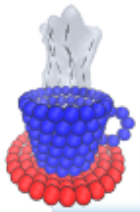


Computergrundlagen Programmieren in C

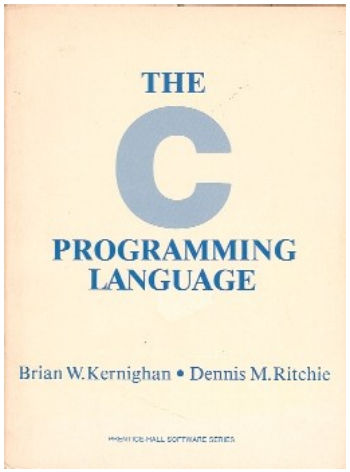
Axel Arnold

Institut für Computerphysik
Universität Stuttgart

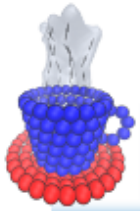
Wintersemester 2013/14

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!

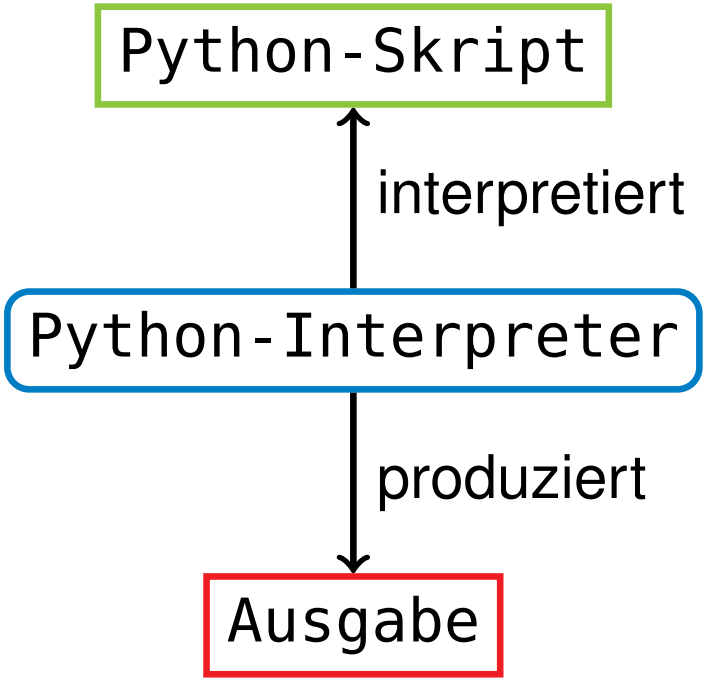
```c
#include <stdio.h>
// main program
int main()
{
  printf("Hello, World\n");
  return 0;
}
```

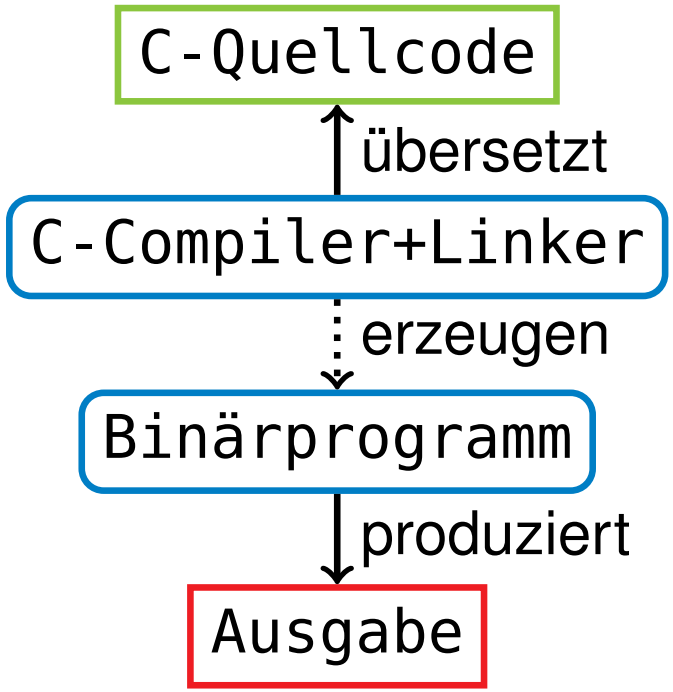
- C-**Quellcode** muss als Textdatei vorliegen (z.B. `helloworld.c`)
- Formatierung freier als in Python, siehe auch `http://www.ioccc.org/main.html`
- Vor der Ausführung mit dem GNU-Compiler **compilieren**:

```
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

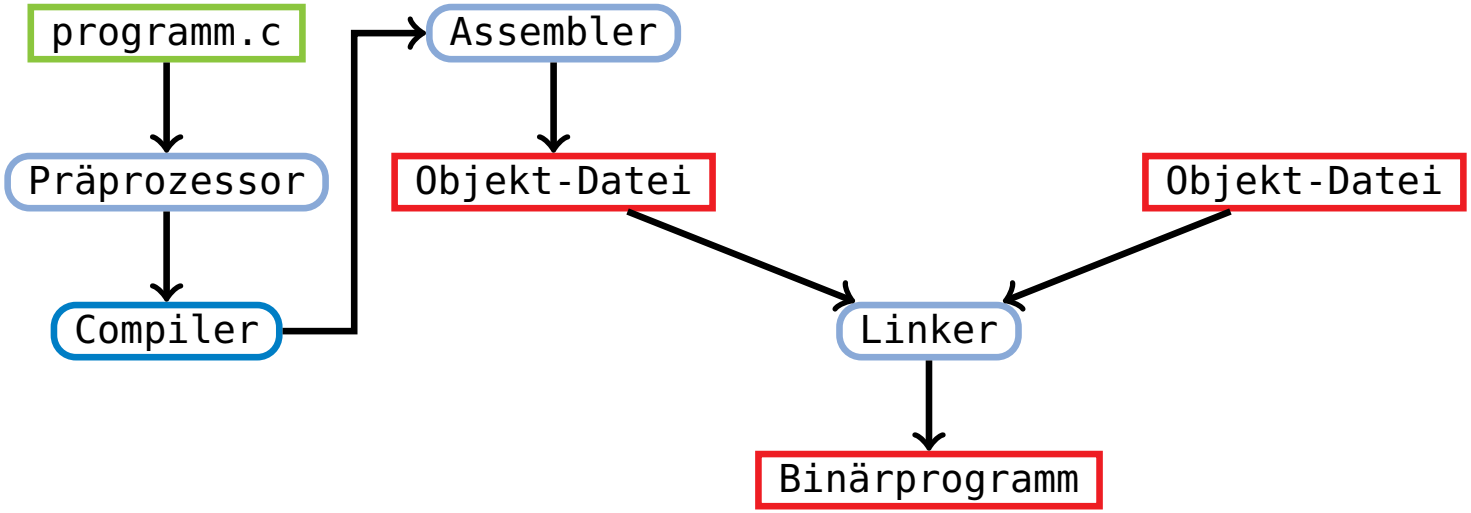
```
AD 34 12 18 69 2A 8D 34 12 AD 35 12 69 00 8d 35 12
```

- Assembler ist die menschenlesbare Form:

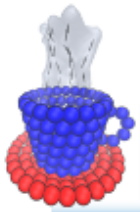
```
        # Maschinensprache # Assembler
                           X    = $1234
        AD 34 12           LDA X
        18                 CLC
        69 2A              ADC #42
        8D 34 12           STA X
        AD 35 12           LDA X+1
        69 00              ADC #00
        8d 35 12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

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

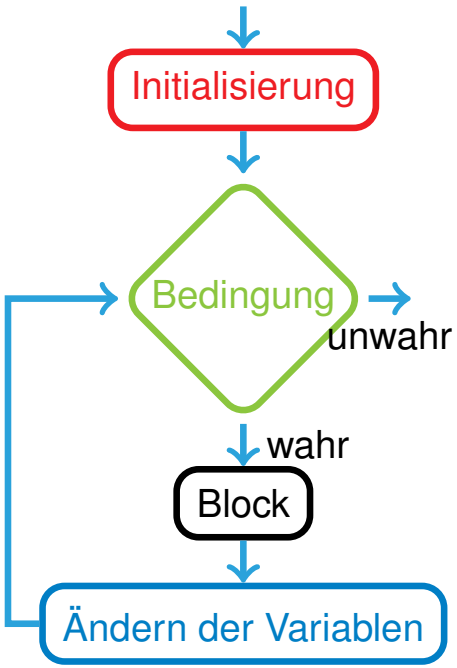
```
int global;

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() {
  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“

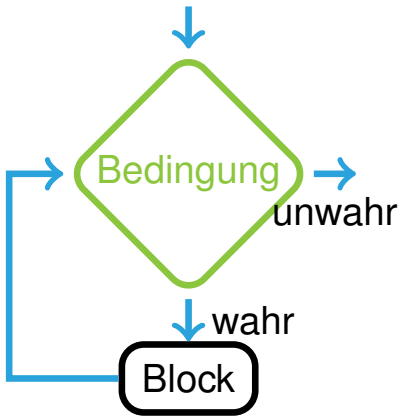
Also z.B.: `!((a == 1) || (a == 2))`

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

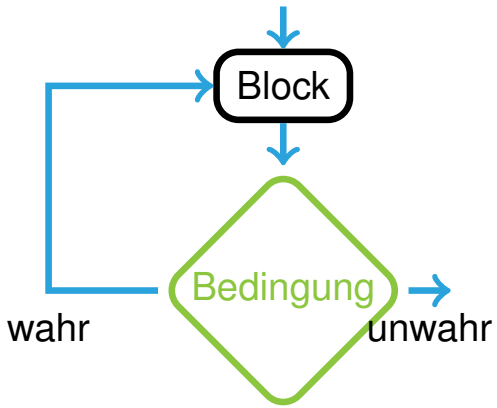
```
for(int i = 0; i < 10; ++i) {
  summe += i;
}
```



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 (Listen)

- **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)

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 (später)

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