

# Computergrundlagen Programmieren in C

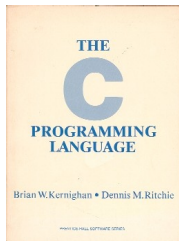
**Axel Arnold**

Institut für Computerphysik  
Universität Stuttgart

Wintersemester 2011/12



D. M. Ritchie, 1941 – 2011



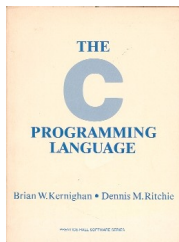
## Geschichte

- 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++



D. M. Ritchie, 1941 – 2011

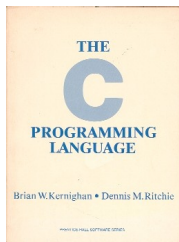


## C-Compiler

- 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.



D. M. Ritchie, 1941 – 2011

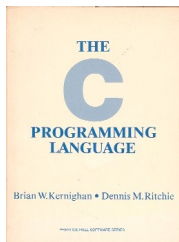


## Einsatzgebiete

- 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)



D. M. Ritchie, 1941 – 2011



## 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

```
#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 architekturenspezifisch
- „-Wall -O3 -std=c99“ schaltet bei gcc alle Warnungen und die Optimierung an, und wählt den ISO C99-Standard anstatt C90

```
#include <stdio.h>
```

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

- main ist eine selbstdefinierte **Funktion**
- Hier startet das Programm, daher muss es main immer geben
- Rückgabewert vom **Datentyp int**
- Keine Parameter („()“)
- Code der Funktion: Block in geschweiften Klammern
- Anweisungen werden mit Semikolon beendet

---

```
#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, vergleiche `exit` in `bash` oder `Python`



---

```
#include <stdio.h>
```

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

- 
- printf ist eine Bibliotheksfunktion
  - Formatierte Textausgabe analog „%“ in Python
  - Benötigt Headerdatei „stdio.h“
  - Einbinden einer Headerdatei entspricht in etwa „import“ in Python

```
#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

---

```
#include <stdio.h>
```

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

---

```
#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>

## • Grunddatentypen

<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
- **Pointer** (Zeiger): Verweise auf Speicherstellen
- **Structs und Unions**: zusammengesetzte Datentypen, Datenverbände
- **void** (nichts): Datentyp, der nichts speichert
  - Rückgabewert von Funktionen, die nichts zurückgeben
  - Zeiger auf Speicherstellen un spezifizierten Inhalts

```
int nenner = 1;
int 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
- Umwandlung in `void` verwirft den Ausdruck

```
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

```
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

```
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



```
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))`

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

**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  $i = 100$  bzw.  $k = 0$ .

- **Ändern der Schleifenvariablen**

hier  $i$  um eins erhöhen,  $k$  durch 2 teilen

```
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 vorzeitig
- **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 (beachte **i** in der letzten Schleife!)

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* 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
```

---

```
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
```

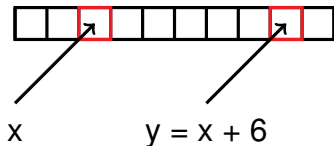
- Arrays (Felder) werden mit eckigen Klammern indiziert
- Mehrdimensionale Arrays erhält man durch mehrere Klammern
- Beim Anlegen wird die Speichergröße festgelegt
- Später lernen wir, wie man Arrays variabler Größe anlegt
- Es wird nicht überprüft, ob Zugriffe innerhalb der Grenzen liegen
- Die Folge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

```
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)

```
#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 Rückgabetyt **void**, gibt die Funktion nichts zurück
- **return** verlässt eine Funktion vorzeitig (bei Rückgabetyt **void**)
- **return** wert liefert zusätzlich wert zurück



- Zeigervariablen (Pointer) zeigen auf Speicherstellen
- 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



```
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

```
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

```
int x[10], *px = x;
printf("int: %ld int[10]: %ld int *: %ld\n",
      sizeof(int), sizeof(x), sizeof(px));
// → int: 4 int[10]: 40 int *: 8
```

- **sizeof**(datentyp) gibt an, wieviel Speicher eine Variable vom Typ `datentyp` belegt
- **sizeof**(variable) gibt an, wieviel Speicher die Variable `variable` belegt
- Bei Arrays: Länge multipliziert mit der Elementgröße
- Zeiger haben immer dieselbe Größe (8 Byte auf 64-Bit-Systemen)
- Achtung: Das Ergebnis ist 64-bittig (Typ **long unsigned int**), daher bei Ausgabe „%ld“ verwenden

```
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)

```
struct Position {  
    float x, y, z;  
};  
struct Particle {  
    struct Position position;  
    float ladung;  
    int identitaet;  
};
```

- **struct** definiert einen Verbund
- Ein Verbund fasst mehrere Variablen zusammen
- Die Größe von Feldern in Verbänden muss konstant sein
- Ein Verbund kann Verbände enthalten

```
struct Particle part, *ptr = &part;
```

```
part.identitaet = 42;
```

```
ptr->ladung = 0;
```

```
struct Particle part1 = { {1, 0, 0}, 0, 42};
```

```
struct Particle part2 = { .position = { 1, 0, 0 },  
                          .ladung = 0,  
                          .identitaet = 43};
```

```
struct Particle part3 = { .identitaet = 44};
```

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

```
struct Particle part;  
Particle part1;           // beides ist ok, selber Typ
```

```
Vektor3D vektor; // auch ok
```

```
struct Vektor3D vektor; // nicht ok, struct Vektor3D fehlt
```

- **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

```
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
- 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
- Bei Zeigern führt das zu Zeigern auf Zeiger (`typ **`) usw.



```
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

```
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 wiederum verhalten sich auch hier wie Zeiger
- Wenn diese Arrays enthalten, werden diese kopiert
- In diesem Fall ist die Größe des Arrays im Voraus bekannt

```
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

---

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

---

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

---

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

---

- **while** (cond) block und **do** block **while** (cond);  
führen block aus, solange die Bedingung cond wahr ist
- Unterschied zwischen den beiden Formen:
  - **while** überprüft die Bedingung cond *vor* Ausführung von block
  - **do ... while** erst danach
- Die drei Beispiele sind äquivalent
- Jede Schleife kann äquivalent als **for**-, **while**- oder **do...while**-Schleife geschrieben werden

```
char buch = argv[1][0];
switch (buch) {
case 'a':
    printf("a, rutscht durch zu b\n");
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
- Der **switch**-Block wird ganz abgearbeitet
- Kann explizit durch **break** verlassen werden

```
#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** testet, ob eine Makro definiert ist

```
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
- Statt Makros besser Variablen mit konstantem Typ
- Diese können nicht verändert werden
- Anders als Makros haben sie einen Typ
- Vermeidet seltsame Fehler, etwa wenn einem Zeiger ein **float**-Wert zugewiesen werden soll
- **static** ist nur bei Verwendung mehrerer Quelldateien wichtig

- 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



```
#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

```
#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);
```

- Wird dauernd Speicher belegt und nicht freigegeben, geht irgendwann der Speicher aus („Speicherleck“)
- Dies kann z.B. so passieren:

```
int *vek = (int *)malloc(100*sizeof(int));
vek = (int *)malloc(200*sizeof(int));
```

- Ein Bereich darf auch nur einmal freigegeben werden

```
#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  $\sin(x)^3$

```
#include <string.h>

char test[1024];
strcpy(test, "Hallo");
strcat(test, " Welt!");
// jetzt ist test = "Hallo Welt!"
if (strcmp(test, argv[1]) == 0)
    printf("%s = %s (%d Zeichen)\n", test, argv[1],
           strlen(test));
```

- `strlen(quelle)`: Länge eines 0-terminierten Strings
- `strcat(ziel, quelle)`: kopiert Zeichenketten
- `strcpy(ziel, quelle)`: kopiert eine Zeichenkette
- `strcmp(quelle1, quelle2)`: vergleicht zwei Zeichenketten, Ergebnis ist  $< 0$ , wenn lexikalisch  $quelle1 < quelle2$ ,  $= 0$ , wenn gleich, und  $> 0$  wenn  $quelle1 > quelle2$

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

```
char test[1024];
```

```
strncpy(test, "Hallo", 1024);
```

```
strncat(test, " Welt!", 1023 - strlen(test));
```

```
if (strncmp(test, argv[1], 2) == 0)
```

```
    printf("die 1. 2 Zeichen von %s und %s sind gleich\n",  
        test, argv[1]);
```

- Zu allen str-Funktionen gibt es auch strn-Versionen
- Die strn-Versionen überprüfen auf Überläufe
- Die Größe des Zielbereichs muss *korrekt* angegeben werden
- Die terminierende 0 benötigt ein Extra-Zeichen!
- Die str-Versionen sind oft potentielle Sicherheitslücken!

```
#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*

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

- printf gibt formatierten Text auf Standardausgabe aus
- printf liest Werte von der Standardeingabe
- Beide benutzen ähnliche Formatangaben wie auch Python

## scanf

- Benötigt *Zeiger* auf die Variablen, um die Werte zu setzen
- Bei Zeichenketten unbedingt die Größe des Puffers angeben!
- Gibt zurück, wieviele Werte gelesen werden konnten

```
#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);
```

- `stdio.h` stellt Dateien durch *Filehandles* vom Typ `FILE *` dar
- `Handle` ist Zeiger auf 0, wenn Datei nicht geöffnet werden kann
- Dateien öffnen mit `fopen`, schließen mit `fclose`
- Zum Schreiben öffnen mit Modus „w“ oder „a“ statt „r“



```
#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);
```

- fprintf und fscanf funktionieren wie printf und scanf
- Aber auf beliebigen Dateien statt stdout und stdin
- stdin, stdout und stderr sind Handles für die Standardgeräte

```
#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` (gefährlich!) und `snprintf`

```
#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

```
#define LANG_EN
```

```
void printer(char *str) {  
#ifdef LANG_EN  
    printf("String <%s>", str);  
#else  
    printf("Zeichenkette <%s>", str);  
} // Autsch, das geht schief!  
#endif
```

- Ersetzung erfolgt nur textuell
- Klammerung kann leicht verletzt werden
- Compilieren klappt hier nur, wenn das Makro nicht definiert ist



## Einteilen eines Programms in mehrere Quelldateien

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

## 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
  
- Namen von Headerdatei und Objektdatei sind unabhängig
- **Bibliotheken** sind Sammlungen von Objektdateien

---

```
// 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
- **extern** und **static** ändern ihre Sichtbarkeit

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

**static** (string1)

- Platz für diese Variable wird in dieser Objektdatei belegt
- Zugriff nur von der aktuellen Objektdatei
- Kann daher in mehreren Objektdateien definiert werden
- Konstanten sollten daher statisch deklariert werden:

```
static const double PI = 3;
```



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

ohne Vorgabe (string2)

- Platz für diese Variable wird in dieser Objektdatei belegt
- Zugriff auch von außen
- Darf nur in einer Objektdatei definiert werden
- Bei mehrfachem Auftreten gibt der gcc/ld sonst den Fehler „multiple definition of ‘var’“
- Insbesondere, wenn eine Objektdatei doppelt eingelinkt wird:  
gcc -o programm test1.o test1.o

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

**extern** (string3)

- Platz für diese Variable wird *nicht* in dieser Objektdatei belegt (wenn keine unqualifizierte Definition folgt)
- Diese Variable muss in einer anderen Objektdatei liegen
- In genau einer Objektdatei muss diese Variable ohne **extern** definiert werden
- Ist die Variable in allen Objektdateien nur extern deklariert, gibt gcc/ld den Fehler „undefined reference to ‘var’“

## modul.h

---

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

---

- Der **#ifndef/#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

## 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)

---

*# 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

- 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
```

---

## 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
```
- Shell-Befehl erzeugt aus den Quellen die Datei ziel

## 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
```

---

- Aufruf einfach mit `make ziel`
- `ziel` wird nur neu gebaut, wenn eine Quelle neuer ist
- Bei Bedarf werden auch alle Quellen erneuert
- Im Beispiel:
  - `make default` baut das Programm `bib`
  - `make libbib.a` baut nur die Bibliothek `libbib.a`



## 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
```

---

- Automatische Regeln für Objektdateien aus C-Quellcode
- Beispiel: `make bib_hello.o` baut die Objektdatei `bib_hello.o` aus `bib_hello.c`, wenn vorhanden
- Regeln ohne Quellen („clean“) werden immer ausgeführt
- Regeln ohne Befehl („default“) erneuern nur die Quellen
- Im Beispiel: `make default` entspricht `make bib`

- 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!

- 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 --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 --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

- 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*`

- **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>