http://www.icp.uni-stuttgart.de

Computergrundlagen
Programmieren in C

Axel Arnold

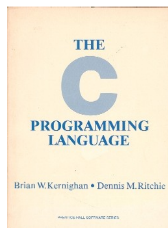
Institut für Computerphysik
Universität Stuttgart

Wintersemester 2010/11

http://www.icp.uni-stuttgart.de

Die Compilersprache C

D. M. Ritchie, *1941



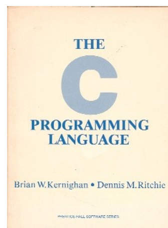
Geschichte

1971-73: Entwickelt von D. M. Ritchie
1978: C-Buch von Kernighan und Ritchie („K&R-C“)
1989: Standard ANSI C89 = ISO C90
1999: Standard ISO C99 (im folgenden benutzt)

- Zur Zeit Arbeiten am nächsten Standard, C1X
- Außerdem Compiler-spezifische Erweiterungen
- Objektorientierte Programmierung: Objective-C, C++

Die Compilersprache C

D. M. Ritchie, *1941



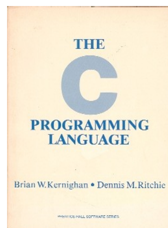
C-Compiler

- übersetzt C-Quellcode in Maschinencode
- GNU gcc, Intel icc, IBM XL C, Portland Group Compiler, ...
- Für praktisch alle Prozessoren gibt es C-Compiler
- Compiler optimieren den Maschinencode
- Compiler übersetzt nur Schleifen, Funktionsaufrufe usw.
- Bibliotheken für Ein-/Ausgabe, Speicherverwaltung usw.

http://www.icp.uni-stuttgart.de

Die Compilersprache C

D. M. Ritchie, *1941

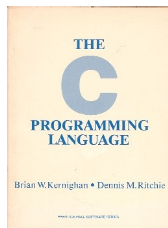


Einsatzgebiete

- C ist geeignet für effiziente und hardwarenahe Programme
- alle modernen Betriebssystem-Kernel sind in C geschrieben
- Der Python-Interpreter ist in C geschrieben
- Gnome oder KDE sind größtenteils C/C++-Code
- Viele physikalische Software ist C (oder Fortran)

Die Compilersprache C

D. M. Ritchie, *1941



Eigenschaften

- C-Code oft deutlich schneller als z.B. Python-Code
- Besonders bei numerischen Problemen
- Intensiver Einsatz von Zeigern
- Kein Schutz gegen Speicherüberläufe
- Erzeugter Maschinencode schwer zu verstehen

http://www.icp.uni-stuttgart.de

Hello, World!

```
#include <stdio.h>

int main()
{
  printf("Hello, World\n");
  return 0;
}
```

- C-**Quellcode** muss als Textdatei vorliegen (z.B. `helloworld.c`)
- Vor der Ausführung mit dem GNU-Compiler **compilieren**:

  `gcc –Wall –O3 –std=c99 –o helloworld helloworld.c`
- Erzeugt ein *Binär*programm `helloworld`,
  d.h. das Programm ist betriebssystem- und architekturspezifisch
- „-Wall -O3 -std=c99“ schaltet alle Warnungen und die Optimierung an, und wählt den ISO C99-Standard anstatt C90

http://www.icp.uni-stuttgart.de

Hello, World!

```
#include <stdio.h>

int main()
{
  printf("Hello, World\n");
  return 0;
}
```

- `printf` und `main` sind **Funktionen**
- `printf`: formatierte Textausgabe (analog „%“ in Python)
- `main`: Hauptroutine, hier startet das Programm
- Rückgabewert vom **Datentyp** **`int`**, keine Parameter („( )“)
- Code der Funktion: Block in geschweiften Klammern
- Anweisungen werden mit Semikolon beendet

Hello, World!

```
#include <stdio.h>

int main()
{
  printf("Hello, World\n");
  return 0;
}
```

- `return` beendet die Funktion `main`
- Argument von **`return`** ist der Rückgabewert der Funktion (hier 0)
- Der Rückgabewert von `main` geht an die Shell

http://www.icp.uni-stuttgart.de

Hello, World!

```
#include <stdio.h>

int main()
{
  printf("Hello, World\n");
  return 0;
}
```

- **`#include`** ist **Präprozessor-Anweisung**
- Bindet den Code der **Headerdatei** `stdio.h` ein
- Headerdateien beschreiben Funktionen anderer Module
  z.B. `stdio.h` Ein-/Ausgabefunktionen wie `printf`
- `stdio.h` ist Systemheaderdatei, Bestandteil der C-Umgebung
- Systemheaderdateien liegen meist unter `/usr/include`

Formatierung

```c
#include <stdio.h>

int main()
{
  printf("Hallo Welt\n");
  return 0;
}
```

```c
#include <stdio.h>
    int main(){printf(
"Hallo Welt\n");return 0;}
```

- C lässt viele Freiheiten bei der Formatierung des Quelltexts
- Aber: falsche Einrückung erschwert Fehlersuche

Lesbarer C-Code erfordert Disziplin!

- Weitere Beispiele: The International Obfuscated C Code Contest
  `http://www.ioccc.org/main.html`

Datentypen

Datentypen in C sind:

- **Grunddatentypen**

`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
- **Pointer** (Zeiger): Verweise auf Speicherstellen
- **Structs und Unions**: zusammengesetzte Datentypen, Datenverbünde
- **`void`** (nichts): Datentyp, der nichts speichert

Variablen

```
int global;
int main() {
  int i = 0, j, k;
  global = 2;
  int i;  // Fehler! i doppelt deklariert
}
void funktion() {
  int i = 2; // Ok, da anderer Gueltigkeitsbereich
  i = global;
}
```

Variablen

- *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

http://www.icp.uni-stuttgart.de

Variablen

```
int global;
int main() {
  int i = 0, j, k;
  global = 2;
  int i;  // Fehler! i doppelt deklariert
}
void funktion() {
  int i = 2; // Ok, da anderer Gueltigkeitsbereich
  i = global;
}
```

Globale Variablen

- hier: Die Ganzzahl `global`
- Deklaration außerhalb von Funktionen
- Aus allen Funktionen les- und schreibbar

Variablen

```
int global;
int main() {
  int i = 0, j, k;
  global = 2;
  int i;  // Fehler! i doppelt deklariert
}
void funktion() {
  int i = 2; // Ok, da anderer Gueltigkeitsbereich
  i = global;
}
```

Lokale Variablen

- hier: Die Ganzzahl `i`
- Gültigkeitsbereich ist der innerste offene Block
- daher kann `i` in `main` und `funktion` deklariert werden

http://www.icp.uni-stuttgart.de

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 – for

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

for-Schleifen bestehen aus

- Initialisierung der Schleifenvariablen
  - Eine hier deklarierte Variable ist nur in der Schleife gültig
  - Hier kann eine beliebige Anweisung stehen (z.B. auch nur `i = 1`)
  - Dann muss die Schleifenvariable bereits deklariert sein
- Wiederholungsbedingung:
  die Schleife wird abgebrochen, wenn die Bedingung unwahr ist (hier, bis i bzw. k = 100)
- Erhöhen der Schleifenvariablen

Schleifen – for

```
int i, j;
for (i = 1; i < 100; ++i) {
  if (i == 2) continue;
  printf("%d\n", i);
  if (i >= 80) break;
}
for (j = 1; j < 100; ++j) printf("%d\n", i);
```

- **`break`** verlässt die Schleife vorzeigt
- **`continue`** überspringt Rest der Schleife (analog Python)
- Jeder Teil kann ausgelassen werden – **`for`**`(;;)` ist eine Endlosschleife („forever“)
- Deklaration in der **`for`**-Anweisung erst seit C99 möglich
- Vorteil: Verhindert unbeabsichtigte Wiederverwendung von Schleifenvariablen

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 bzw. erniedrigen i um 1 *nach* der Auswertung 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
```

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)

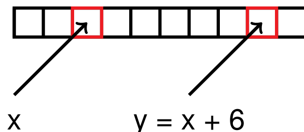
Zeichenketten

```
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)

http://www.icp.uni-stuttgart.de

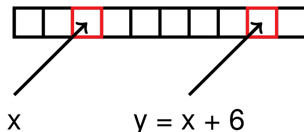
Zeiger




```
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
```

- Zeigervariablen (Pointer) zeigen auf Speicherstellen
- Ihr Datentyp bestimmt, als was der Speicher interpretiert wird (**void** * ist unspezifiziert)
- Zeiger werden mit einem führendem Stern deklariert
- Bei Mehrfachdeklarationen: genau die Variablen mit führendem Stern sind Pointer

Zeiger




```
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
```

- 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
- Addition von n zu einem Pointer versetzt um n Elemente

Referenzieren und Dereferenzieren

```
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

Unterschied zwischen Variablen und Zeigern

```
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)

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;
}
```

- Funktionen werden definiert mit
  `rettyp funktion(typ1 arg1, typ2 arg2,...) {...}`
- Ist der Rückgabetyp **`void`**, gibt die Funktion nichts zurück
- **`return`** verlässt eine Funktion vorzeitig (bei Rückgabetyp **`void`**)
- **`return`** `wert` liefert zusätzlich `wert` zurück

http://www.icp.uni-stuttgart.de

main

```
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