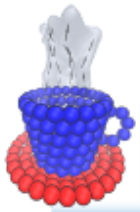


Computergrundlagen Programmieren in C

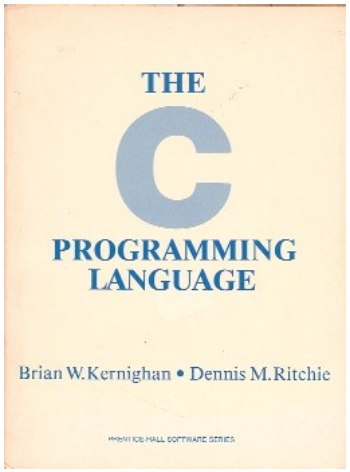
Axel Arnold

Institut für Computerphysik
Universität Stuttgart

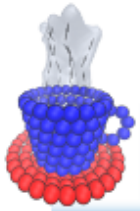
Wintersemester 2013/14

Die Sprache C

D. M. Ritchie, 1941 – 2011



- Entwickelt 1971-1973
- aktueller Standard: C11 (2011), wir benutzen C99
- Compiler: GNU gcc, Intel icc, IBM XL C, PGI Compiler, ...
- geeignet für effiziente und hardwarenahe Programme
- Python-Interpreter und Linux-Kernel sind in C geschrieben
- manuelle Speicherverwaltung, dadurch fehleranfälliger



Hello, World!

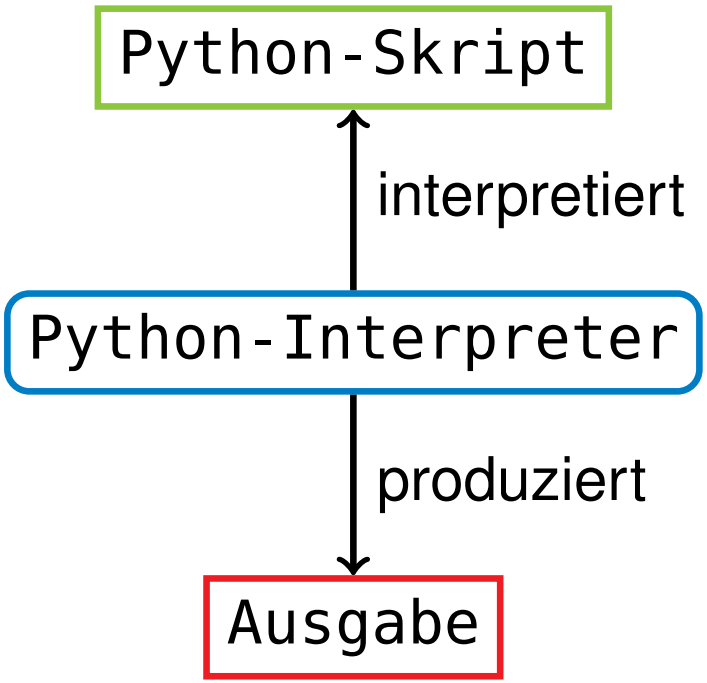
```c
#include <stdio.h>
// main program
int main()
{
  printf("Hello, World\n");
  return 0;
}
```

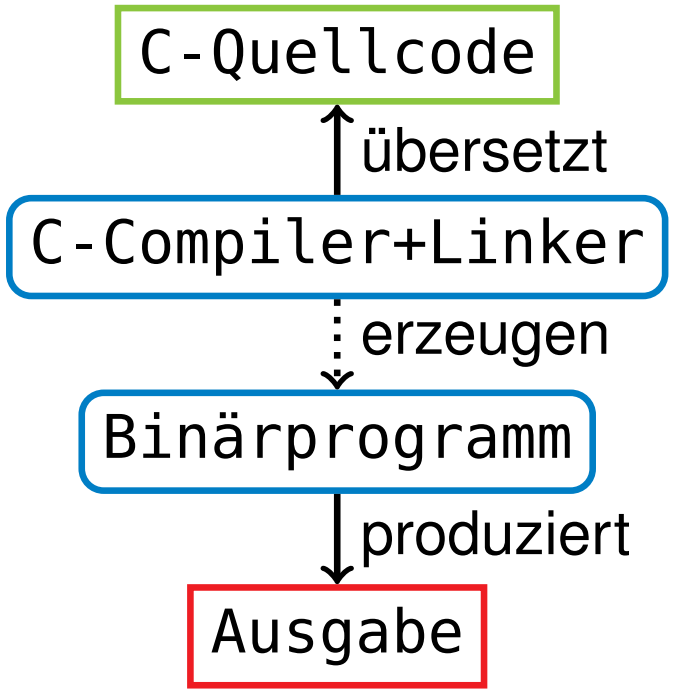
- C-**Quellcode** muss als Textdatei vorliegen (z.B. `helloworld.c`)
- Formatierung freier als in Python, siehe auch `http://www.ioccc.org/main.html`
- Vor der Ausführung mit dem GNU-Compiler **compilieren**:

```
gcc -Wall -O3 -std=c99 -o binary helloworld.c
```

- Erzeugt ausführbares Programm `binary`
- Warnt bei möglichen Fehlern (-Wall), optimiert (-O3) und wählt Standard C99

Interpreter- vs. Compilersprachen





- **Interpreter** liest Programm bei Ausführung und führt Anweisungen aus
- Interpreter zur Ausführung nötig

- **Compiler** übersetzt Programm in Maschinensprache
- Programm läuft ohne Compiler, aber nur auf Zielhardware + Betriebssystem

Assembler und Maschinensprache

- Maschinensprache sind Zahlenkolonnen

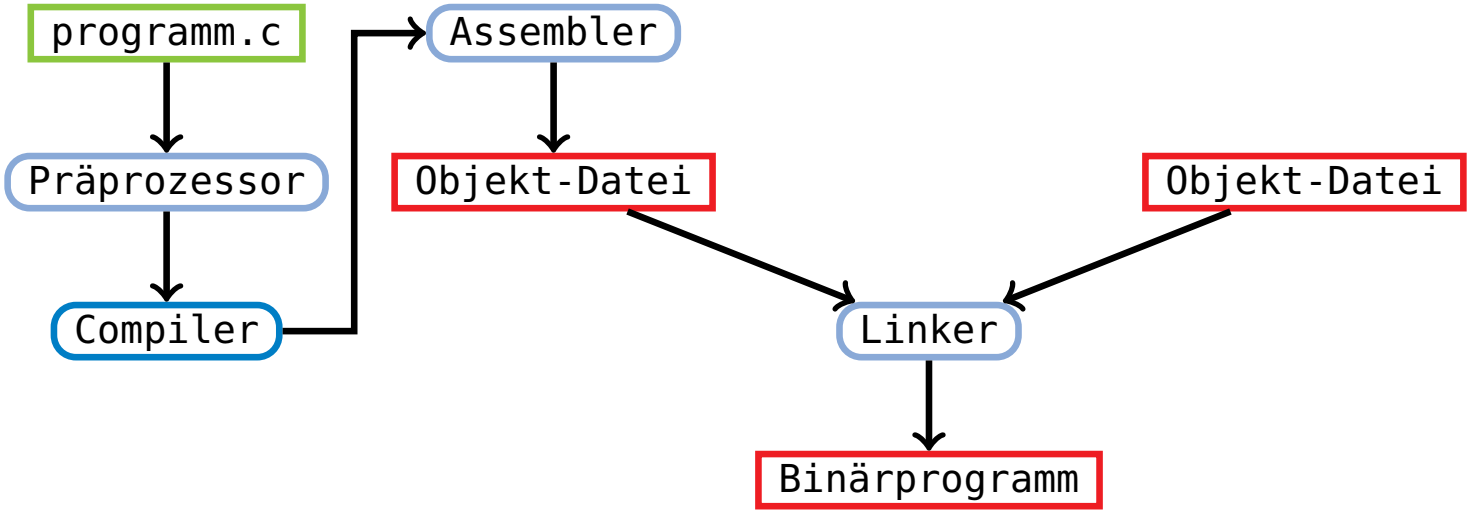
```
AD 34 12 18 69 2A 8D 34 12 AD 35 12 69 00 8d 35 12
```

- Assembler ist die menschenlesbare Form:

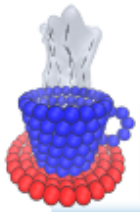
```
        # Maschinensprache # Assembler
                           X    = $1234
        AD 34 12           LDA X
        18                 CLC
        69 2A              ADC #42
        8D 34 12           STA X
        AD 35 12           LDA X+1
        69 00              ADC #00
        8d 35 12           STA X+1
```

- Zeitkritische Anwendungen werden manchmal direkt in Assembler geschrieben

Vom Sourcecode zum Programm



- Ein (C-)Programm durchläuft viele Schritte bis zur fertigen ausführbaren Datei
- **Präprozessor**, **Compiler**, **Assembler** und **Linker** sind meist separate Programme
- meist mehrere Objektdateien aus verschiedenen Quelltextdateien



Komponenten

- **Präprozessor** ersetzt Code textuell
  - **`#include`** bindet weiteren Quellcode ein
  - **`#define`** definiert Makros (Textersetzung)
- **Compiler** erzeugt Assembler aus präprozessierten Quellcode
  - übersetzt C in Zwischencode
  - optimiert, weist Prozessorregister zu
  - übersetzt dann in Assembler
- **Assembler** erzeugt eine Objektdatei mit Maschinencode
- **Linker** verbindet Objektdateien zu **Binärprogramm**
  - löst Abhängigkeiten (Aufrufe, globale Variablen) unter den Objektdateien auf
  - Objektdateien können in **Bibliotheken** zusammengefasst werden

Beispiel: Fakultät

Berechnen der Fakultät in C:

```
#include <stdio.h>
int fakultaet(int n) {
  int fak = 1;
  for (int i = 2; i <= n; ++i) {
    fak *= i;
  }
  return fak;
}
int main(int argc, char **argv)
{
  int n = 20;
  printf("%d\n", fakultaet(n));
  return 0;
}
```

Funktionen

```
#include <math.h>
void init(float a)
{
  if (a <= 0) return;
  printf("%f\n", log(a));
}
float max(float a, float b)
{
  return (a < b) ? b : a;
}
```

- Funktionsdefinition
  `rettyp funktion(typ1 arg1, typ2 arg2,...) {...}`
- Ist der Rückgabetyp **`void`**, gibt die Funktion nichts zurück
- **`return`** verlässt wie in Python eine Funktion vorzeitig
- Hauptprogramm `main` ist auch Funktion

Datentypen in C

- anders als in Python sind Variablen fest typisiert
- Grunddatentypen

`void`	(leer) Datentyp, der nichts enthält nötig für manche Sprachkonstrukte	
`char`	8-Bit-Ganzzahl, für Zeichen	'1','a','A',...
`int`	32- oder 64-Bit-Ganzzahl	1234, -56789
`float`	32-Bit-Fließkommazahl	3.1415, -6.023e23
`double`	64-Bit-Fließkommazahl	-3.1415, +6.023e23

- **Arrays** (Felder): ganzzahlig indizierter Vektor fester Größe
- **Pointer** (Zeiger): Verweise auf Speicherstellen
- **Structs und Unions**: zusammengesetzte Datentypen, Verbünde
- *keine* Listen oder Wörterbücher

Variablen

```
int foo() {
  int i = 0;
  int j, k;
  int i;  // Fehler! i doppelt deklariert
}

void bar() {
  int i = 2; // Ok, da anderer Gueltigkeitsbereich
  i = k;     // Fehler! k unbekannt
}
```

- *Müssen* vor Benutzung mit ihrem Datentyp **deklariert** werden
- Dürfen *nur einmal* deklariert werden
- Können bei der Deklaration mit Startwert **initialisiert** werden
- Mehrere Variablen desselben Typs mit „,“ getrennt deklarieren
- Gültigkeitsbereich ist innerster Block, markiert durch „{}“

Globale Variablen

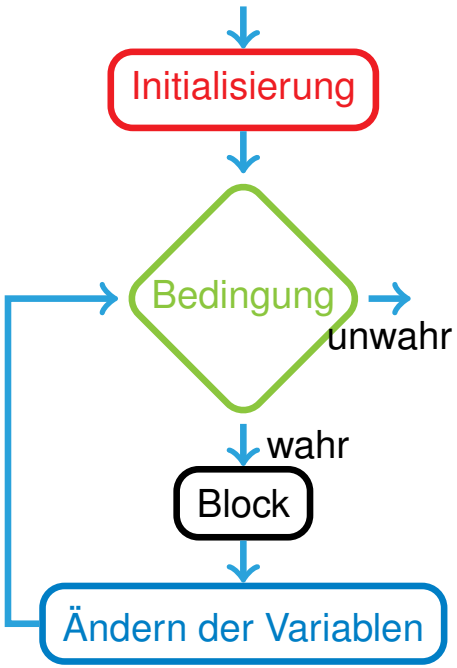
```
int global;

void foo() {
  printf("%d\n", global);
}

int bar() {
  global = 2;
  funktion();
}
```

- Globale Variable werden außerhalb von Funktionen deklariert
- Aus allen Funktionen les- und schreibbar

Schleifen – for

```
for (int i = 1; i < 100; ++i) {
  printf("%d\n", i);
}
int k;
for (k = 100; k > 0; k /= 2) { printf("%d\n", k); }
```

- Initialisierung
  - Beliebige Anweisung
  - Deklarierte Variable nur in Schleife gültig
- Wiederholungsbedingung
  Schleifenende, wenn unwahr
- Ändern der Schleifenvariablen
  hier `i` um eins erhöhen, `k` durch 2 teilen
- Abbruch wie in Python mit **`break`** / **`continue`**
- Alle Teile können leer sein



Inkrement und Dekrement

Kurzschreibweisen zum Ändern von Variablen:

- `i += v, i -= v; i *= v; i /= v`
  - Addiert *sofort* v zu i (zieht v von `i` ab, usw.)
  - Wert im Ausdruck ist der *neue* Wert von `i`

```
int k, i = 0;
k = (i += 5);
printf("k=%d i=%d\n", k, i); → i=5 k=5
```

- `++i` und `--i` sind Kurzformen für `i += 1` und `i -= 1`
- `i++` und `i--`
  - Erhöhen / erniedrigen i um 1 *nach* Auswerten des Ausdrucks
  - Wert im Ausdruck ist also der *alte* Wert von `i`

```
int k, i = 0;
k = i++;
printf("k=%d i=%d\n", k, i); →i=1 k=0
```

Beispiel: Pythagoräische Zahlentripel

Der schnelle Algorithmus in C:

```
#include <stdio.h>
int main() {
  const int c = 10;
  int a = 1, b = c - 1;
  // walk along the arc till diagonal
  while (a <= b) {
    if (a*a + b*b < c*c) { a += 1; }
    else if (a*a + b*b > c*c) { b -= 1; }
    else {
      // found a Pythagorean triple for c
      printf("%d^2 + %d^2 = %d^2\n", a, b, c);
      break;
    }
  }
}
```

Bedingte Ausführung – if

```
if (anzahl == 1) { printf("ein Auto\n"); }
else             { printf("%d Autos\n", anzahl); }
```

- `if` wie in Python
- Es gibt allerdings kein `elif`

Bedingungen

Ähnlich wie in Python, aber

- logisches „und“: „&&“ statt „and“
- logisches „oder“: „||“ statt „or“
- logisches „nicht“: „!“ statt „not“

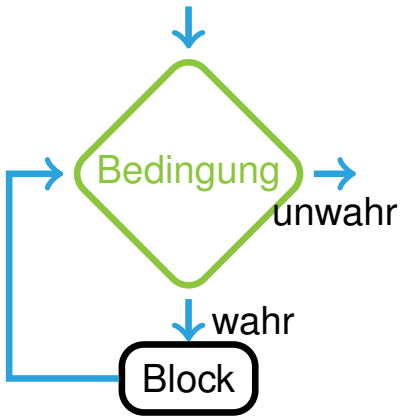
Also z.B.: `!((a == 1) || (a == 2))`

Schleifen – while

```
int i = 0;
while (i < 10) {
  summe += i;
  ++i;
}
```

- **while** (cond) block
  führt block aus, solange cond wahr ist
- **break** und **continue** möglich
- Beispiel ist äquivalent zu

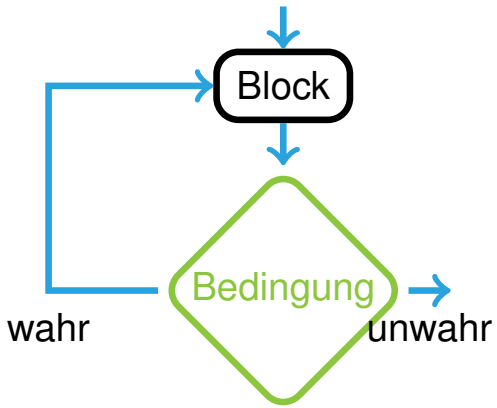
```
for(int i = 0; i < 10; ++i) {
  summe += i;
}
```



Schleifen – do ... while

```
int i = 0;
do {
 summe += i; ++i;
} while (i < 10);
```

- **do** block **while** (cond);
  führt block aus, solange die Bedingung cond wahr ist
- Unterschied zur **while**-Schleife:
  do ... while überprüft nach dem Block



- Ist auch äquivalent zu

```
for(int i = 0; i < 10; ++i) { summe += i; }
```

- Jede Schleife kann äquivalent als **for**-, **while**- oder **do...while**-Schleife geschrieben werden

Beispiel: Sieb des Eratosthenes

- **Problem**
  Gegeben: Eine ganze Zahl N
  Gesucht: Alle Primzahlen kleiner als N
- **Methode**: Sieb des Eratosthenes
  - Betrachte Liste aller Zahlen zwischen 2 und N
  - Streiche nacheinander alle echten Vielfachen von (Prim-)zahlen
  - Was übrig bleibt, sind Primzahlen $\Longrightarrow$ Sieb

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

Implementation

```
#include <stdio.h>
int main() {
  const int N = 100;
  // initially, assume all numbers > 1 are prime
  int is_prime[N];
  for (int i = 2; i < N; ++i) is_prime[i] = 1;
  // now remove true multiples
  for (int i = 2; i < N/2; ++i) {
    if (!is_prime[i]) continue; // multiples already deleted
    for (int multiple = 2*i; multiple < N; multiple += i)
      is_prime[multiple] = 0;
  }
  // print primes to console
  for (int i = 2; i < N; ++i) {
    if (is_prime[i]) printf("%d, ", i); }
  printf("\n");
}
```

Arrays

```
float x[3] = {0, 0, 0};
float A[2][3];
for (int i = 0; i < 2; ++i) {
  for (int j = 0; j < 3; ++j) {
    A[i][j] = 0.0;
  }}
x[10] = 0.0; // compiliert, aber Speicherzugriffsfehler
```

- Arrays (Felder) werden mit eckigen Klammern indiziert
- Mehrdimensionale Arrays erhält man durch mehrere Klammern
- Beim Anlegen wird die Speichergröße festgelegt
- Später lernen wir, wie man Arrays variabler Größe anlegt
- Es wird nicht überprüft, ob Zugriffe innerhalb der Grenzen liegen
- Die Folge sind Speicherzugriffsfehler (segmentation fault)

const – unveränderbare Variablen

```
static const float pi = 3.14;

pi = 5; // Fehler, pi ist nicht schreibbar

// Funktion aendert nur, worauf ziel zeigt, nicht quelle
void strcpy(char *ziel, const char *quelle);
```

- Datentypen mit **`const`** sind konstant
- Variablen mit solchen Typen können nicht geändert werden
- Verwendung wie benamte Konstanten
- „ pi “ ist viel klarer als 3.14 im Quelltext
- **`static`** bei globalen Konstanten

Beispiel: Zeichenketten

In Python:

```
string1 = "Hallo "; string2 = "Welt"
joined = string1 + string2
print "{} = {}{}".format(joined,string1,string2)
```

wird in C zu:

```
#include <stdio.h>
#include <string.h>
int main() {
  char string1[] = "Hallo ", string2[] = "Welt";
  // result length including terminating 0
  int len = strlen(string1) + strlen(string2) + 1;
  char joined[len];
  strncpy(joined, string1, len - 1);
  strncat(joined, string2, len - strlen(joined) - 1);
  printf("%s = %s%s\n", joined, string1, string2);
}
```

Zeichenketten

```
char string[] = "Ballon";
string[0] = 'H';
string[5] = 0;
printf("%s\n", string); → Hallo
```

- Strings sind Arrays von Zeichen (Datentyp **char**)
- Das String-Ende wird durch eine Null markiert
- Daher ist es einfach, mit Strings Speicherzugriffsfehler zu bekommen
- Zusammenhängen usw. von Strings erfordert Handarbeit oder Bibliotheksfunktionen

Stringfunktionen

```
#include <string.h>
char test[1024];
strncpy(test, "Hallo", 1024);
strncat(test, " Welt!", 1023 - strlen(test));
if (strncmp(test, argv[1], 2) == 0)
  printf("%s und %s starten gleich\n", test, argv[1]);
```

- Headerdatei **`string.h`** einbinden
- `strlen(quelle)`: Länge eines 0-terminierten Strings
- `strncpy(ziel, quelle, lange)`: kopiert eine Zeichenkette
- `strncat(ziel, quelle, lange)`: hängt eine Zeichenkette an
- `lange` begrenzt die Länge des zu kopierenden Teils
- Korrekte Größe des Zielbereichs und terminierende 0 beachten!
- `strncmp(quelle1, quelle2, lange)`: vergleicht zwei Zeichenketten

memcpy und memset

```
#include <string.h>

float test[1024];
memset(test, 0, 1024*sizeof(float));
// erste Haelfte in die zweite kopieren
memcpy(test, test + 512, 512*sizeof(float));
```

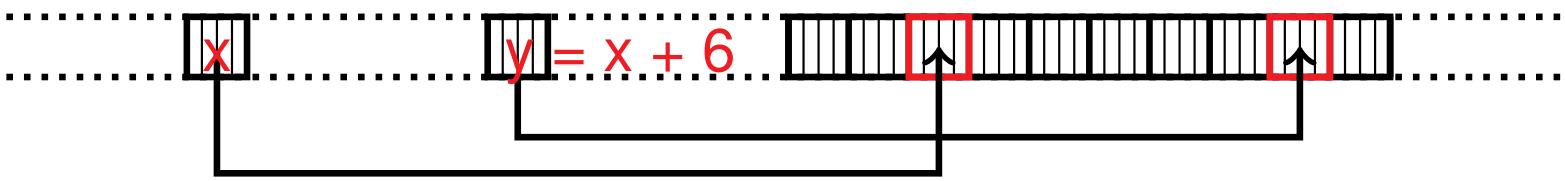
- Lowlevel-Funktionen zum Setzen und Kopieren von Speicherbereichen
- Z.B. zum initalisieren oder kopieren von Arrays
- `memset(ziel, wert, groesse)`: füllt Speicher byteweise mit dem *Byte* `wert`
- `memcpy(ziel, quelle, groesse)`: kopiert `groesse` viele *Bytes* von `quelle` nach `ziel`
- keine Null-Terminierung
- **`sizeof`** gibt die Größe eines Datentyps an

printf und scanf

```
#include <stdio.h>
char  kette[11];
float fliess;
int   ganz;
if (scanf("%10s %f %d", kette, &fliess, &ganz) == 3) {
  printf("%s %10.3f %d\n", kette, fliess, ganz);
}
```

- Lesen und Schreiben von Standardein-/ausgabe
- Headerdatei **`stdio.h`** einbinden
- Ganzahlen **%d, %x**, Fliesskomma **%e, %f, %g**, Strings **%s**
- Kann mit **Länge.Genauigkeit** ergänzt werden (hier 10 Buchstaben, 3 Stellen in `%10.3f`)
- `scanf` gibt Anzahl gelesener Werte zurück, Werte in den gegebenen **Zeigern** auf die Variablen
- Bei Strings auf genug Platz und abschließende 0 achten

Datentypen – Zeiger



- Zeigervariablen (Pointer) zeigen auf Speicherstellen
- Sonst normale Variablen, die selber im Speicher liegen
- Ihr Datentyp bestimmt, als was der Speicher interpretiert wird (**`void`** $*$ ist unspezifiziert)
- Es gibt keine Kontrolle, ob die Speicherstelle gültig ist (existiert, les-, oder schreibbar)
- Pointer verhalten sich wie Arrays, bzw. Arrays wie Pointer auf ihr erstes Element
- Funktionen können Variablen ändern, auf die sie Zeiger bekommen

Zeiger in C

```
char x[] = "Hallo Welt";
x[5] = 0;
char *y = x + 6, *noch_ein_pointer, kein_pointer;
y[2] = 0;
printf("%s-%s\n", y, x); → We-Hallo
```

- Zeiger werden mit einem führendem Stern deklariert
- Bei Mehrfachdeklarationen: genau die Variablen mit führendem Stern sind Pointer
- `+,-,+=,-=, ++,--` funktionieren wie bei Integern
- `p += n` z.B. versetzt p um n Elemente
- Pointer werden immer um ganze Elemente versetzt
- Datentyp bestimmt, um wieviel sich die Speicheradresse ändert

Zeiger (de-)referenzieren

```
float *x;
float array[3] = {1, 2, 3};
x = array + 1;
printf("*x = %f\n", *x); // →*x = 2.000000
float wert = 42;
x = &wert;
printf("*x = %f\n", *x); // →*x = 42.000000
printf("*x = %f\n", *(x + 1)); // undefinierter Zugriff
```

- `*p` gibt den Wert an der Speicherstelle, auf die Pointer `p` zeigt
- `*p` ist äquivalent zu `p[0]`
- `*(p + n)` ist äquivalent zu `p[n]`
- &v gibt einen Zeiger auf die Variable v
- so bekommt `scanf` Zeiger auf die Zielvariablen

Variablen und Zeiger

```
int a, b;
int *ptr1 = &a, *ptr2 = &a;

b = 2; a = b; b = 4;
printf("a=%d b=%d\n", a, b); // →a=2 b=4

*ptr1 = 5; *ptr2 = 3;
printf("*ptr1=%d *ptr2=%d\n", *ptr1, *ptr2);
// →ptr1=3 ptr2=3
```

- Zuweisungen von Variablen in C sind tief, Inhalte werden kopiert
- Entspricht einfachen Datentypen in Python (etwa Zahlen)
- Mit Pointern lassen sich flache Kopien erzeugen, in dem diese auf denselben Speicher zeigen
- Entspricht komplexen Datentypen in Python (etwa Listen)

Zeiger als Funktionsparameter

```
void aendern(int ganz) { ganz = 5; }
int ganz = 42;
aendern(ganz);
printf("%d\n", ganz); // → 42

void wirklich_aendern(int *ganz) { *ganz = 5; }
wirklich_aendern(ganz);
printf("%d\n", ganz); // → 5
```

- In C werden alle Funktionsparameter kopiert (**Call by value**)
- Die Variablen bleiben im aufrufenden Code stets unverändert
- Hintergrund: um Werte zu ändern, müssten deren Speicheradressen bekannt sein
- Abhilfe: Übergabe der Speicheradresse der Variablen in Zeiger (**Call by reference**)
- Bei Zeigern führt das zu Zeigern auf Zeiger (`typ **`) usw.

Arrays als Funktionsparameter

```
void aendern(int array[5]) { array[0] = 5; }

int array[10] = { 42 };
aendern(array);
printf("%d\n", array[0]); // → 5
```

- Arrays verhalten sich auch hier wie Zeiger
- Die Werte des Arrays werden nicht kopiert
- Hintergrund: die Größe von Arrays ist variabel, Speicher für die Kopie müsste aber zur Compilezeit bereitgestellt werden

main – die Hauptroutine

```
int main(int argc, char **argv)
{
  printf("der Programmname ist %s\n", argv[0]);
  for(int i = 1; i < argc; ++i) {
    printf("Argument %d ist %s\n", i, argv[i]);
  }
  return 0;
}
```

- `main` ist die Hauptroutine
- erhält als **`int`** die Anzahl der Argumente
- und als **`char`** `**` die Argumente
- Zeiger auf Zeiger auf Zeichen $\widehat{=}$ Array von Strings
- Rückgabewert geht an die Shell

Dynamische Arrays – malloc und free

```
#include <stdlib.h>
// Array mit Platz fuer 10000 integers
int *vek = (int *)malloc(10000*sizeof(int));
for(int i = 0; i < 10000; ++i) vek[i] = 0;
// Platz verdoppeln
vek = (int *)realloc(vek, 20000*sizeof(int));
for(int i = anzahl; i < 20000; ++i) vek[i] = 0;
free(vek);
```

- Speicherverwaltung für variabel große Bereiche im *Freispeicher*
- `malloc` reserviert Speicher
- `realloc` verändert die Größe eines reservierten Bereichs
- `free` gibt einen Bereich wieder frei

Speicherlecks und andere Fehler

- Wird dauernd Speicher belegt und nicht freigegeben, geht irgendwann der Speicher aus („Speicherleck“):

```
// 100 ints anfordern
int *vek = (int *)malloc(100*sizeof(int));
// nochmal 200 dazu, macht 300 belegt
vek = (int *)malloc(200*sizeof(int));
// aber nur auf die letzten 200 habe ich einen Zeiger
```

- Ein Bereich darf nur einmal freigegeben werden

```
int *vek = (int *)malloc(100*sizeof(int));
free(vek); free(vek); // Fehler!
```

- `free` nur auf von `malloc` erhaltenen Zeiger

```
int *vek = (int *)malloc(100*sizeof(int));
vek += 5; // ok, aber 5 erste Elemente unerreichbar
free(vek); // Fehler, free nur auf den malloc-Zeiger
```

Explizite Typumwandlung

```
int *vek = (int *)malloc(100*sizeof(int));

int nenner = 1, zaehler = 1000;
printf("Ganzzahl: %f\n",
  (float)(nenner/zaehler)); // → 0.000000
printf("Fliesskomma: %f\n",
  ((float) nenner)/zaehler); //→ Fliesskomma: 0.001000
```

- C wandelt Typen nach Möglichkeit automatisch um
- Explizite Umwandlung: geklammerter Typname vor Ausdruck
  Beispiel: `(int)( ((float) a) / b)`
- Notwendig bei Umwandlung
  - `int` ↔ `float`
  - von Zeigern
- insbesondere bei `malloc`, da es `void *` liefert

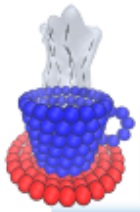
Beispiel: Mergesort

```
void sort(int A[], int len) {
  if (len <= 1) { return; }
  // lengths of sublists to operate on
  int llen = len/2, rlen = len - llen;
  // split into copied sublists
  int *left =  copylist(A, llen);
  int *right = copylist(A + llen, rlen);
  // sort the two sublists
  sort(left,  llen);
  sort(right, rlen);
  // merge them
  mergelists(A, left, llen, right, rlen);
  // and free the intermediate memory
  free(left); free(right);
}
```

Beispiel: Mergesort

The copying function is easy:

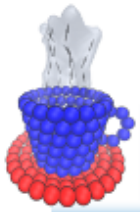
```
#include <stdlib.h>
#include <string.h>

int *copylist(int A[], int len) {
  // allocate temporary heap memory
  int *result =  (int *)malloc(sizeof(int)*len);
  // copy A to the temporary heap memory
  memcpy(result, A, sizeof(int)*len);
  return result;
}
```



Beispiel: Mergesort

Merging is a bit shorter in C:

```
void mergelists(int *A, int *l, int ln, int *r, int rn) {
  // read indexes in left/right lists
  int lind = 0, rind = 0;
  for (int i = 0; i < ln + rn; ++i) {
    /* use from left list if left list is not empty
       and right list is empty or left element is
       smaller then the right element */
    if (lind < ln && (rind >= rn || l[lind] <= r[rind]))
      A[i] = l[lind++];
    else
      A[i] = r[rind++];
  }
}
```



Debugger

- Programmausführung beobachten
- Dazu Programm mit `-g -O0` compilieren
- `gdb <programm>`

Die wichtigsten gdb-Befehle

- `run`: Programm starten
- **`break`**: Anhalten bei Erreichen einer Codezeile/Funktion
- `cont`: Nach Breakpoint weitermachen
- `next`: Eine Zeile ausführen, Unterfunktionen überspringen
- `step`: Einen Schritt ausführen, in Unterfunktionen anhalten
- `up/down`: Navigation auf dem Stack, Funktionsaufrufe
- `print`: Variablen ausgeben

Datenstrukturen: Dynamische Arrays als Listen

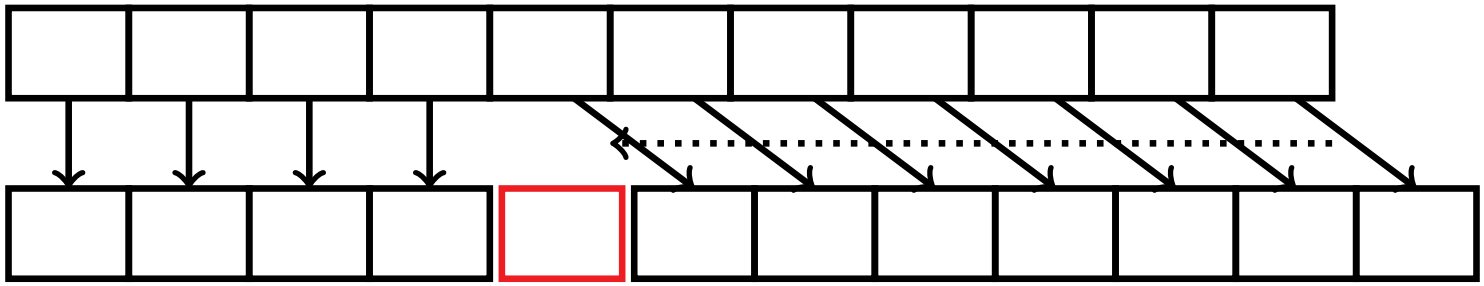
- Zentrale Datentypen in Python: Listen und Wörterbücher
- Wie sind diese implementiert?
- Python-Liste entspricht einem dynamischen C-Array

Dynamisches Array

Elementzugriff	$\mathcal{O}(1)$
Einfügen/Löschen	$\mathcal{O}(N)$
Einfügen/Löschen am Ende	$\mathcal{O}(1)$

- Bei großen Listen wird das Einfügen im allgemeinen langsam
- Geht das auch anders? Ja: **verkettete Listen**

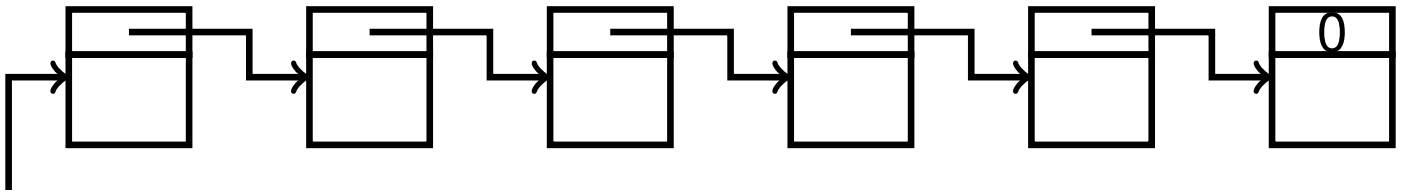
Implementierung des Einfügens



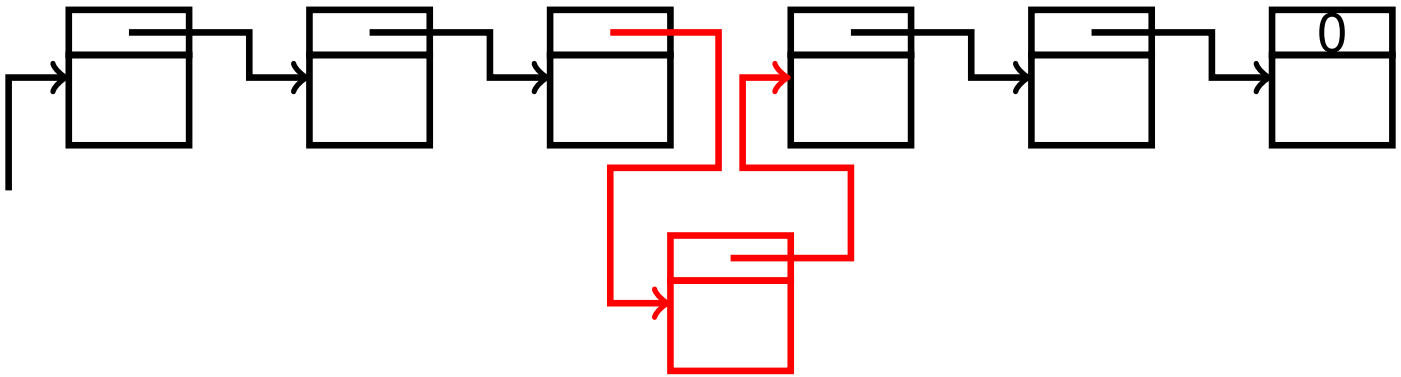
Einfügen von `value` an Position `pos`
des Arrays `list` mit Gesamtlänge `len`:

```
int *insert_array(int value, int pos, int *list, int len) {
  // allocate necessary space
  list = realloc(list, (len+1)*sizeof(int));
  // copy starting at the end to avoid overwriting
  for (int i = len - 1; i >= pos ; --i)
    list[i+1] = list[i];
  list[i] = value;
  return list;
}
```

Verkettete Listen



- Jedes Element hat neben den Daten einen *Zeiger* auf das nächste Element
- Zeiger auf das erste Element stellt die Liste dar
- Das letzte Element hat ungültigen Zeiger (NULL, 0)



struct – Datenverbünde

```
struct Position {
  float x, y, z;
};
struct ListElement {
  struct Position position;
  struct ListElement *next;
};
```

- **`struct`** definiert einen Verbund
- Ein Verbund fasst mehrere Variablen zusammen
- Die Größe von Feldern in Verbünden muss konstant sein
- Ein Verbund kann Verbünde enthalten
- Offensichtlich nur *Zeiger* auf sich selber

Variablen mit einem struct-Typ

```
struct Position pos, *ptr = &pos;

pos.x = 42.0;
ptr->y = 0.3;

struct Position pos1 = { 1, 0, 0 };
struct Position pos2 = { .x = 1, .y = 2, .z = 3 };
```

- Elemente des Verbunds werden durch „.“ angesprochen
- Kurzschreibweise für Zeiger: `(*pointer).x = pointer->x`
- Verbünde können wie Arrays initialisiert werden
- Initialisieren einzelner Elemente mit Punktnotation

typedef

```
typedef float real;
typedef struct Position Position;
typedef struct { real v[3]; } Vektor3D;

struct Position pos;
Particle pos1;        // beides ist ok, selber Typ

Vektor3D vektor; // auch ok
struct Vektor3D vektor; // nicht ok, da struct namenlos
```

- **typedef** definiert neue Namen für Datentypen
- **typedef** ist nützlich, um Datentypen auszutauschen, z.B. double anstatt float
- Achtung: **struct** Particle und Particle können auch verschiedene Typen bezeichnen!
- **typedef struct** {...} typ erzeugt keinen Typ **struct** typ

Structs als Funktionsparameter

```
struct Position { int v[3]; };

void aendern(struct Position p) { p.v[0] = 5; }

struct Position pos = {{ 1, 2, 3 }};
aendern(pos);
printf("%d\n", pos.v[0]); // → 1
```

- Strukturen verhalten sich wie einfache Datentypen
- Wenn diese Arrays enthalten, werden diese kopiert
- In diesem Fall ist die Größe des Arrays im Voraus bekannt

Verkettete Liste

Datenstruktur

```
typedef struct Element {
  struct Element *next;
  int value;
} Element;
```

Die Liste wird durch einen Zeiger auf das erste Element dargestellt

Erzeugen eines neuen Elements

```
Element *makeElement(int value, Element *next) {
  Element *element = (Element *)malloc(sizeof(Element));
  element->value = value; element->next = next;
  return element;
}
```

Einfügen bei einer verketteten Liste

```
Element *insertAfter(int value, Element *pos,
                     Element **listhead) {
  if (pos == NULL) {
    // insert at head
    if (*listhead == NULL) {
      // not a list yet
      return *listhead = makeElement(value, NULL);
    }
    return *listhead = makeElement(value, *listhead);
  }
  // insert inbetween, just update given element
  pos->next = makeElement(value, pos->next);
  return pos->next;
}
```

Iterieren über eine verkettete Liste

Eine Schleife über eine verkettete Liste:

```
Element *search(int value, Element *list) {
  for (Element *element = list; element;
       element = element->next) {
    if (element->value == value)
      return element;
  }
  return NULL;
}
```

- Sucht Wert `value` in der Liste
- Gibt NULL zurück, wenn nicht gefunden

Datenstrukturen: verkettete Listen

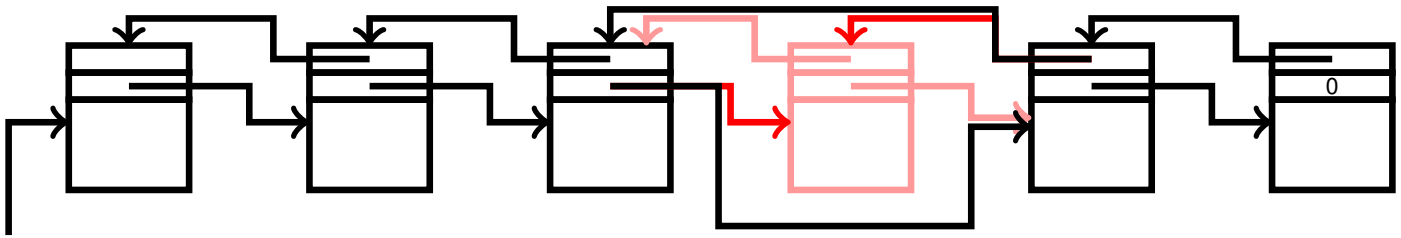
Verkettete Liste

Elementzugriff	$\mathcal{O}(N)$
Einfügen	$\mathcal{O}(1)$
Löschen	$\mathcal{O}(N)$
Löschen am Anfang	$\mathcal{O}(1)$

Doppelt verkettete Listen

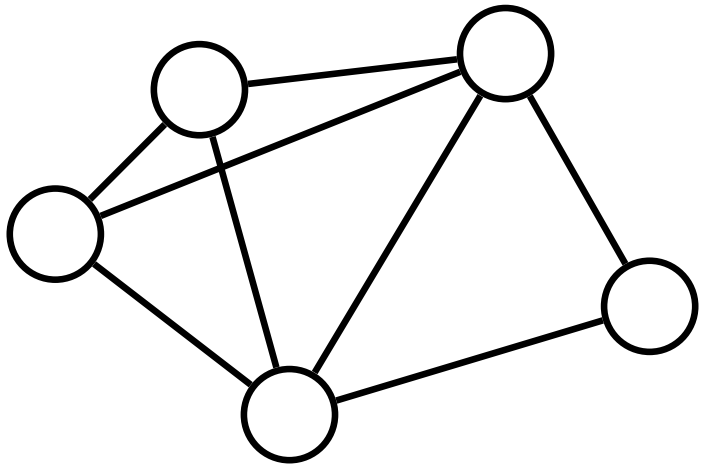
Elementzugriff	$\mathcal{O}(N)$
Einfügen	$\mathcal{O}(1)$
Löschen	$\mathcal{O}(1)$

Doppelt verkettete Listen:



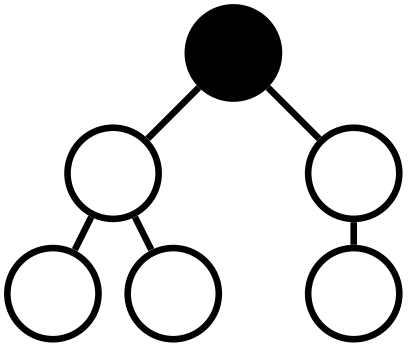
- Schnelleres Löschen durch Rückzeiger: Vorgänger bekannt

Datenstruktur: Graphen



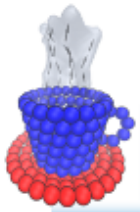
- Abstrakte Datenstruktur aus Ecken und Kanten
- Kanten können gerichtet sein oder Gewichte tragen
- Beschreiben Beziehungen, z. B. durch Straßen/Fluglinien verbundene Städte
- Darstellung: jeder Knoten hat Liste von Zeigern auf seine Nachbarn

Datenstruktur: Bäume



```
struct Node {
  type data;
  struct Node *left, *right;
};
struct Node *root;
```

- Graph ohne Zyklen
- Genau ein ausgezeichneter Knoten (**Wurzel**)
- Alle anderen Knoten sind Kinder von Kindern der Wurzel
- **Blätter** sind Knoten ohne Kinder
- **Binäre** Bäume: maximal zwei Kinder pro Knoten
- Binärer Baum in C:
  - Knoten-**`struct`** enthält Zeiger auf Kinder
  - Zeiger auf die Wurzel
  - NULL-Zeiger, wenn kein weiteres Kind mehr



Wörterbücher

- Wörterbücher enthalten **Schlüssel**-**Wert**-Paare:

  „Axel“ $\rightarrow$1,0
  „Bernd“ $\rightarrow$1,3
  „Christian“ $\rightarrow$1,0

- Beispiel: `dict` in Python
- Wie kann ich das in C *effizient* implementieren?

Implementation mit dynamischen Arrays

- Speichern der Schlüssel-Wert-Paare
- Anfügen erfordert $\mathcal{O}(1)$ Operationen ☺
- Elementzugriff erfordert Suche
  $\Longrightarrow$ Schleife über alle Paare
  $\Longrightarrow$ $\mathcal{O}(N)$ Operationen ☹

Schnellere Suche: Bisektion

- Speichere Schlüssel-Wert-Paare *sortiert* nach Schlüssel
- Vergleiche Wörterbuch, Telefonbuch, ...
- Bisektion:
  Betrachte Pivotelement in der Arraymitte
  1. Element ist der gesucht Schlüssel $\Longrightarrow$ gefunden!
  2. Element ist kleiner $\Longrightarrow$ durchsuche größere Elemente (rechts)
  3. Element ist größer $\Longrightarrow$ durchsuche kleinere Elemente (links)

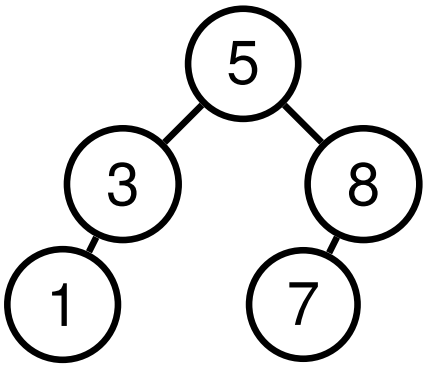
  Halbiert Listenlänge in jedem Schritt $\Longrightarrow$ $\mathcal{O}(\log N)$ Schritte ☺

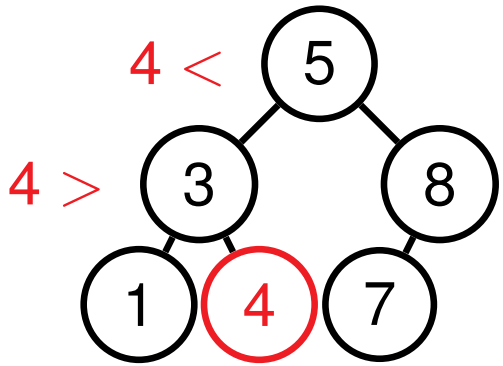
  Beispiel

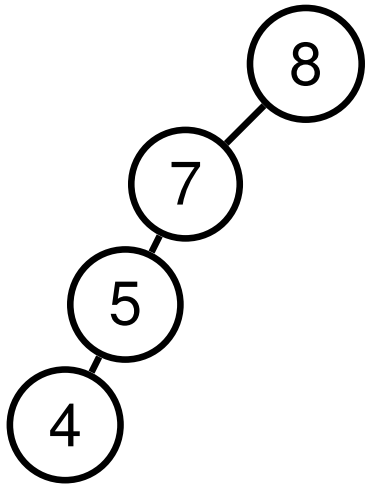
1	3	4	5	7	8	$4 < 5 \Longrightarrow$ links weitersuchen
1	3	4	5	7	8	$4 > 3 \Longrightarrow$ rechts weitersuchen
1	3	4	5	7	8	gefunden!

- Aber sortiertes Einfügen i. a. $\mathcal{O}(N)$ ☹

Binäre Suchbäume

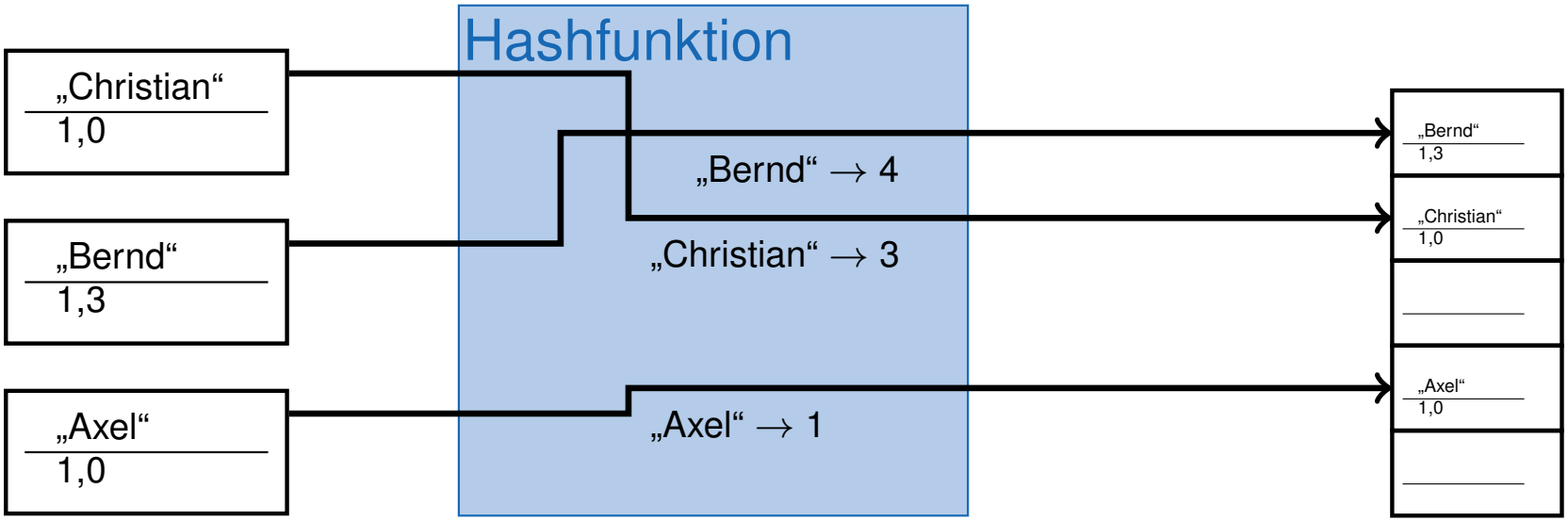






- Linkes Kind ist kleiner als der Elter, rechtes Kind größer
- Werte werden stets als Blätter hinzugefügt
- Einfügen und Suchen besuchen immer nur eins von zwei Kindern
  $\Longrightarrow$ erfordern $\mathcal{O}(\log N)$ Operationen ☺
- Aber: im ungünstigsten Fall ist der Baum eine verkettete Liste! ☹
  (wenn Elemente sortiert einfügt werden)
- Abhilfe: **Balancierte** Bäume, z. B. Red-Black-Bäume

Datenstruktur: Hashtables



- Python benutzt eine **Hashtable** für das Wörterbuch
- Weitere Anwendungen in Datenbankindexing, Cacheing,...
- Die wahrscheinlich wichtigste bekannte Datenstruktur (S. Yegge)
- Dynamisches Array von Schlüssel-Wert-Paaren
- Position wird aus Schlüssel durch **Hashfunktion** bestimmt
- Kein Umsortieren beim Einsortieren $\Longrightarrow \mathcal{O}(1)$ ☺
- Auslesen wie aus dynamischem Array $\Longrightarrow \mathcal{O}(1)$ ☺

Hashfunktionen

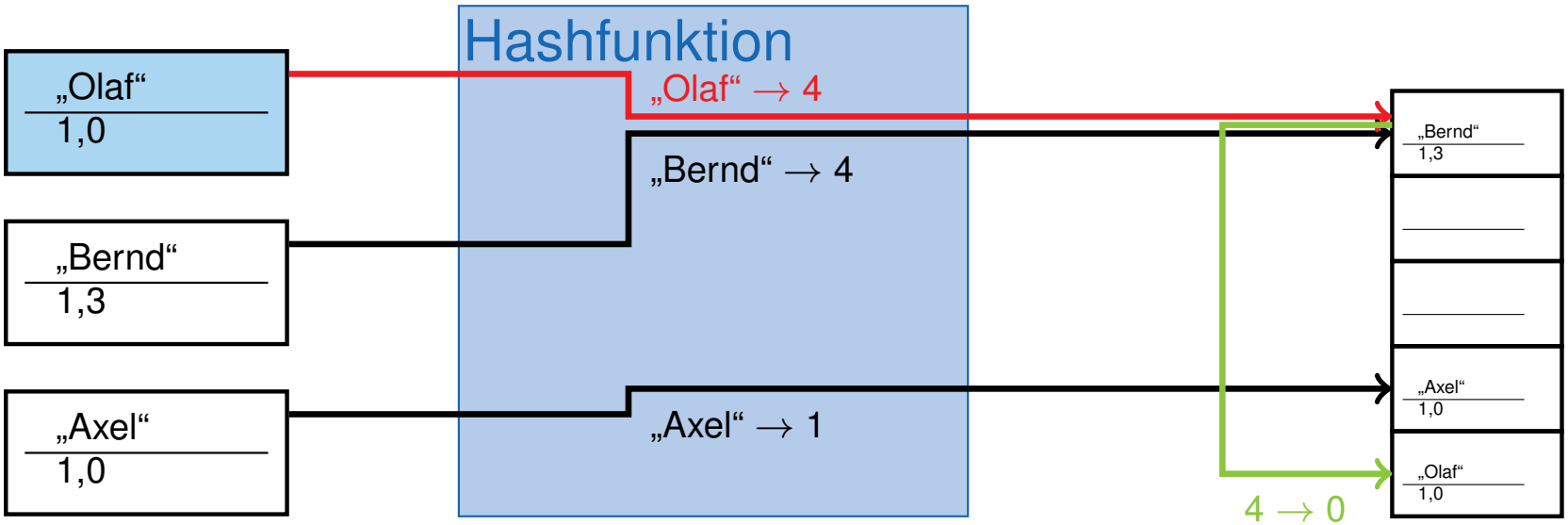
- Schlüssel möglichst gleichmässig über das Array verteilen
- Kein Muster sollte zu Häufungen / Lücken führen
  (z. B. Zeichenketten nur aus Großbuchstaben oder Zahlen)
- Indizes müssen in das Array passen, daher meist:
  Zeichenkette $\rightarrow$ Ganzzahl modulo Array-Größe

Beispiel Zeichenketten (Dan Bernstein)

```
unsigned int hash(unsigned char *str) {
  unsigned int hash = 5381;
  for (int i = 0; str[i] != 0; ++i)
    hash = 33*hash + c;
  return hash;
}
```

- **Kryptographische Hashes**: Rekonstruktion äußerst schwer, zur Verifikation von Daten

Kollisionsbehandlung



- Verschiedene Schlüssel haben gleichen Hash $\Longrightarrow$ **Kollision**
- Immer möglich, da Arraygröße beschränkt
- Um die Chance klein zu halten, mindestens 20% Slots leer lassen
- Mögliche Lösungen:
  - (Verkettete) Listen für jeden Schlüssel
  - Bei Kollision Hashwert ändern, z. B. um eins erhöhen

Pythons Hashfunktion

- Ganzzahlen sind ihr eigener Hash (außer -1, das Hash -2 hat)
- Für Strings sieht die Hashfunktion ungefähr so aus:

```
int string_hash(const char *p)
{
    long x; // hash to compute
    x = p[0] * 128;
    int i;
    for (i = 0; p[i] != 0; ++i)
        x = (1000003*x) ^ p[i];
    x ^= i; // xor length of the string
    if (x == -1) x = -2;
    return x;
}
```

http://svn.python.org/projects/python/trunk/Objects/stringobject.c

Pythons Kollisionsbehandlung

- Python passt die Hashwerte an, und zwar ungefähr so:

```
find_key(const char *key, KeyValuePair *slot, int mask) {
  int i = string_hash(key);
  for (long perturb = hash; ; perturb >>= 5) {

    i = 5*i + perturb + 1 & mask;

    if (is_empty(slot[i].key)
      return NULL;    // not in the hash table
    else if (strcmp(slot[i].key, key) == 0)
      return slot[i]; // found
    // if we get here, we had a collision, one more round
  }
}
```

http://svn.python.org/projects/python/trunk/Objects/dictobject.c

- Bei ca. 2/3 Füllstand wird die Tabellengröße verdoppelt