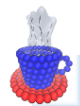


# Computergrundlagen Programmieren in C

**Axel Arnold**

Institut für Computerphysik  
Universität Stuttgart

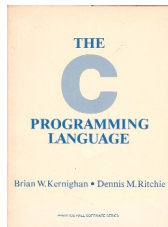
Wintersemester 2014/15



## 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



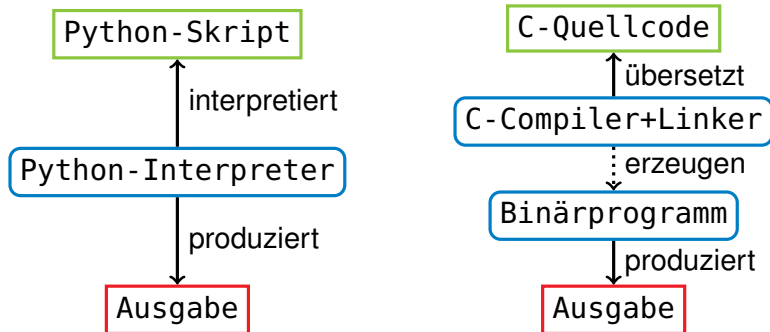
## Hello, World!

```
#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

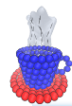


## Interpreter- vs. Compilersprachen

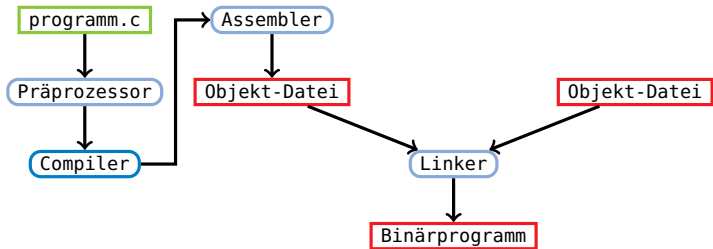


- **Interpreter** liest Programm bei Ausführung und führt Anweisungen aus
- Interpreter zur Ausführung nötig

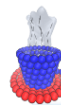
- **Compiler** übersetzt Programm in Maschinsprache
- Programm läuft ohne Compiler, aber nur auf Zielhardware + Betriebssystem



## 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



## Assembler und Maschinensprache

- Maschinensprache sind Zahlenkolonnen

---

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

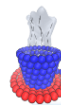
---

- Assembler ist die menschenlesbare Form:

# 6502 Maschinensprache	# Assembler
	X = \$1234
AD 34 12	<b>LDA X</b>
18	<b>CLC</b>
69 2A	<b>ADC #42</b>
8D 34 12	<b>STA X</b>
AD 35 12	<b>LDA X+1</b>
69 00	<b>ADC #00</b>
8d 35 12	<b>STA X+1</b>

---

- Meist von Compiler erzeugt
- Zeitkritische Anwendungen werden manchmal direkt in Assembler geschrieben



## 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

## 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
- `where`: Stack anzeigen
- `up/down`: Navigation auf dem Stack, Funktionsaufrufe
- `print`: Variablen ausgeben





## 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;
}
```

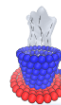
---



## 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 Rückgabetyt **void**, gibt die Funktion nichts zurück
- **return** verlässt wie in Python eine Funktion vorzeitig
- Hauptprogramm main ist auch Funktion

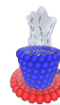


## Datentypen in C

- anders als in Python sind Variablen fest typisiert
- Grunddatentypen

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

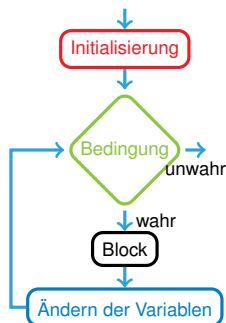
### ■ Wiederholungsbedingung

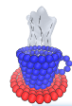
Schleifenende, wenn unwahr

### ■ 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
```

---

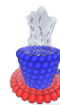


## Beispiel: Pythagoräische Zahlentripel

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))`

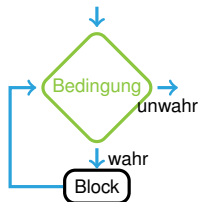


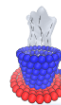
## Schleifen – while

```
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;  
}
```

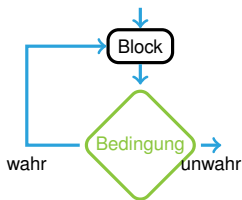




## Schleifen – do ... while

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

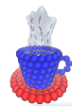
- **do** block **while** (cond);  
führt block aus, solange die Bedingung cond wahr ist
- Unterschied zur **while**-Schleife:  
**do ... while** überprüft nach dem Block



- 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  $\implies$  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

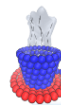
```
#include <stdio.h>
int main() {
    const int N = 100;
    // initially, assume all numbers > 1 are prime
    int is_prime[N];
    for (int i = 2; i < N; ++i) is_prime[i] = 1;
    // now remove true multiples
    for (int i = 2; i < N/2; ++i) {
        if (!is_prime[i]) continue; // multiples already 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; // kompiliert, 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 ändert 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 benannte 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);  
    strcat(joined, string2, len - strlen(joined) - 1);  
    printf("%s = %s%s\n", joined, string1, string2);  
}
```

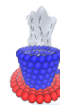




## Zeichenketten

```
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  
⇒ ein String ist immer eins länger, als er Buchstaben hat!
- 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 (strcmp(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!
- `strcmp(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 initialisieren 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 fluess;
int ganz;
if (scanf("%10s %f %d", kette, &fluess, &ganz) == 3) {
    printf("%s %10.3f %d\n", kette, fluess, ganz);
}
```

- Lesen und Schreiben von Standardin-/ausgabe
- Headerdatei **stdio.h** einbinden
- Ganzzahlen **%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
```

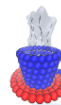
- Zeiger werden mit einem führendem Stern deklariert
- Bei Mehrfachdeklarationen: genau die Variablen mit führendem Stern sind Pointer
- +, -, +=, -=, ++, -- funktionieren wie bei Integern
- p += n z.B. versetzt p um n Elemente
- Pointer werden immer um ganze Elemente versetzt
- Datentyp bestimmt, um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

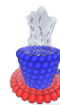


## Variablen und Zeiger

```
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
```

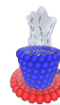
- Arrays verhalten sich auch hier wie Zeiger
- Die Werte des Arrays werden nicht kopiert
- Hintergrund: die Größe von Arrays ist variabel, Speicher für die Kopie müsste aber zur Compilezeit bereitgestellt werden



## 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;
}
```

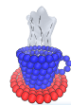
- main ist die Hauptroutine
- erhält als **int** die Anzahl der Argumente
- und als **char \*\*** die Argumente
- Zeiger auf Zeiger auf Zeichen  $\hat{=}$  Array von Strings
- Rückgabewert geht an die Shell



## 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;
}
```

---

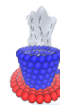




## Beispiel: Mergesort

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++];  
    }  
}
```



## Datenstrukturen: Dynamische Arrays als Listen

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

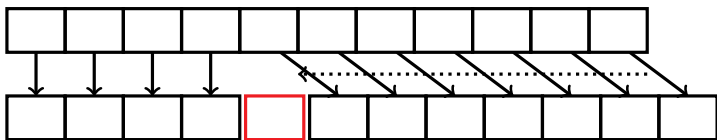
### Dynamisches Array

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

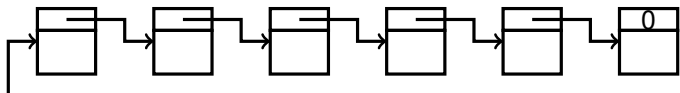


Einfügen von value an Position pos  
des Arrays list mit Gesamtlänge len:

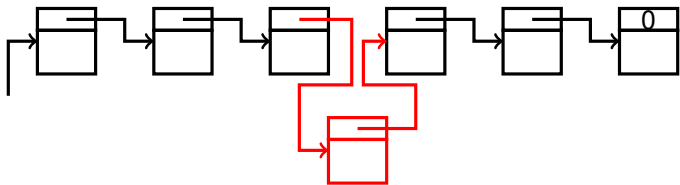
```
int *insert_array(int value, int pos, int *list, int len) {  
    // allocate necessary space  
    list = realloc(list, (len+1)*sizeof(int));  
    // copy starting at the end to avoid overwriting  
    for (int i = len - 1; i >= pos ; --i)  
        list[i+1] = list[i];  
    list[i] = value; 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

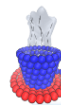


## Verkettete Listen



- 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)
- Einfügen geht durch lokale Zeigeränderungen:





## struct – Datenverbände

```
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 Verbänden muss konstant sein
- Ein Verbund kann Verbände enthalten
- Offensichtlich nur *Zeiger* auf sich selber



## Variablen mit einem struct-Typ

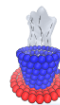
```
struct Position pos, *ptr = &pos;
```

```
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 Particle;  
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;
```

---

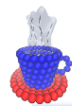
Die Liste wird durch einen Zeiger auf das erste Element dargestellt

### Erzeugen eines neuen Elements

---

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

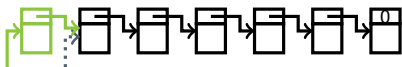
---



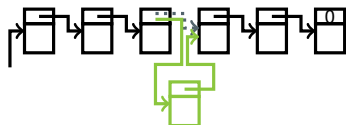
## Einfügen bei einer verketteten Liste

```
Element *insert_after(int value, Element *pos,  
                    Element **listhead) {  
    if (pos == NULL) {  
        // insert at head  
        return *listhead = makeElement(value, *listhead);  
    }  
    // insert inbetween, just update given element  
    pos->next = makeElement(value, pos->next);  
    return pos->next;  
}
```

### Einfügen am Listenanfang



### Einfügen in der Listennitte





## Iterieren über eine verkettete Liste

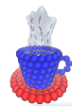
Eine Schleife über eine verkettete Liste:

---

```
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

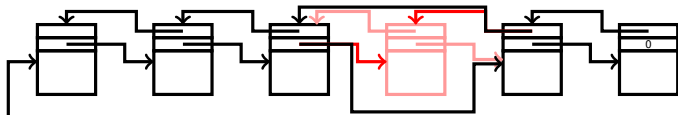
## Verkettete Liste

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

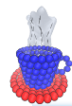
## Doppelt verkettete Listen

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

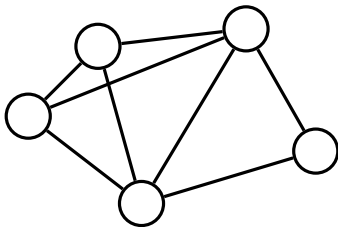
Doppelt verkettete Listen:



- Schnelleres Löschen durch Rückzeiger: Vorgänger bekannt



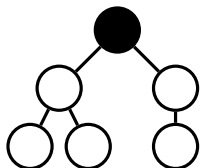
## Datenstruktur: Graphen



- 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

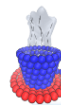


---

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

---

- Graph ohne Zyklen
- Genau ein ausgezeichnetener 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“ → 1,0

„Bernd“ → 1,3

„Christian“ → 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
  - ⇒ Schleife über alle Paare
  - ⇒  $\mathcal{O}(N)$  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  $\implies$  gefunden!
2. Element ist kleiner  $\implies$  durchsuche größere Elemente (rechts)
3. Element ist größer  $\implies$  durchsuche kleinere Elemente (links)

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

### Beispiel

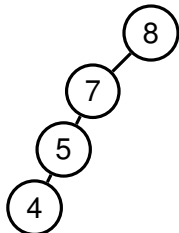
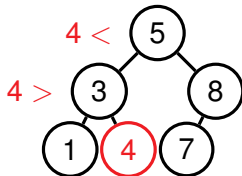
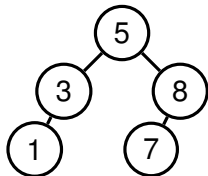
1	3	4	5	7	8	$4 < 5 \implies$ links weitersuchen
1	3	4	5	7	8	$4 > 3 \implies$ rechts weitersuchen
1	3	4	5	7	8	gefunden!

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



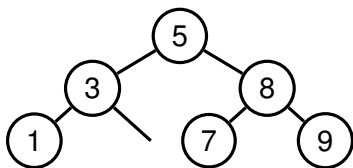


## Binäre Suchbäume



- 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  
⇒ erfordern  $\mathcal{O}(\log N)$  Operationen 😊
- Aber: im ungünstigsten Fall ist der Baum eine verkettete Liste! 😞  
(wenn Elemente sortiert eingefügt werden)
- Abhilfe: **Balancierte** Bäume, z. B. Red-Black-Bäume

## Sortieren mit Suchbäumen



- Sortiere Teilchen wieder so, dass überall die kleineren Schlüssel im linken Ast sind, die größeren im rechten
- Die aufsteigend sortierte Liste ergibt sich rekursiv durch **symmetrische Traversierung**:

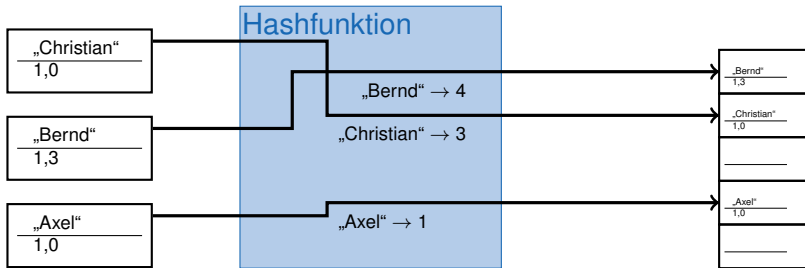
---

```
def print_tree_sorted(tree):  
    if tree.left: print_tree_sorted(tree.left)  
    print tree.key  
    if tree.right: print_tree_sorted(tree.right)
```

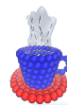
---

- Rechter Baum wird zuerst besucht  $\implies$  Liste absteigend sortiert

## 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  $\implies \mathcal{O}(1)$  😊
- Auslesen wie aus dynamischem Array  $\implies \mathcal{O}(1)$  😊



## 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 → Ganzzahl modulo Array-Größe

### Beispiel Zeichenketten (Dan Bernstein)

---

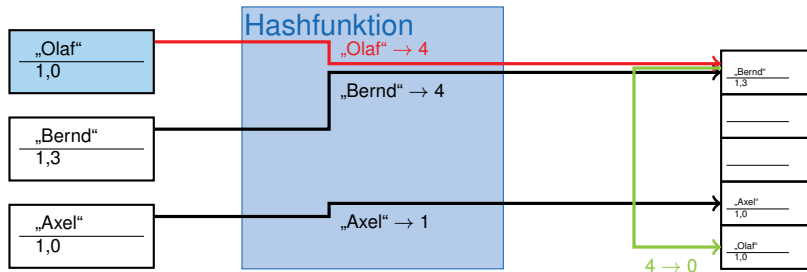
```
unsigned int hash(unsigned char *str) {  
    unsigned int hash = 5381;  
    for (int i = 0; str[i] != 0; ++i)  
        hash = 33*hash + str[i];  
    return hash;  
}
```

---

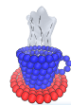
- **Kryptographische Hashes:** Rekonstruktion äußerst schwer, zur Verifikation von Daten



## Kollisionsbehandlung



- Verschiedene Schlüssel haben gleichen Hash  $\implies$  **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

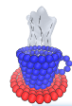
- Ganzzahlen sind ihr eigener Hash (außer -1, das Hash -2 hat)
- Für Strings sieht die Hashfunktion ungefähr so aus:

---

```
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

## Beispiel: Datenanalyse

**Gegeben:** Laaange Zahlenkolonne in einer Datei

**Gesucht:** Mittelwert

$$M = \frac{1}{N} \sum_{i=1}^N x_i$$

und Standardabweichung

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2 - M^2}$$

der Daten

oder

ein Histogramm mit variable Größe





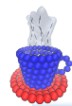
## Implementation: Mittelwert und Standardabweichung

```
#include <math.h>
/* compute first and second moment (mean and variance) */
void moments(const double *data, int len,
             double *mean, double *stddev)
{
    double _m1 = 0, _m2 = 0;
    for (int i = 0; i < len; ++i) {
        _m1 += data[i];
        _m2 += data[i]*data[i];
    }
    *mean = _m1/len;
    *stddev = sqrt(_m2/len - (*mean)*(*mean));
}
```



## Implementation: Lesen aus der Datei

```
/* read a file of double data into an array */  
int read_data(const char *argv, double **data, int *len) {  
    FILE *file = fopen(argv, "r");  
    if (!file) { return -1; }  
    // initialize empty data list  
    double *_data = NULL; int _len = 0;  
    while (!feof(file)) {  
        _len++;  
        _data = (double *)realloc(_data, _len*sizeof(double));  
        fscanf(file, "%lg", &_data[_len - 1]);  
    }  
    *data = _data; *len = _len;  
    return 0;  
}
```



## Implementation: Parsen der Kommandozeile

Wir nutzen die getopt-Bibliothek:

```
// histogram parameters with defaults
int bins = 20; double min = 0, max = 1;
int task = MOMENTS;
// use getopt to parse the commandline
while (getopt(argc, argv, "hb:m:M:") != -1) {
    switch (optopt) {
        case 'h': task = HISTOGRAM; break;
        case 'b': bins = atoi(optarg); break;
        case 'm': min = atof(optarg); break;
        case 'M': max = atof(optarg); break;
        default: return usage();
    }
}
// skip parsed options
argc -= optind; argv += optind;
```



## Bedingte Ausführung – switch

```
char buch = argv[1][0];  
switch (buch) {  
case 'a':  
    printf("a, rutscht durch zu b\n");  
    // fall through!  
case 'b':  
    printf("b, hier geht es nicht weiter\n");  
    break;  
default:  
    printf("Buchstabe '%c' ist unerwartet \n", buch);  
}
```

- Das Argument von **switch** (wert) muss ganzzahlig sein
- Die Ausführung geht bei **case** konst: weiter, wenn wert=konst
- **default**: wird angesprungen, wenn kein Wert passt
- Abarbeitung stoppt nicht an **case**, durch **break** verlassen



## Bibliotheksfunktionen

- In C sind viele Funktionen in Bibliotheken realisiert
- Diese sind selber in C / Assembler geschrieben
- Basisfunktionen sind Teil der C-Standardbibliothek
- Andere Bibliotheken müssen mit `-l` geladen werden, z.B.  
`gcc -Wall -O3 -std=c99 -o mathe mathe.c -lm`  
zum Laden der Mathematik-Bibliothek „libm“
- Um die Funktionen benutzen zu können, sind außerdem Headerdateien notwendig



## math.h – mathematische Funktionen

```
#include <math.h>
```

```
float pi = 2*asin(1);
```

```
for (float x = 0; x < 2*pi; x += 0.01) {  
    printf("%f %f\n", x, pow(sin(x), 3)); // x, sin(x)^3  
}
```

- math.h stellt mathematische Standardoperationen zur Verfügung
- Bibliothek einbinden mit  
gcc -Wall -O3 -std=c99 -o mathe mathe.c -lm
- Beispiel erstellt Tabelle mit Werten  $x$  und  $\sin(x)^3$



## stdio.h – Datei-Ein-/Ausgabe

```
#include <stdio.h>
FILE *datei = fopen("test.txt", "r");
if (datei == 0) {
    fprintf(stderr, "Kann Datei nicht oeffnen\n");
    return -1;
}
if (fscanf(datei, "%f %d", &fliess, &ganz) == 2) {
    fprintf(stdout, "%f %d\n", fliess, ganz);
}
fclose(datei);
```

- stdin, stdout und stderr sind **Handles** für die Standardgeräte
- Dateien öffnen mit fopen, schließen mit fclose
- Handle ist NULL, wenn Datei nicht geöffnet werden kann
- Zum Schreiben öffnen mit Modus „w“ oder „a“ statt „r“
- fprintf und fscanf funktionieren wie printf und scanf



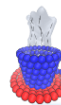
## stdio.h – sscanf

```
#include <stdio.h>

int main(int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, "Kein Parameter gegeben\n");
        return -1;
    }
    if (sscanf(argv[1], "%f", &fliess) == 1) {
        fprintf(stdout, "%f\n", fliess);
    }
    return 0;
}
```

- sscanf liest statt aus einer Datei aus einem 0-terminierten String
- Dies ist nützlich, um Argumente umzuwandeln
- Es gibt auch sprintf zum Schreiben in Strings

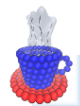




## getopt.h – Kommandozeile parsen

```
#include <unistd.h>
int main(int argc, char **argv) {
    int bins = 20;
    while (getopt(argc, argv, "hb:") != -1) {
        switch (optopt) {
            case 'b': bins = atoi(optarg); break;
            case 'h':
                default: return usage();
        }
    }
    argc -= optind; argv += optind;
}
```

- getopt: Parsen einbuchstabiger Optionen
- Doppelpunkt hinter Option: diese hat ein Argument
- optind: Index des aktuellen Arguments in argc
- optarg: Argument, falls Doppelpunkt (String)
- optopt: Buchstabe der aktuellen Option



## Modularisierung



Einteilen eines Programms in mehrere Quelldateien

- Übersichtlicherer Code
- Wiederverwendbarkeit von Teilen des Codes
- Schnelleres Neucompilieren bei kleinen Codeänderungen



## Modularisierung in C

### Headerdateien

- haben üblicherweise Endung „.h“
- beschreiben, welche Funktionen und Variablen eine Objektdatei/Bibliothek bereitstellt (**exportiert**)
- beschreiben **Signatur** der Funktionen (Typ von Parametern)
- beschreiben den Typ von exportierten Variablen

### Objektdateien

- haben üblicherweise Endung „.o“
- werden wie Programme aus C-Quellcode erzeugt
- oder aus Fortran, Assembler,...-Code
- enthalten Programmcode und/oder globale Variablen
- **Bibliotheken** sind Sammlungen von Objektdateien



## extern und static

```
// nur fuer die aktuelle Objektdatei
static char *string1;
// fuer alle sichtbar
    char *string2;
// kommt woanders her
extern char *string3;
```

- *Globale* Variablen können in mehreren Objektdateien verwendet werden
- **static**
  - Nur in dieser Objektdatei, nach außen unsichtbar
  - Für private globale Variablen von Bibliotheken
- **extern**
  - Diese Variable *muss* in genau einer anderen Objektdatei liegen
  - Sonst gibt es „undefined reference“-Fehler



## Aufbau von Headerdateien

modul.h

---

```
#ifndef MODUL_H
#define MODUL_H
// Funktionsdeklaration
const char *funktion();
// Variablendeklaration
extern const char *var;
#endif
```

---

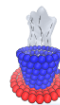
- Der **#ifdef/#define**-Konstrukt sichert, dass die Headerdatei modul.h nur einmal pro Objektdatei eingebunden wird
- Variablen werden hier **extern** deklariert
- Funktionen werden einfach ohne Funktionskörper deklariert
- Variablen und Funktionen müssen in genau einer Objektdatei definiert werden



## #define – Makros

```
#define PI 3.14
/* Position eines Teilchens
   x sollte Zeiger auf Particle sein */
#define POSITION(x) ((x)->position)
POSITION(part).z = PI;
#undef PI
float test = PI; // Fehler, PI undefiniert
```

- **#define** definiert Makros
- **#undef** entfernt Definition wieder
- Ist Präprozessorbefehl, d.h. Makros werden *textuell* ersetzt
- Ohne Parameter wie Konstanten einsetzbar
- Makros mit Parametern nur sparsam einsetzen!
- Klammern vermeiden unerwartete Ergebnisse



## #ifdef – bedingte Compilierung

```
#define USE_STDIO_H
```

```
#ifdef USE_STDIO_H  
#include <stdio.h>  
#endif
```

- **#ifdef** MAKRO bindet Quellcode nur ein, wenn das Makro MAKRO definiert ist
- **#ifndef** MAKRO bindet Quellcode nur ein, wenn das Makro MAKRO *nicht* definiert ist
- Code für verschiedene Umgebungen, Bibliotheken, ... anpassen
- Abschalten nicht immer benötigten Codes
- Makros können auch auf der Kommandozeile definiert werden:  
gcc -Wall -O3 -std=c99 -**DUSE\_STDIO\_H** -o test test.c
- Meist benutzt, um Headerdateien nicht doppelt einzubinden



## Aufbau der Quelldateien

modul.c

---

```
#include "modul.h"
```

```
// die exportierte Funktion
```

```
const char *funktion() { return "Hello"; }
```

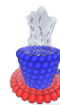
```
// und die exportierte Variable
```

```
const char *var = "World";
```

---

- Normaler C-Code wie gehabt, aber i.A. ohne main
- main muss in genauer einer Objektdatei definiert sein
- Üblicherweise bindet man die zugehörige Headerdatei ein
- Dadurch fallen Unterschiede in Deklaration und Definition schneller auf (z.B. Änderung der Parameter einer Funktion)





## Compilieren und Linken

```
# Das Modul modul.o aus modul.c erzeugen
```

```
gcc -Wall -O3 -std=c99 -c modul.c
```

```
# und main.o aus main.c
```

```
gcc -Wall -O3 -std=c99 -c main.c
```

```
# programm aus main und modul zusammenbauen
```

```
gcc -o programm main.o modul.o
```

- C-Quellcode wird mit der Option „-c“ in Objektdatei übersetzt
- Der Compiler verbindet mehrere Objektdateien zu einem Programm (**linken**)
- Tatsächlich ruft er dazu den Linker (ld) auf
- make kann benutzt werden, um den Vorgang zu automatisieren



## Bibliotheken

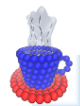
- Statische Bibliotheken sind Archive von Objektdateien
- Erzeugen eines Archivs:  
`ar rcs libbib.a bib_hello.o bib_world.o`
- Linken mit allen Objektdateien in der Bibliothek:  
`gcc main.c -L. -lbib`
- Bei Angabe von „-lbib“ lädt der Compiler automatisch `libbib.a`
- Mit „-L“ werden Suchpfade für Bibliotheken angegeben  
(„-L.“ = aktuelles Verzeichnis)
- Mit `nm` lassen sich die definierten Symbole auslesen:

```
nm libbib.a
```

---

```
bib_hello.o:  
000000000000000000 T get_hello  
000000000000000000 d hello
```

```
bib_world.o:  
000000000000000000 D world
```



## make – automatisiertes Bauen

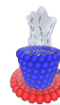
### Makefile

---

```
default: bib
libbib.a: bib_hello.o bib_world.o
        ar rcs $@ $^
bib: main.o libbib.a
        gcc -o bib main.o -L. -lbib
clean:
        rm -f bib libbib.a *.o
```

---

- Die Regeldatei muss Makefile oder makefile heißen
- besteht aus Regeln der Form  
ziel: quelle1 quelle2 ...  
Shell-Befehl zum Erzeugen von ziel
- make ziel baut ziel nur neu, wenn eine Quelle neuer ist
- Automatische Regeln für Objektdateien aus C-Quellcode



## Pakete selber compilieren

- Es gibt im WWW eine riesige Menge an Programmpaketen
- Meist gibt es Installer bzw. fertige Pakete für die eigene Distribution

Selber bauen ist aber nötig, wenn

- man keine Root-Rechte hat und der Admin ein Paket nicht installieren will (z.B. auf Großrechnern)
- es kein fertiges Paket für die Distribution gibt
- das fertige Paket ein Feature noch nicht bietet / einen Bug hat
- man an dem Paket weiterentwickeln will

Das geht natürlich nur mit Open-Source-Paketen, wenn ich nicht gerade den Autor kenne!



## Linux-Dreisprung

- Die meisten Open-Source-Pakete lassen sich mit drei Befehlen compilieren:
- 

```
./configure
```

```
make
```

```
make install
```

---

- Davor muss das Paket natürlich ausgepackt werden:

```
tar -xzf paket-1.2.3.tgz!      oder
```

```
tar -xjf paket-1.2.3.tar.bz2!
```

- erzeugt normalerweise ein Unterverzeichnis paket - 1.2.3
- Skript configure konfiguriert das Paket zum Bauen, erzeugt ein Makefile
- Wird vom Paket-Maintainer mit Hilfe der GNU-autotools erzeugt

## configure

```
./configure --prefix=WOHIN \  
  --with-OPTION --without-OPTION \  
  CPPFLAGS="-I$HOME/include" \  
  LDFLAGS="-L$HOME/include"
```

configure-Optionen können variieren, es gibt (fast) immer:

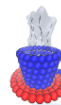
- `--help`: beschreibt alle Konfigurationsmöglichkeiten
- `--prefix`: wohin das Paket installiert werden soll.  
Hier einen Platz im eigenen Home-Directory angeben, sonst kann nur root das Paket installieren
- `--with-OPTION`: schaltet Optionen an (z.B. GUI, Unterstützung für double,...)
- `--without-OPTION`: schaltet Optionen aus

## configure

```
./configure --prefix=WOHIN \  
  --with-OPTION --without-OPTION \  
  CPPFLAGS="-I$HOME/include" \  
  LDFLAGS="-L$HOME/include"
```

configure-Optionen können variieren, es gibt (fast) immer:

- **CPPFLAGS:** Flags für den C-Präprozessor.  
Hier kann man weitere Verzeichnisse mit Headerdateien angeben  
(CPPFLAGS="-I/my/include -I\$HOME/include")
- **LDFLAGS:** Flags für den Linker.  
Hier kann man weitere Verzeichnisse mit Bibliotheken angeben  
(LDFLAGS="-L/my/lib -L\$HOME/lib")
- Das ist insbesondere bei selbstcompilierten Paketen notwendig
- Viele Pakete installieren ein Programm paket-config, das über die benötigten Pfade Auskunft gibt



## Linux-Dreisprung: make

- Das Paket wird dann einfach mit make gebaut
- Je nach Komplexität kann das auch mal Stunden dauern...
- Tipp:  
`make -j N`  
compiliert N Objektdateien gleichzeitig
- Ist der Computer nicht anderweitig belastet, ist es am schnellsten, wenn N der doppelten Anzahl der Prozessor-Kerne entspricht
- Auf einem Quadcore also `make -j 8`
- `make install` installiert das compilierte Paket
  - Ausführbare Dateien landen in `PREFIX/bin`
  - Bibliotheken in `PREFIX/lib`
  - Headerdateien in `PREFIX/include`
  - Hilfedateien in `PREFIX/share/man/man*`





## Beispiele nützlicher Bibliotheken für C

- **GSL – GNU Scientific Library**  
<http://www.gnu.org/software/gsl>
  - viele Zufallszahlengeneratoren
  - Spezielle Funktionen (Bessel-, Gammafunktion, ...)
  - Auswertetools (Statistik, Histogramme, ...)
- **FFTW – Fastest Fourier Transform in the West**  
<http://www.fftw.org>
- **GTK für graphische Interfaces à la GIMP, GNOME, XFCE, ...**  
<http://www.gtk.org>
- **Open MPI, eine MPI-Implementation für viele Plattformen**  
<http://www.open-mpi.org>