variable **argv** ist ein Array von C-Strings. Beim "Aufruf" von **main** durch das Betriebssystem wird dem Element **argv[i]** das i-te Argument der Kommandozeile übergeben, wobei das Programm selbst als nulltes Argument gezählt wird.

Die Variable **argc** erhält die Anzahl dieser Argumente (inklusive dem nullten).

Um Zahlen von C-Strings nach **int** resp. **double** zu konvertieren, verwendet man die Funktionen:

```
#include <cstdlib>
double atof (const char str[])
int    atoi (const char str[])
```

Der durch `return` [`Ausdruck`] `;` produzierte (ganzzahlige) Rückgabewert kann vom Betriebssystem interpretiert werden, z.B. innerhalb von Scripts.
Als Konvention gilt, dass bei fehlerfreiem Abschluss ein Wert von Null zurückgegeben wird.

Mit `exit (` [`Ausdruck`] `);` kann man das Programm auch aus einem Unterprogramm heraus verlassen.

Inline-Funktionen

Der Aufruf eines Unterprogramms ist mit einem gewissen Aufwand verbunden:

- Die Registerinhalte werden auf den Stack “gerettet”.
- Die aktuellen Parameter und die Rücksprungadresse werden dem Unterprogramm via Stack übergeben.
- Vor dem Rücksprung gibt das Unterprogramm den Rückgabewert auf den Stack.
- Nach dem Rücksprung wird der Stack wieder abgebaut und die alten Registerinhalte werden wieder hergestellt.

Eine *Inline-Funktion* wird dagegen *nicht aufgerufen*.
Der Compiler ersetzt den Aufruf durch den Inhalt der Funktionsdefinition, wobei für die formalen Parameter die aktuellen Parameter(-ausdrücke) eingesetzt werden.

Inline-Funktionen machen (wenn mehrfach benutzt) das kompilierte Programm länger.

Sie bringen dafür einen Zeitgewinn

- wenn es sich um kleine Funktionen handelt, und
- wenn sie in innersten Schleifen gebraucht werden.

Beispiel: Die Inline-Funktion

```
inline double det(double mat[2][2])
{
  return mat[0][0] * mat[1][1] -
         mat[0][1] * mat[1][0];
}
```

kann z.B. so verwendet werden:

```
double produkt = det(a) * det(b);
```

und wird dann übersetzt als:

```
double produkt =
  (a[0][0]*a[1][1] - a[0][1]*a[1][0]) *
  (b[0][0]*b[1][1] - b[0][1]*b[1][0]);
```

det.cpp

Für Inline-Funktionen gilt, im Unterschied zu anderen Funktionen:

- Die Funktion muss definiert (nicht bloss deklariert) werden, bevor sie zum ersten Mal benutzt wird. Die Definition beginnt mit dem Schlüsselwort **`inline`**.
- Die Definition darf wiederholt vorkommen.

Inline-Funktionen, die in mehreren Modulen gebraucht werden, *definiert* man am besten in einer Header-Datei.

Rekursion

Funktionen dürfen andere Funktionen aufrufen, aber auch sich selbst. Man spricht dann von *Rekursion*.

Beispiel:

```
int ggT(int a, int b)
{
   if (b == 0) return a;
   else return ggT(b, a%b);
}
```

recGGT.cpp

Verlauf: `ggT(8,12)` → `ggT(12,8)` → `ggT(8,4)` → `ggT(4,0)` → `4`

Wichtig:

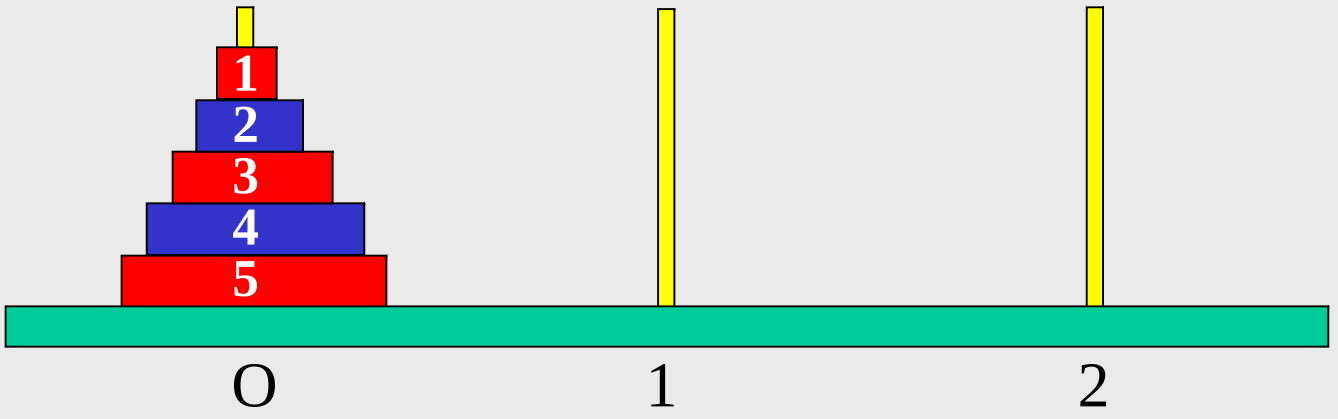
- Es braucht eine *Verankerung:* für bestimmte Parameterwerte bricht die Rekursion ab. Die Verankerung entspricht der Abbruchbedingung bei der **`while`**-Schleife.
- Die Parameter ändern sich bei jedem Aufruf und erreichen nach endlich vielen Malen eine Verankerung. Typischerweise werden die Parameter jedes Mal kleiner.

Die Rekursion kann als Alternative zur *Iteration* (**`while`**-Schleife) verwendet werden. Sie ist gleich mächtig wie diese im Sinne der Berechenbarkeit.

Die Türme von Hanoi

Die Rekursion ist besonders dann geeignet, wenn bereits die Problemstellung rekursiv ist.

Beispiel: Das Problem der Türme von Hanoi: Der Turm aus *n* Scheiben soll vom Platz 0 auf den Platz 2 gebracht werden. Dabei darf immer nur eine Scheibe aufs Mal den Platz wechseln und es darf nie eine grössere Scheibe auf eine kleinere gelegt werden.



Die Lösung kann mit Rekursion sehr einfach formuliert werden.

Man kann nämlich den Turm der Scheiben 1 bis k von Platz x auf Platz y bringen, indem man:

1. den Turm der Scheiben 1 bis $k-1$ (rekursiv) von Platz x auf Platz z bringt (Zwischenplatz, $z = 3-x-y$),
2. die Scheibe k auf Platz y setzt, und
3. den Turm der Scheiben 1 bis $k-1$ (rekursiv) von Platz z auf Platz y bringt.

Mit dem Aufruf `hanoi(n,0,1,2);` erzeugt die folgende Funktion die richtige Sequenz von Scheibenbewegungen:

```
void hanoi(int anzahl, int von, int ueber, int nach)
{
    if (anzahl == 1) {
      cout << von << "->" << nach << "\n";
    }
    else {
        hanoi(anzahl-1, von, nach, ueber);
        hanoi(1, von, ueber, nach);
        hanoi(anzahl-1, ueber, von, nach);
    }
}
```

hanoi.cpp

Der Stack und statische Variablen

Interessant ist, dass über den Zustand der drei Türme gar nicht Buch geführt werden muss.

Der Zustand ist aber wenigstens teilweise auf dem Stack gespeichert, denn:

Die Parameter einer Funktion werden auf dem Stack abgelegt. Im Falle von rekursiven Aufrufen gibt es mehrere *Inkarnationen* jedes Parameters.

Bild des Stacks während der Ausführung: