

# Computergrundlagen Programmieren in C

**Jens Smiatek and Maria Fyta and Axel Arnold**

Institut für Computerphysik  
Universität Stuttgart

Wintersemester 2016/2017

# Die Compilersprache C

## Geschichte

- 1969-1973: entwickelt in Bell Labs von D.M. Ritchie für Systemprogrammierung des Unix
- 1978: C-Buch ('K&R-C') von Kernighan and Ritchie
- 1989: Standard ANSI C89 = ISO C90
- 1999: Standard ISO C99
- 2011: Standard C11

# Die Compilersprache C

## Eigenschaften

- 'general-purpose'
- imperative und strukturierte Programmiersprache
- Die meisten physikalische Software sind in C, bzw. Fortran geschrieben
- C-Code oft deutlich schneller als z.B. Python-Code
- Besonders bei numerischen Problemen
- Intensiver Einsatz von Zeigern
- Kein Schutz gegen Speicherüberläufe



# Die Compilersprache C

## 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.

## Anwendungen

- System- und Anwendungsprogrammierung
- Grundlegende Programme aller Unix-Systeme
- Systemkernel vieler Betriebssysteme, größtenteils GNOME, KDE
- Zahlreiche 'höhere' Programmiersprachen (Perl, Java, C++, C#, Objective-C, usw.) orientieren sich an Syntax von C
- Python-Interpreter ist in C geschrieben

# Hello, World!

---

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

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

---

---

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

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

---

- C-**Quellcode** muss als Textdatei vorliegen (z.B. helloworld.c)
- **Formatierung**: viele Freiheiten bei der Formatierung des C-Quelltexts (aber falsche Einrückung erschwert Fehlersuche)

## Compilieren

- 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 alle Warnungen und die  
Optimierung an, und wählt den ISO C99-Standard anstatt C90

## Standard Bibliothek

- **#include** ist **Präprozessor-Anweisung**
- Bindet den Code eine **Headerdatei** ein
- Die Standard-Bibliothek ist kein Teil der Programmiersprache C selbst, aber eine Umgebung die Standard-C realisiert
- Stellt Funktionen, Typen und Makros der Bibliothek zur Verfügung
- 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`

## Funktionen

- Jedes C-Programm besteht aus Funktionen (z.B. `printf` und `main`)
- Die Funktionen müssen nicht unbedingt einen Rückgabewert geben
- Das Hauptprogramm (sowie auch die Funktionen) werden durch geschweifte Klammern in Blöcke eingeteilt.
- Innerhalb des Blocks stehen dann die Anweisungen des C-Programms
- Anweisungen werden in C durch ein Semikolon `;` abgeschlossen.

### `main`

- `main`: Hauptroutine, hier startet das Programm
- Der Rückgabewert von `main` geht an die Shell



# Hello, World!

```
#include <stdio.h>

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

- `printf`: formatierte Textausgabe
- Rückgabewert vom **Datentyp** `int`, keine Parameter („()“)
- **return** beendet die Funktion `main`
- Argument von **return** ist der Rückgabewert der Funktion (hier 0)

# Datentypen in C

## • 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



## Variablen

```
int a,b,c,summe;  
a=1; b=2; c=3;  
summe=a+b+c;  
printf("Die Summe ist %d\n",sum);  
return 0;
```

Hier: a, b, c, summe  
sind Ganzzahlen.

- Alle Variablen in C müssen (nur einmal) vor Benutzung deklariert werden
- können bei der Deklaration mit Startwert **initialisiert** werden
- Die Variablenamen können 1-9 Zeichen haben (A-Z, a-z, 0-9, \_)
- Es gibt **lokale** und **globale** Variablen
- Globale Variablen werden außerhalb von Funktionen deklariert und sind von 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;
}
  
```

Hier:

- Ganzzahl `global` : **globale** Variable
- Ganzzahl `i` : **lokale** Variable
  - Gültigkeitsbereich ist der innerste offene Block
  - daher kann `i` in `main` und `funktion` deklariert werden

## Typumwandlung

```
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)
- Umwandlung in **void** verwirft den Ausdruck



## Ein-/Ausgabe Funktionen

putchar(c)

Ausgabe eines Zeichens  
auf der Standard-ausgabe  
(Bildschirm)

puts(str)

Ausgabe einer  
Zeichenfolge

---

printf ("%d + %d ergibt %d.", x  
, y, ergebnis)

---

formatierte Ausgabe (von  
Variablen unterschiedlichen  
Typs)

getchar()

Einlesen eines Zeichens  
von der Standard-eingabe  
(Tastatur)

gets(str)

Einlesen einer  
Zeichenfolge

scanf ("%d %f %d", &i, &x, &j)

Einlesen von Variablen  
verschiedenen Typs

## Einfaches I/O (getchar, putchar)

---

```
char c;  
c=getchar();  
putchar(c);
```

---

- `getchar()`, `putchar()`: die einfachsten Funktionen in der Standard-Bibliothek die **einzelne** Zeichen lesen/schreiben können



## Ausgabe – printf

```
int n=511;  
printf("Welche ist die Oktalzahl fuer %d?",n);  
printf("%s! %d dezimal ist %o oktal\n","Richtig", n,n);
```

- '%d' für eine Zahl im Dezimalsystem
- '%s' gibt einzelne Zeichen aus
- '%s' gibt eine ganze Zeichenkette (string) aus
- '%o' gibt eine Oktalzahl statt eine Dezimalzahl aus (ohne führende Null)
- '%xh' gibt eine Hexadezimalzahl statt eine Dezimalzahl aus
- '%n' Zeilenendezeichen (newline)
- '%h' horizontaler Tabulator (Tab)
- '%v' vertikaler Tabulator



## Eingabe – scanf

```
int num1 = 0, num2 = 0;

printf ("Geben Sie die erste Zahl ein: ");
scanf ("%3d", &num1);

printf ("Geben Sie nun die zweite Zahl ein: ");
scanf ("%d", &num2);

printf ("\nDie eingegebenen Zahlen waren:\n");
printf ("num1: %d\nnum2: %d\n", num1, num2);

return 0;
```

2 Ganzzahlen werden eingelesen. Die erste muss max. 3 Stelle haben ('%3d').

längere Eingabe: Rest bleibt im Eingabepuffer und wird von `scanf` weitergelesen.

## Bedingte Ausführung – if

if (Ausdruck) Anweisung

---

```
c=getchar();  
if ( ) c=='?')  
printf("warum hasst du ein Fragezeichen gegeben?\n");
```

---

### Vergleiche und logische Verknüpfungen

---

- == gleich
- != : nicht gleich
- > (<) : größer (kleiner)
- >= (<=): größer(kleiner) oder gleich

```
if (a<b) {  
    t=a;  
    a=b;  
    b=t;  
}
```

---

- „&&“: logisches „und“
- „||“ : logisches „oder“
- „!“ : logisches „nicht“

## else Anweisung

if (Ausdruck) Anweisung1 else Anweisung2

---

**if**(a<b)

    x=a;

**else**

    x=b;

---

oder

Ausdruck1 ? Ausdruck2 : Ausdruck3:

Ausdruck1 wird ausgewertet; wenn nicht Null ist der Wert gleich dem Ausdruck2, sonst ist der Wert gleich Ausdruck3. z.B. der Wert von

---

a<b ? a : b;

---

ist a wenn a kleiner als b und b sonst.

## if, else

- Es gibt kein 'elseif', aber Blocks von **if**'s und **else**'s `if(...)`

```
{...}  
else if(...)  
  {...}  
else  
  {...}
```

---

```
if( c==' ' || c=='\t' || c=='\n')...
```

---

Überprüft ob das Zeichen  
c leer, ein Tabulator oder  
ein Zeilenendezeichen ist.

---

```
int i=1, j=5, m;  
if (i>j)  
  m=i;  
else  
  m=j;
```

---

i,j vergleichen um den  
Wert von m  
auszuwerten

## while – Schleife

### while (Ausdruck) Anweisung

Die Anweisung wird ausgeführt, wenn der Kontrollausdruck einen Wert ungleich 0 ergeben hatte.

---

```
int i=0;
while(i!= 10)                Inkrement von i, bis i nicht 10 wird
{
    printf(" i is %d\n", i);
    i++;
}
```

---

```
int c;
c=getchar();
while (c!=EOF)               Dateien kopieren
{
    putchar(c);
    c=getchar();
}
```

---

## do while – Schleife

do Anweisung while (Ausdruck);

Die Ausführung der Anweisung wird solange wiederholt, bis der Kontrollausdruck den Wert 0 ergibt. (Im Unterschied zur while-Anweisung wertet die do-Anweisung den Kontrollausdruck erst nach der Ausführung einer zugehörigen Anweisung aus.)

---

```
int i=1;
do
{
    printf("%d\n", i++);
} while (i<=10);
```

---

Im Beispiel, werden die Zahlen von 1 bis 10 gegeben.

## Arithmetik

Die übliche Rechenzeichen werden verwendet: '+', '-', '\*', '/' und Modulo '%'.  
 $x = a \% b \rightarrow a \bmod b$ :  $x$  ist der Rest der Division  $a$  geteilt durch  $b$ .

**char** Variablen können wie **int** Variablen behandelt werden.

z.B. `c=c+'A'-'a'`: wandelt einen kleingeschriebenen ascii Buchstaben der in `c` gespeichert ist in einen großgeschriebenen Buchstaben um (Voraussetzung: fester Abstand aller ascii Zeichen)

---

```
char c;
while((c=getchar())!='\0')
    if('A'<=c && c<='Z')           wandelt Groß- in
        putchar(c+'a'-'A');       Kleinschreibung um.
    else
        putchar(c);
```

---

## 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* 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
```

---



## for – Schleife

for (Ausdruck-1<sub>opt</sub>; Ausdruck-2<sub>opt</sub>; Ausdruck-3<sub>opt</sub>) Anweisung  
for (Deklaration Ausdruck-2<sub>opt</sub>; Ausdruck-3<sub>opt</sub>) Anweisung

---

```
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**

## for – Schleife

---

```
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 („forever“)
- Deklaration in der **for**-Anweisung erst seit C99 möglich
- Vorteil: Verhindert unbeabsichtigte Wiederverwendung von Schleifenvariablen

## Bedingte Ausführung – switch

switch (Ausdruck)

```
{  
case Wert1: Anweisung1;  
           Anweisung2;  
           ...  
           break;  
case Wert2: Anweisung1;  
           Anweisung2;  
           ...  
           break;  
           ...  
           default : Anweisungen;  
}
```

- 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

## switch – Beispiel

```
int eingabe;
printf ("Geben Sie einen Wert ein: ");
scanf ("%d",&eingabe);
switch (eingabe)
{
    case 1: printf ("Eine eins wurde eingegeben.");
            break;
    case 2: printf ("Die Eingabe war zwei.");
            break;
    case 3: printf ("Drei wurde eingegeben.");
            break;
    default: printf ("Es wurde etwas anderes angegeben.");
}
return 0;
```

- Mehrfache Unterscheidungen: Der Eingabewert muss sich auf drei (hier:1, 2 oder 3) verschiedene Werte überprüft werden.
- Mit `switch` lassen sich Verzweigungen bilden.

## Arrays (Felder)

---

```
int arrayname[n];
```

---

- Ein Array dient zur Speicherung größerer Datenmengen des gleichen Datentyps
- Arrays werden mit eckigen Klammern indiziert
- Um auf ein einzelnes Element zuzugreifen, ist der Name des Arrays und den Index des Elements benötigt
- In C beginnt die Nummerierung bei Null (also maximale Länge  $n-1$ )
- Beim Anlegen wird die Speichergröße festgelegt
- Es ist auch möglich Arrays variabler Größe anzulegen
- Alle Feldelemente, die nicht ausdrücklich initialisiert wurden, bekommen automatisch den Wert 0.

## Arrays – Beispiel

```
int dat[zahl], i;
for (i = 0; i < zahl; i++)
    dat[i] = 0;
for (i = 0; i < zahl; i++)
{
    printf ("%d. Element: ", i);
    scanf ("%d", &dat[i]);
    if (dat[i] == 0)
    {
        printf ("\n Die Eingabe war: \n");
        break;
    }
}
for (i = 0; i < zahl; i++)
{
    if (dat[i] != 0)
        printf ("%d. Element: %d\n", i, dat [i]);
    else
        break; }

```

## Mehrdimensionale Arrays

---

**int** arrayname[n][m];      Feld mit n Zeilen und m Spalten

---

- Mehrdimensionale Arrays erhält man durch mehrere Klammern
- Arrays unflexibel was die Anzahl der Elemente angeht.
- Die Anzahl der Elemente muss vor Ablauf des Programm bekannt oder überdimensioniert sein
- Lösung durch 'dynamische Datenstrukturen' (`malloc`, `free`.)

## Mehrdimensionale Arrays – Beispiel

```
#include <stdio.h>
int main(void) {
    int i,j;
    int name[n][m];
    for(i=0; i < n; i++) {
        for(j=0; j < m; j++) {
            printf("Wert fuer name [%d] [%d]:", i, j);
            scanf("%d", &name[i][j]);
        }
    }
    printf("\nAusgabe von name [%d] [%d]...\n\n", n,m);
    for(i=0; i < n; i++) {
        for(j=0; j < m; j++) {
            printf("\t%4d ",name[i][j]);
        }
        printf("\n\n");
    }
}
```



## Strings – Zeichenketten

---

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

---

- Strings: eine Folge (Arrays) von Typ **char** (oder pointer **char\*** )
- Jedes Element des Arrays ist für ein Zeichen zuständig
- Einzelne Zeichen werden im Arbeitsspeicher nacheinander abgespeichert
- Das String-Ende wird durch eine Null markiert die auch auf ein Byte Speicherplatz zuweist
- Es ist einfach, mit Strings Speicherzugriffsfehler zu bekommen.

## Strings – Beispiel

---

```
char datain[255];  
printf ("Text eingeben:\n");  
scanf ("%s", datain);  
// fgets (datain, 255, stdin);  
printf ("Text ausgeben:\n%s\n", datain);  
return 0;
```

---

- `scanf()` liest immer nur bis zum Auftreten des ersten Leerzeichens (bis zu einem Zeilenumbruch (`\n`))
- Abhilfe durch `fgets()` (in `stdio.h`) ohne `&`
- `fgets()` verlangt maximale Länge, sowie Stream von dem gelesen werden soll.



# Funktionen

Rückgabetyp Funktionsname(Parameterliste)

```
{  
Anweisungen  
}
```

wobei Parameterliste=typ1 arg1 typ2 arg2,...

- Modularisierung: das Programm wird in mehrere Programmabschnitte (Module oder Funktionen) zerlegt
- Vorteile:
  - Fehler lassen sich leichter finden
  - Bessere Lesbarkeit
  - Wiederverwendbarkeit
- Funktionstyp: sagt Compiler dass ein Werte mit dem bestimmten Typ zurückgegeben wird

## Funktionen-Beispiel

---

```
cpstring(s1,s2)
    char s1[], s2[];
{
    int i;
    for (i=0;(s2[i]=s1[i])!='\0';i++);
return(s1,s2);
}
```

---

Die Funktion `cpstring()` kopiert ein `char` Feld (array).

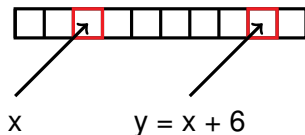
- `return`: Funktion wird beendet und ein Wert an die aufrufende Funktion wird zurückgegeben
- Der Datentyp des Ausdrucks muss mit dem Typ des Rückgabewertes des Funktionskopfs übereinstimmen
- Ist der Rückgabebetyp `void`, gibt die Funktion nichts zurück
- `return` verlässt eine Funktion vorzeitig (bei Rückgabebetyp `void`)
- `return` wert liefert zusätzlich wert zurück



## Variablen ansprechen

- Bisher: Variablen wurden immer direkt über ihren Namen angesprochen
- Intern im Rechner: Variablen werden immer über eine Adresse angesprochen (wenn Variable nicht in einem Prozessorregister befindet)
- also die Variablen bleiben im aufrufenden Code stets unverändert
- Die Speicherzellen erhalten eine Adresse
- eine Variable kann auch direkt über diese Adresse angesprochen werde
- Abhilfe: Übergabe der Speicheradresse der Variablen in Zeiger
- Bei Zeigern führt das zu Zeigern auf Zeiger (typ  $\star\star$ ) usw.

## Datentypen – Zeiger



- 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

## Datentypen – Zeiger

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

- 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
  
- Der unäre Adressoperator & (Referenzoperator) bestimmt die Adresse der Variable: & **Variablenname**
- Der unäre Zugriffoperator \* (Dereferenzoperator) erlaubt den (indirekten) Zugriff auf den Inhalt, auf den der Pointer zeigt: \* **pointer**
- Die Daten können wie Variablen manipuliert werden.

## Zeiger – 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+1) ist äquivalent zu p[1]
- \*(p + n) ist äquivalent zu p[n]
- &v gibt einen Zeiger auf die Variable v



## Beispiele – mit/ohne Zeiger

[1] Die Länge eines `char`

[2] Ein `char` Feld kopieren.

Feldes ausgeben.

Ohne Zeiger:

---

```
length(s)
char s[];{
  int n;
  for (n=0; s[n] !='\0';)
    n++;
  return(n);    }
```

---

```
cpstring(s,t)
char *s,*t;{
  while(*t++=*s++);
}
```

---

Mit Zeiger:

---

```
length(s)
char *s;{
  int n;
  for (n=0; s* !='\0';s++)
    n++;
  return(n);    }
```

---

```
cpstring(s1,s2)
char s1[]s2[];
{
  int i;
  for (i=0;(s2[i]=s1[i])!='\0';i++);
}
```

---

## Funktionen – Argumente

- C: 'call by value'
- When man eine Funktion wie  $f(x)$  aufruft, der Wert von  $x$  und nicht seine Adresse wird übergeben,
- also man kann  $x$  nicht innerhalb von  $f$  ändern
- kein Problem when  $x$  ein array ( $x$  ist dann sowieso eine Adresse), aber when  $x$  eine Skalarvariable?

---

```
flip(x,y)
  int*x,*y;{
    int tmp;
    tmp=*x;
    *x=*y;
    *y=tmp;
  }
```

---

- **flip** wird dann aufgerufen: `flip(&a,&b)`.

## Argumente aus der Kommandozeile

`main(argc, argv[])`

- Argumente aus der Kommandozeile beim Start an Programm übergeben
- `argc`(**a**rgument **c**ount): Anzahl der Argumente auf der Kommandozeile, mit der das Programm aufgerufen wurde
- `argv`(**a**rgument **v**ector): Zeiger auf einen Vektor von Zeigern auf Zeichenketten, die die Argumente enthalten (ein Argument pro Zeichenkette)
- `argv[0]`: Name mit dem das Programm aufgerufen wurde → `argc` wenigstens 1
- `argc=1`, gibt keine Argumente nach dem Programmnamen auf der Kommandozeile
- Das erste optionale Argument ist `argv[1]` und das letzte `argv[argc-1]`

z.B.: `echo hello world:`

`argc=3, argv[0]=echo, argv[1]=hello, argv[2]=world`

## Argumente aus der Kommandozeile – Beispiel

---

```
#include <stdio.h>
main (int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
        printf("\n");
    return 0;
}
```

---

- Programm aufrufen: a.out hello world
- Ausgabe: hello world

## struct – Datenverbände

---

```
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 Datenverbände können als eine Einheit verwendet werden
- Die Größe von Feldern in Verbänden muss konstant sein
- Ein Verbund kann Verbände enthalten
- Verweis auf eine Komponente eines bestimmten Verbundes:  
**Verbund-Variablenname.Komponente**

## Variablen mit einem struct-Typ

```
struct complex
{ double re; // 1. Komponente der Struktur
  double im; // 2. Komponente der Struktur
} x,          // Variable x dieses Typs
  *cptr      // Zeiger auf eine Variable dieses Typs
float f;     // reelle Variable
f=x.re;      // Zugriff auf die 1. Komponente re von x
f=cptr->im;  // Zugriff auf die 2. Komponente im von cptr
```

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

## Dateien - fopen/fclose

---

```
FILE *fopen (const char *Pfad, const char *Modus);
```

---

- `stdio.h` stellt Dateien durch *Filehandles* dar
- Pfad: Dateiname
- `fopen` gibt NULL zurück, wenn der Datenstrom nicht geöffnet werden konnte, ansonsten einen Zeiger vom Typ FILE auf den Datenstrom.
- Modus:
  - r: Datei nur zum Lesen öffnen (read)
  - w: Datei nur zum schreiben öffnen (write)
  - a: Daten an das Ende der Datei anhängen (append)
  - r+: Datei zum Lesen und Schreiben öffnen (Datei muss schon existieren)
  - usw...
  - b: Binärmodus (anzuhängen an die obigen Modi, z.B. 'rb' oder 'r+b')
  - (in unixoide Systemen hat es keine Auswirkungen)

## fopen/fclose – Beispiel

---

```
#include <stdio.h>
int main (void)
{ FILE *datei;
  datei = fopen ("testdatei.txt", "w");
  fprintf (datei, "Hallo, Welt\n");
  fclose (datei);
  return 0; }
```

---

- Dateien öffnen mit `fopen`, schließen mit `fclose`
- Alle nicht geschriebenen Daten des Stroms `*datei` werden gespeichert
- alle ungelesenen Eingabepuffer geleert
- der automatisch zugewiesene Puffer wird befreit
- der Datenstrom `*datei` geschlossen
- der Rückgabewert der Funktion ist EOF, falls Fehler aufgetreten sind, ansonsten ist er 0 (Null)





## Sonstige I/O stdio.h Funktionen

**fread** (**void** \*daten, size\_t groesse, size\_t anzahl, FILE \*datei)  
**fwrite** (**const void** \*daten, size\_t groesse, size\_t anzahl, FILE \*datei)  
**int** fseek (FILE \*datei, **long** offset, **int** von\_wo);

- **fprintf** und **fscanf** funktionieren wie **printf** und **scanf**,
- aber auf beliebigen Dateien statt stdout (Bildschirm) und stdin (Tastatur)
- **fread**: liest einen Datensatz
- **fwrite**: speichert einen Datensatz
- **fwrite**, **fread** Funktionen geben die Anzahl der geschriebenen/gelesenen Zeichen zurück
- die **groesse** ist jeweils die Größe eines einzelnen Datensatzes
- es können **anzahl** Datensätze auf einmal geschrieben werden.
- Sonst noch in **stdio.h**: **fseek**, **fflush** ...

## 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 (gefährlich!) und snprintf



## 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 (z.B. die Wurzeln, Potenzen, Logarithmen, usw.) zur Verfügung
- Bibliothek einbinden mit  
gcc -Wall -O3 -std=c99 -o mathe mathe.c -lm
- Beispiel erstellt Tabelle mit Werten  $x$  und  $\sin(x)^3$

## string.h – Stringfunktionen

---

```
#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  $quelle1 > quelle2$

## string.h – Stringfunktionen

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

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

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

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

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

```
    printf("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!

## Größe von Datentypen – sizeof

---

```
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 Speicherplätze eine Variable vom Typ datentyp belegt
- **sizeof**(variable) gibt an, wieviel Speicherplätze die Variable variable belegt
- **sizeof** liefert einen ganzzahligen Wert ohne Vorzeichen zurück
- Bei Arrays: Länge multipliziert mit der Elementgröße
- Zeiger haben immer dieselbe Größe (8 Byte auf 64-Bit-Systemen)

## Speicherverwaltung – 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
- malloc reserviert Speicher
- realloc verändert die Größe eines reservierten Bereichs
- free gibt einen Bereich wieder frei
- Wird dauernd Speicher belegt und nicht freigegeben, geht irgendwann der Speicher aus („Speicherleck“)
- Ein Bereich darf auch nur einmal freigegeben werden



## typedef

---

```
typedef float length;  
length len, maxlen;  
length *vlength[];
```

---

- **typedef** definiert neue Namen für Datentypen
  - **typedef** ist nützlich, um Datentypen auszutauschen, z.B. double anstatt float
- 

```
typedef struct MyStruct {  
    int data1;  
    char data2;  
} newtype;  
  
newtype a;
```

---

- Deklarationen vereinfachen



## 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
- 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