Grundlagen der Programmiersprache C

Stephanie Friedhoff, Martin Lanser

Universität zu Köln

Einführung in HPC
SS 2016

Überblick

Überblick

Aufbau eines C-Programms

Listing 1: Hello World

```c
/* Einbinden der Standard Input and Output Library
   damit die Funktion printf() fuer die Ausgabe
   zur Verfuegung steht. */
#include <stdio.h>

int main (void)
{
    /* Ausgabe von "Hello world!"
       \n erzeugt einen Zeilenumbruch */
    printf("Hello world!\n");

    return 0;
}
```

- `main()` ist die Hauptfunktion bzw. das Hauptprogramm.
- Andere Funktionen können vor `main()` deklariert und definiert werden und stehen dann innerhalb der `main()` zur Verfügung.

Einbinden von Bibliotheken

(Programm-) Bibliothek

Bibliotheken sind Sammlungen von Funktionen.

- Bibliotheken werden am Beginn der C-Datei mit dem Befehl

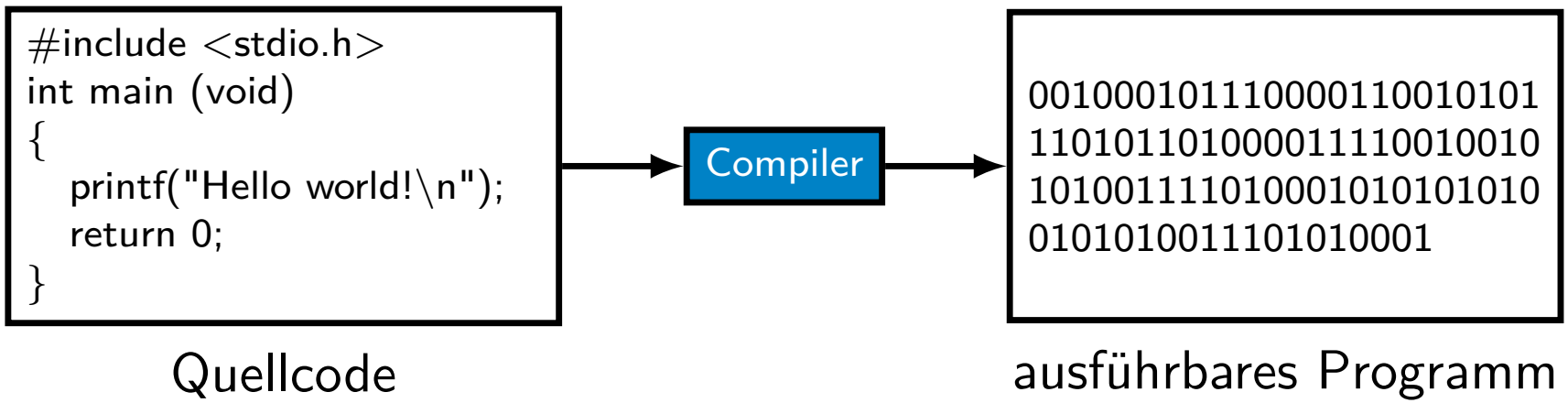
  ```
  #include <Name_der_Bibliothek.h>
  ```

  eingebunden und somit werden Funktionen der Bibliothek der C-Datei bereitgestellt.
- Nützliche Bibliotheken sind u. a.
  - stdio.h (**St**andar**d** **I**nput and **O**utput Library)
    enthält Ein- und Ausgabefunktionen wie z. B. `printf`
  - stdlib.h (**St**andar**d** **Lib**rary)
    enthält u. a. Funktionen zur Umwandlung von Datentypen (`atoi`, `atof`) und zur Speicherverwaltung (`malloc`)
  - math.h vereinbart gebräuchliche mathematische Funktionen wie z. B. `sin`, `cos`, `fabs` (Betrag) oder `floor`

Compiler zur "Übersetzung" eines C-Programms

Compiler

Ein Compiler ist ein Computerprogramm, das Quellcode (ein in einer bestimmten Programmiersprache geschriebenes Programm) in eine Form übersetzt, die von einem Computer ausgeführt werden kann.



- Für die Programmiersprache C gibt es verschiedene Compiler.
- Wir verwenden gcc aus der **G**NU **C**ompiler **C**ollection.

---

https://de.wikipedia.org/wiki/Compiler

Kompilieren und ausführen eines C-Programms im Konsolenfenster

```
gcc hello.c -o hello_world
```

- `gcc` ist der Befehl für den C-Compiler
- `hello.c` ist die zu kompilierende Quelldatei
- `-o` ist die Option von gcc zur Wahl des Namens des ausführbaren Programms
- `hello_world` ist der Name des ausführbaren Programms
- *Hinweis:*

  ```
  gcc hello.c
  ```

  erzeugt das ausführbare Programm `a.out`
- Die Ausführung des Programms erfolgt mit dem Befehl `./Programmname`, z. B.

  ```
  ./hello_world    oder    ./a.out
  ```

Überblick

Variablen in C

- Der Datentyp einer Variablen beschreibt, wie der Inhalt zu verstehen ist, z. B. ganze Zahl, Gleitkommazahl, Zeichen, Zeichenkette (String)
- Deklaration und Definition einer Variablen in C:

```
Datentyp variablenName;
```

  → Speicherreservierung (Anzahl Bits ist durch den Datentyp bestimmt)

  → Benennung der Variable für den Compiler

- Initialisierung einer Variablen in C:

```
Datentyp variablenName = wert;
```

  → Der Variablen `variablenName` vom Typ `Datentyp` wird der Wert `wert` zugewiesen

- *Beispiel:* Die Definition einer ganzen Zahl und Initialisierung mit dem Wert 3 lautet in C-Syntax:

```
int eineGanzeZahl = 3;
```

Ganzzahlige Datentypen in C

- 4 (bzw. 5) verschiedene, ganzzahlige Datentypen:
  (`char`,) `short`, `int`, `long` und `long long`
- vorzeichenlos mit dem Präfix `unsigned`, z. B. `unsigned int`
- Die 4 Typen unterscheiden sich durch ihren Wertebereich und ihren Speicherbedarf (verschieden für 32 und 64 Bit Systeme):
  - Wertebereich: 0 bis $2^{\text{Anzahl Bits}-1} - 1$ (`unsigned`)

    $$\text{bzw. } -\left(2^{\text{Anzahl Bits}-1}\right) \text{ bis } 2^{\text{Anzahl Bits}-1} - 1$$

  - `char` (8 Bit) Wertebreich: -128 bis 127 bzw. 0 bis 255
  - `short` (16 Bit) Wertebreich: -32 768 bis 32 767 (`signed`)
  - `int` (32 Bit) Wertebreich:
    -2 147 483 648 bis 2 147 483 647 / 0 bis 4 294 967 295
  - `long` bzw. `long long` (64 Bit) Wertebreich:
    -9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807
    0 bis 18 446 744 073 709 551 615 (`unsigned`)

Gleitkommazahlen in C

- 3 verschiedene Datentypen: `float`, `double` und `long double`
- Gleitkommazahlen werden in der Form $s * m * b^e$ gespeichert
- s bestimmt das Vorzeichen (Speicherbedarf: 1 Bit)
- b ist die Basis, meistens $b = 2$
- e ist der Exponent; die Länge des Exponenten bestimmt die Spannweite des betrachteten Zahlenraums
- m ist die Mantisse bestehend aus p Ziffern zur Basis b; p (precision) ist die Genauigkeit
- Einfache Genauigkeit: `float` (32 Bit), Mantissenlänge: 23 Bit (Genauigkeit: 7 bis 8 Dezimalstellen)
- Doppelte Genauigkeit: `double` (64 Bit), Mantissenlänge: 52 Bit (Genauigkeit: 15 bis 16 Dezimalstellen)
- `long double` (80-128 Bit), Mantissenlänge: mind. 63 Bit (Genauigkeit: mind. 19 Dezimalstellen)

Ausgabe von Daten auf den Bildschirm

- Ausgabefunktion aus der stdio.h-Bibliothek: `printf()`; so gibt

```
printf("Hello world!\n");
```

  den Text zwischen den Anführungszeichen (Hello world!) auf den Bildschirm aus.
- Der Backslash `\` markiert sogenannte Escape-Sequenzen, mit denen sich z. B. Zeilenumbrüche (`\n`) realisieren lassen.
- Für die Ausgabe von Daten verwenden wir Formatierungstypen der Form `%T`, wobei T den Datentypen angibt, z. B. `%d` für `int`, `%f` für `float` und `double`
- Auszugebende Variablen werden durch Kommata getrennt hinter dem auszugebenden Text aufgelistet.

```
int    einInt = 7;
double einDouble = 3.14;
printf("einInt: %d, einDouble: %f", einInt, einDouble);
```

Formatierte Ausgabe ganzer Zahlen

- Länge vorgeben / Auffüllen mit Leerzeichen (`%XXd`)

```
int aepfel = 10, orangen = 4;
printf("%3d Aepfel\n%3d Orangen\n", aepfel, orangen);
```

erzeugt die Ausgabe

```
 10 Aepfel
  4 Orangen
```

- Auffüllen mit Nullen (`%0d`)

```
int tag = 11, monat = 4, jahr = 2016;
printf("Ein Datum: %02d.%02d.%4d\n", tag, monat, jahr);
```

erzeugt die Ausgabe

```
Ein Datum: 11.04.2016
```

Formatierte Ausgabe von Gleitkommazahlen

- Anzahl der Vor- und Nachkommastellen vorgeben (`%VKS.NKSf`)

```
double myPi = 3.141592653589793;
printf("Ohne Formatierung: %f\n", myPi);
printf("3 VKS, 4 NKS: %3.4f\n", myPi);
```

erzeugt die Ausgabe

```
Ohne Formatierung: 3.141593
3 VKS, 4 NKS: 3.1416
```

Eingabe von Daten über die Tastatur

- Eingabefunktion aus der stdio.h-Bibliothek: `scanf()`
- Wie bei `printf()` geben wir in Anführungszeichen eine Formatanweisung der Form `%T` für einzulesende Datentypen an, z. B. `%d` für `int`, `%f` für `float` und `%lf` für `double`
- Einzulesende Variablen werden durch Kommata getrennt hinter der Formatanweisung aufgelistet, wobei vor jede Variable ein kaufmännisches Und (`&`) geschrieben wird.

```
int     einInt;
double  einDouble;
printf("Bitte eine ganze Zahl eingeben: ");
scanf("%d", &einInt);
printf("Bitte eine Gleitkommazahl eingeben: ");
scanf("%lf", &einDouble);

printf("Ganze Zahl und Gleitkommazahl "
       "(durch Leerzeichen getrennt) eingeben: ");
scanf("%d %lf", &einInt, &einDouble);
```

Überblick

Syntax einer Funktion

```
Datentyp_Rueckgabe Name(Datentyp_Par_1 Parameter_1, ...
                        Datentyp_Par_n Parameter_n)
{
    /* Anweisungen */
    return Rueckgabe;
}
```

Dabei ist der Funktionskopf definiert durch

- den Datentypen des Rückgabewertes (Datentyp_Rueckgabe)
  Bemerkung: Bei einer Funktion ohne Rückgabewert: void
- den Funktionsnamen (Name) und
- den Parametern inklusive Datentyp der Parameter (Datentyp_Par Parameter)

und der Funktionsblock durch

- Anweisungen und
- die Rückgabe (return Rueckgabe;)

Übung 1: Subtraktion zweier Gleitkommazahlen

Schreiben Sie ein C-Programm zur Subtraktion zweier Gleitkommazahlen. Die Rechnung soll in einer eigenen Funktion erfolgen.

Hauptprogramm `main`

- Ausgabe zur Eingabeaufforderung zweier Gleitkommazahlen
- Einlesen des Minuenden und Subtrahenden
- Aufruf der Funktion `subrahiere`
- Ausgabe der Differenz

Funktion `subtrahiere`

- Parameter: Minuend und Subtrahend
- Rückgabewert: Differenz

Parameter der Hauptfunktion `main()`

- Funktionskopf der Hauptfunktion bisher:

```
int main (void)
```

  d. h. `main` ohne Parameter

- Wollen wir einem Programm bei der Ausführung Werte übergeben, so muss der Funktionskopf der Hauptfunktion angepasst werden:

```
int main (int argc, char **argv)
```

  mit den Parametern

  - `argc` (**arg**ument **c**ount); Datentyp `int`
    Anzahl der übergebenen Argumente (=von Leerzeichen getrennte Strings)
  - `argv` (**arg**ument **v**alues); Datentyp `char`-Zeiger-Array
    Werte der übergebenen Argumente

Parameter der Hauptfunktion `main()` – Ein Beispiel

```
int main (int argc, char **argv)
```

sei die Hauptfunktion eines Programms `meinProgramm`

- Lautet der Programmaufruf z. B.

  `./meinProgramm max 10 7.8`

  so gilt:

  → `argc = 4`

  → `argv[0] = "./meinProgramm"`, `argv[1] = "max"`
  `argv[2] = "10"` und `argv[3] = "7.8"`

- **Achtung:** Alle Argumente sind Zeichenketten (`strings`)!
  Daher ist eine Typ-Umwandlung notwendig:

  → `argv[2] = "10"` als `int`: int ganzeZahl = atoi(argv[2]);

  → `argv[3] = "7.8"` als `double`: double zahl = atof(argv[3]);

  (`atoi` und `atof` sind in der `stdlib`-Bibliothek definiert.)

Überblick

Verzweigungen mit if und else

- Verzweigungen innerhalb eines Programms werden durch Bedingungen entschieden.
- In C können dazu `if-else-Anweisungen` benutzt werden:

```
if (Bedingung)
{
    /* Anweisungen_if */
}
else
{
    /* Anweisungen_else */
}
```

- Ist die Bedingung wahr, so werden `Anweisungen_if` im `if`-Block ausgeführt.
- Ist die Bedingung falsch, so werden `Anweisungen_else` im `else`-Block ausgeführt.

Verzweigungen mit if und else – ein einfaches Beispiel

Wenn die Zahl a kleiner 0 ist, gib aus, dass a negativ ist.
Ansonsten gib aus, dass a nicht negativ ist.

lautet in C-Syntax

```
if (a < 0)
{
    printf("a ist negativ\n");
}
else
{
    printf("a ist nicht negativ\n");
}
```

Hinweis: Da jeweils nur eine Anweisung in den `if`- und `else`-Blöcken steht, können die geschweiften Klammern (`{ }`) auch weggelassen werden.

Verzweigungen mit if und else – Verschachtelungen

Zum Prüfen von verschiedenen Fällen nacheinander, können `if`-Anweisungen verschachtelt werden:

```
if (a < 0)
{
    printf("a ist negativ\n");
}
else
{
    if (a > 0)
    {
        printf("a ist positiv\n");
    }
    else
    {
        printf("a ist gleich Null\n");
    }
}
```

Vergleichsoperatoren und logische Operatoren

- Damit ein `if`-Block ausgeführt wird, muss die Bedingung zwischen den runden Klammern wahr sein.
- Vergleichsoperatoren
  - → `a == b` (a ist gleich b), `a != b` (a ungleich b)
  - → `a < b` (a kleiner b), `a >= b` (a größer gleich b)
- logische Operatoren (Verknüpfung mehrerer Bedingungen)
  - → UND: `bedingung_1 && bedingung_2`
  - → ODER: `bedingung_1 || bedingung_2`
- Zum Beispiel können die zwei Bedingungen

```
if ( (a <= b) && (a >= b) )
    printf("a ist gleich b\n");
```

auch als eine Bedingung folgendermaßen geschrieben werden:

```
if (a == b)
    printf("a ist gleich b\n");
```

Übung 2: Maximum zweier ganzer Zahlen

Schreiben Sie ein C-Programm zur Bestimmung des Maximums zweier ganzer Zahlen. Die Eingabe der Zahlen soll über die Kommandozeile bei Ausführung des Programms erfolgen. Verwenden Sie für die Bestimmung des Maximums eine eigene Funktion.

Hauptprogramm `main`

- Typen-Umwandlung der Eingaben
- Aufruf der Funktion `maximum`
- Ausgabe des Maximums

Funktion `maximum`

- Parameter: zwei ganze Zahlen
- Rückgabewert: Maximum

Verzweigungen mit switch

- Zur Unterscheidung von vielen Fällen (ggf. sehr verschachtelte `if`-Anweisungen) verwenden wir `switch-case-Anweisungen`:

```
switch (Ausdruck)
{
    case konst_Ausdruck_1: /* Anweisungen_1 */ break;
    case konst_Ausdruck_2: /* Anweisungen_2 */ break;
    ...
    case konst_Ausdruck_n: /* Anweisungen_n */ break;
    default: /* Anweisungen_default */
}
```

- Verschiedene Fälle werden mittels `case` unterschieden.
- Jeder `case`-Block beginnt mit einem Doppelpunkt (`:`) und endet mit dem `break`-Befehl (verlässt `switch`-Anweisung).
- Gilt z. B. `Ausdruck == konst_Ausdruck_1`, so werden `Anweisungen_1` ausgeführt.
- Wird kein Fall erreicht, wird der `default`-Block ausgeführt.

Verzweigungen mit switch – ein Beispiel

Was macht folgendes Programm?

```c
#include <stdio.h>

int main (void)
{
    int a, s1=1, s2=3;
    a = 2;

    switch (a)
    {
        case 1: printf("Ich gebe etwas aus.\n"); break;
        case 2: s1 = s1 + s2; break;
        default: printf("a ist weder 1 noch 2. Was nun?\n");
    }

    return 0;
}
```

Überblick

Die `for`-Schleife

- Für Wiederholungen verwenden wir Schleifen.
- `for`-Schleifen sind zählergesteuert (Anzahl der Durchläufe ist bekannt), d. h. die Schleife wird solange ausgeführt, wie eine Schleifenvariable (der Zähler) eine Bedingung erfüllt:

```
for (Initialisierung; Bedingung; Aenderung)
{
    /* Anweisungen */
}
```

- **Vor dem 1. Durchlauf:** Initialisierung einer Schleifenvariablen
- **Vor Ausführung des `for`-Blocks:** Bedingung prüfen
- **Nach Ausführung des `for`-Blocks:** `Aenderung` durchführen, z. B. für eine Schleifenvariable `i`
  - → `i++` oder `i--` Der Wert der Variablen `i` wird nach jedem Durchlauf um 1 erhöht (`i++`) oder verringert (`i--`)
  - → `i=i+3` bzw. `i+=3` zum Erhöhen um 3

Die for-Schleife – ein Beispiel

Wir wollen die ersten 100 Fibonacci-Zahlen ausgeben.

```c
#include <stdio.h>

int main (void)
{
    int fib1=1, fib2=1, fib_tmp;
    int i;

    /* Ausgabe der ersten beiden Fibonacci-Zahlen */
    printf("\n%4d\n", fib1);
    printf("%4d\n", fib2);

    /* Berechnung und Ausgabe der 3. bis 100. Fibonacci-Zahl */
    for (i=3; i <= 100; i++)
    {
        fib_tmp = fib1 + fib2;
        fib1    = fib2;
        fib2    = fib_tmp;
        printf("%4d\n",fib_tmp);
    }

    return 0;
}
```

Die `while`-Schleife

- Für Wiederholungen verwenden wir Schleifen.
- `while`-Schleifen werden verwendet, wenn die Anzahl der Durchläufe nicht bekannt ist, jedoch Anweisungen solange ausgeführt werden sollen wie eine Bedingung erfüllt ist:

```
while (Bedingung)
{
    /* Anweisungen */
}
```

- **Vor Ausführung des `while`-Blocks:** Bedingung prüfen
- Ist die Bedingung wahr, so werden die Anweisungen im `while`-Block (Schleifenrumpf) ausgeführt.
- *Hinweis:* Es können auch **mehrere Bedingungen** (mithilfe logischer Operatoren) verknüpft werden.

Die `while`-Schleife – ein Beispiel

Wir wollen alle Fibonacci-Zahlen kleiner 1000 ausgeben.

```c
#include <stdio.h>

int main (void)
{
    int fib1=1, fib2=1, fib_tmp;
    int i;

    /* Ausgabe der ersten beiden Fibonacci-Zahlen */
    printf("\n%4d\n", fib1);
    printf("%4d\n", fib2);

    /* Berechnung und Ausgabe aller Fibonacci-Zahlen < 1000 */
    while ( (fib1 + fib2) < 1000 )
    {
        fib_tmp = fib1 + fib2;
        fib1    = fib2;
        fib2    = fib_tmp;
        printf("%4d\n",fib_tmp);
    }

    return 0;
}
```

Die `do while`-Schleife

- Für Wiederholungen verwenden wir Schleifen.
- `do while`-Schleifen sind `while`-Schleifen sehr ähnlich:
  - → Anzahl der Schleifen-Durchläufe ist nicht bekannt
  - → Anweisungen sollen solange ausgeführt werden wie eine Bedingung erfüllt ist

```
do
{
    /* Anweisungen */
}
while (Bedingung)
```

- Unterschied zu `while`-Schleifen: Bedingung(en) wird/werden **nach** Ausführung des Schleifen-Blocks geprüft
  ⇒ Die Schleife wird mindestens einmal ausgeführt.
- *Hinweis:* Es können auch **mehrere Bedingungen** (mithilfe logischer Operatoren) verknüpft werden.

Übung 3: Ausgabe von Hauptfunktions-Parametern

Schreiben Sie ein C-Programm, das die Anzahl und Werte der Parameter der Hauptfunktion sowie die Zeichenanzahl der einzelnen Paramter-Werte ausgibt.

Hinweise

- Geben Sie die Parameter als Zeichenketten aus (Formatierungstyp `%s`; keine Typen-Umwandlung)
- Zur Bestimmung der Zeichenanzahl einer Zeichenkette können Sie die Funktion

```
size_t strlen (const char *str)
```

aus der Bibliothek `string.h` verwenden: Z. B.

```
int anzahl = (int) strlen("123")
```

entspricht der Zuweisung `anzahl = 3`.
(`(int)` dient zur Typen-Umwandlung des Rückgabewertes.)

Überblick

Zeiger (Pointer)

- Zeiger = Variablen, deren Wert eine **Adresse** im Speicher ist.
- **Idee:** Zeiger "zeigen" auf eine Stelle im Speicher, an der eine "normale" Variable abgespeichert ist.
- Zeigervariablen werden in C deklariert, indem wir einen **Stern** (∗) **vor den Variablennamen** schreiben:

```
Datentyp *einZeiger;
```

  → `einZeiger` ist ein Zeiger, der auf eine Variable vom Typ `Datentyp` "zeigen" kann

- Für die Wertzuweisung eines Zeigers (das "Zeigen") brauchen wir die Adresse einer Variablen. Dazu schreiben wir ein **Kaufmanns-Und** (&) **vor den Variablennamen**:

```
Datentyp  eineVariable;
Datentyp *einZeiger;
einZeiger = &eineVariable;
```

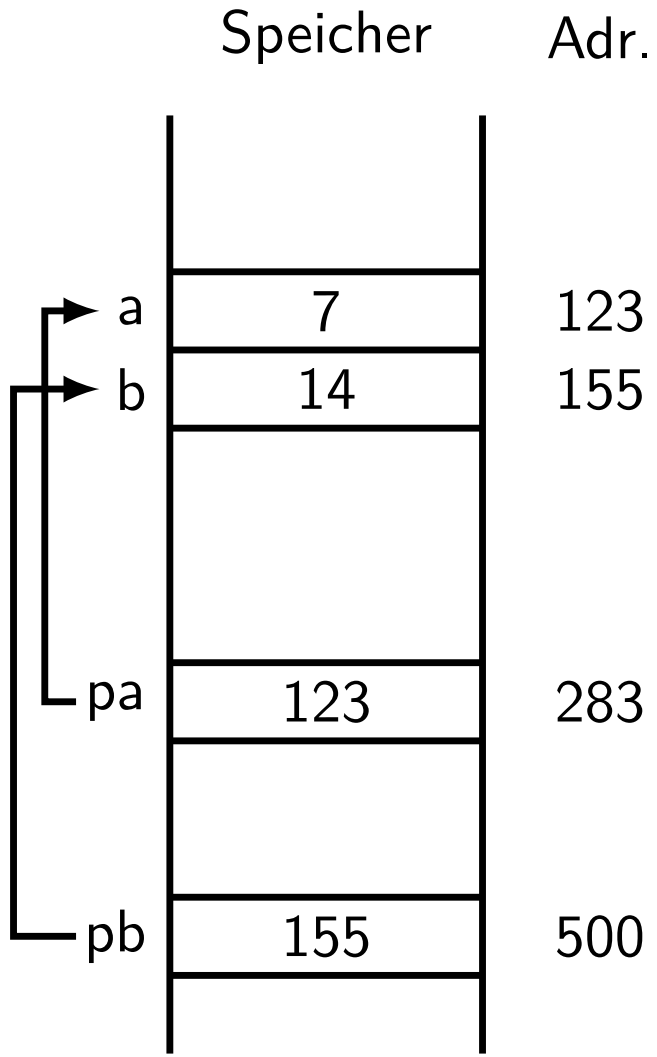
  → `einZeiger` "zeigt" auf die Variable `eineVariable`

Anschauliche Interpretation von Zeigern

```
int a = 7, b = 14;
int *pa, *pb;

pa = &a;
pb = &b;
```

- pa "zeigt" auf a
- pb "zeigt" auf b



Verwendung von Zeigern

Ziel: Vertauschung zweier ganzer Zahlen mithilfe einer Funktion

Listing 2: Vertauschung zweier ganzer Zahlen (1. Versuch)

```
1  #include <stdio.h>
2
3  void tauschen (int a, int b)
4  {
5      int tmp=a;
6      a = b;
7      b = tmp;
8      printf("tauschen: a = %d, b = %d\n", a, b);
9  }
10
11 int main (void)
12 {
13     int a=7, b=14;
14     tauschen(a,b);
15     printf("main: a = %d, b = %d\n", a, b);
16     return 0;
17 }
```

Was wird in den Zeilen 8 und 15 ausgegeben?

Was läuft im 1. Versuch schief?

```
/* ... */
void tauschen (int a, int b)
{
    int tmp=a;
    a = b;
    b = tmp;
}

int main (void)
{
    int a=7, b=14;
    tauschen(a,b);
/* ... */
```

- In den Funktionen `tausche` und `main` gibt es jeweils die **lokalen Variablen** `a` und `b`
  - → in `tausche` werden die Werte von `a` und `b` getauscht
  - → in `main` bleiben die Werte von `a` und `b` unverändert

Zeiger umgehen das Problem von lokalen Variablen

Ziel: Vertauschung zweier ganzer Zahlen mithilfe einer Funktion

Listing 3: Vertauschung zweier ganzer Zahlen mit Zeigern

```
1  #include <stdio.h>
2
3  /* Die Parameter von tauschen sind jetzt Zeiger auf int.
4     tauschen(&a, &b): pa zeigt auf a und pb zeigt auf b. */
5  void tauschen (int *pa, int *pb)
6  {
7      int tmp = *pa;
8      *pa = *pb;
9      *pb = tmp;
10 }
11
12 int main (void)
13 {
14     int a=7, b=14;
15     /* Uebergabe der Adressen von a und b an tauschen. */
16     tauschen(&a, &b);
17     printf("a = %d, b = %d\n", a, b);
18     return 0;
19 }
```

Vorteile von Zeigern

- Mithilfe von Zeigern können Variablen, die in einer Funktion deklariert wurden, in einer anderen Funktion geändert werden.
  $\rightarrow$ Verbesserung der Übersichtlichkeit bei langen Quellcodes
- Speicherbedarf von Zeigern: 64 Bit (64 Bit System) **unabhängig vom Datentyp**
  $\rightarrow$ effiziente Speichernutzung möglich

Überblick

Felder (Arrays)

- Zur Speicherung von Daten des gleichen Typs
- Arrays liegen als **zusammenhängender Block** im Speicher.
- Unterschied zu "normalen" Variablen: Arrays besitzen einen Index. **Achtung:** In C beginnt der Index bei 0! Zu einem Feld der Länge 3 gehören also die Indizes 0, 1 und 2.
- Die Größe eines Arrays (d. h. die Anzahl Einträge) schreiben wir in **eckigen Klammern hinter den Variablennamen**:

```
Datentyp ein1dArray[Laenge];
```

- Felder können ein-, zwei- oder auch mehrdimensional sein, z. B.

```
int    ein1dArray[3];
double ein2dArray[10][5];
```

  → `ein1dArray` ist ein eindimensionales Feld der Länge 3 vom Typ `int` (z. B. ein Vektor mit 3 Einträgen)
  → `ein2dArray` ist ein zweidimensionales Feld mit 10∗5 Elementen vom Typ `double` (z. B. eine 10×5-Matrix)

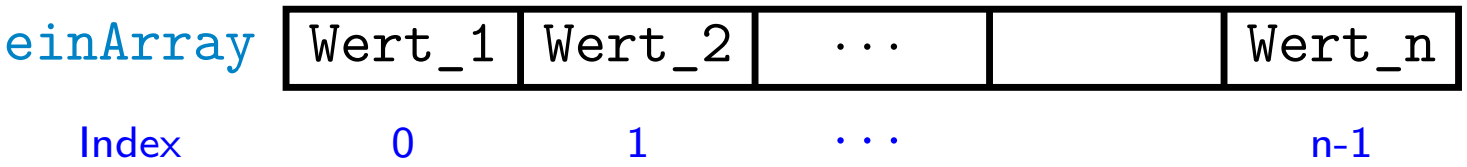
Initialisierung von eindimensionalen Feldern

- Zur Initialisierung der Werte eines Arrays schreiben wir die Werte einfach in **geschweifte Klammern**:

```
Datentyp einArray[Laenge] = {Wert_1, ..., Wert_n};
```

  → reserviert einen zusammenhängenden Speicherblock für `Laenge` Variablen vom Typ `Datentyp`

  → Initialisiert die Einträge 0 bis n des Arrays `einArray` mit den Werten `Wert_1`, ..., `Wert_n`



Hinweis: Ist `Laenge` $<$ `n`, so werden die restlichen Werte auf 0 gesetzt.

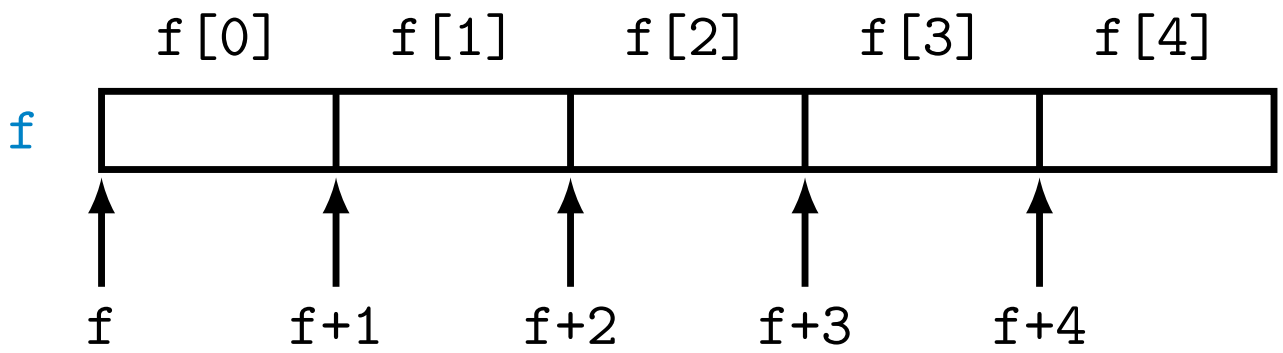
- `Datentyp einArray[] = {Wert_1, ..., Wert_n};`

  setzt die Größe des Arrays automatisch auf `n`

Feld-Zugriffe bei eindimensionalen Feldern

- Zum Zugriff auf die Werte eines Arrays verwenden wir **eckige Klammern**: `f[i]` liefert den Wert des (i+1). Eintrags des Arrays `f` (*Erinnerung:* Der Index beginnt bei 0.)
- Intern werden Zugriffe mithilfe von Zeigern umgesetzt: Der Feldname ist intern ein Zeiger auf den 1. Eintrag des Feldes

  → `f[i]` ist äquivalent zu `*(f + i)`



Zweidimensionale Felder – Definition und Zugriff

- Definition

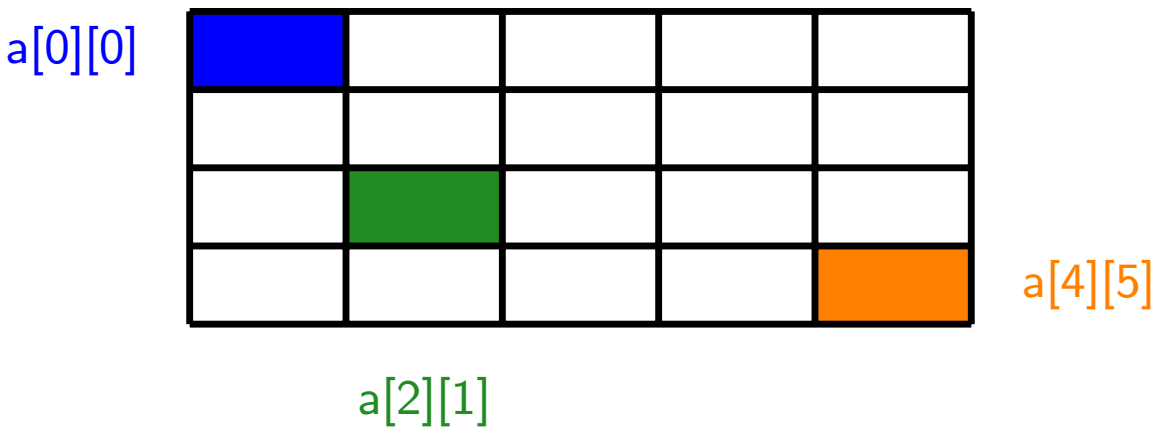
```
Datentyp Name[Groesse_1][Groesse_2];
```

  → `Name` ist ein zweidimensionales Array vom Typ `Datentyp[Groesse_1][Groesse_2]`

  → z. B. double a[4][5];

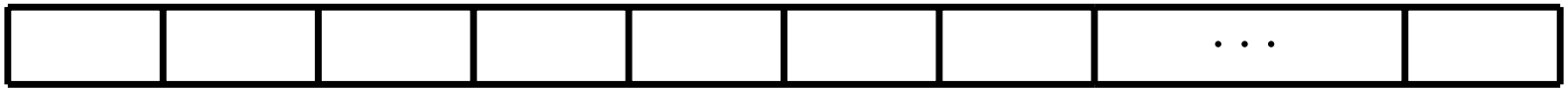
- Zugriff auf die Komponenten über **Indexoperator "[...]"**

  → ein Klammerpaar [...] je Dimension



Zweidimensionale Felder – interne Umsetzung

- Lineare Ablage der Komponenten im Speicher "**zeilenweise**"
  - → "letzter" Index läuft "am schnellsten"
  - → "erster" Index läuft "am langsamsten"

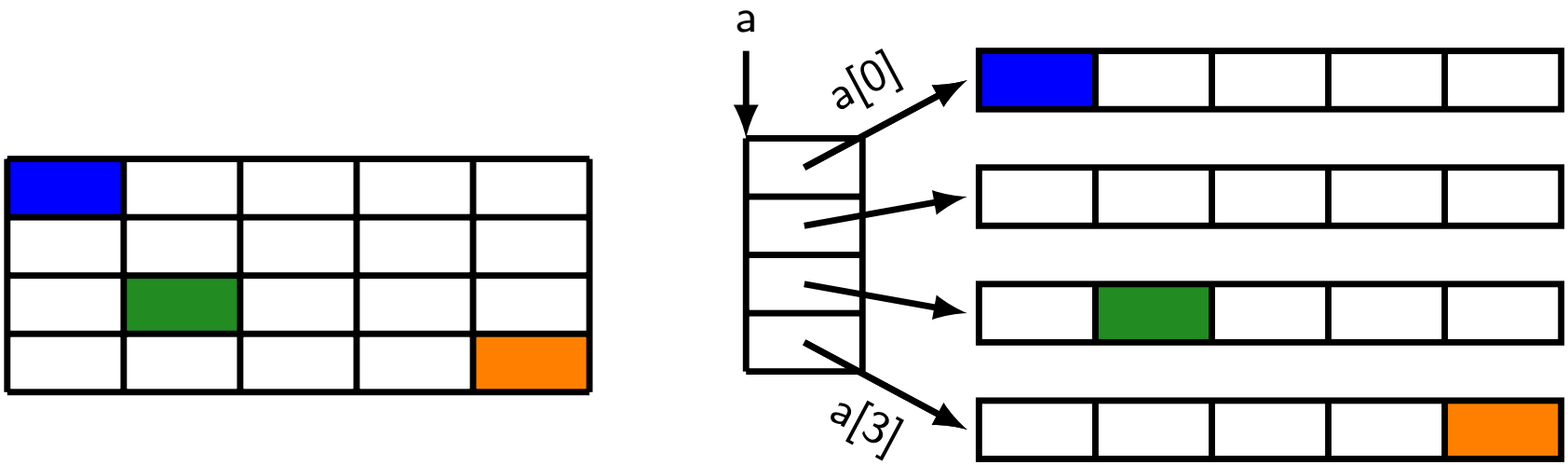


a[0][0] a[0][1] a[0][2] a[0][3] a[0][4] a[1][0] a[1][1] a[4][5]

- Was macht C aus dem Zugriff `a[ i ][ j ]`?

$$\begin{aligned} \texttt{a[i][j]} &= \texttt{(a[i])[j]} \\ &= \texttt{*((a[i]) + j)} \\ &= \texttt{*(*(a + i) + j)} \end{aligned}$$

Zweidimensionale Felder – anschauliche Interpretation



- Intern ist `a[ i ]` ein Zeiger auf die `i`-te Zeile des Feldes
- Die `i`-te Zeile des Feldes ist wieder ein Feld – also ein Zeiger

→ Zweidimensionale Felder sind intern also **Zeiger auf Zeiger**

Zweidimensionale Felder – Initialisierung

- Initialisierung über **Listenoperator "{...}"**
- Initialisierliste in zeilenweiser Reihenfolge:

```
Typ Name[n][m] = {Wert_11, ..., Wert_1m, Wert_21, ...
                  Wert_n1, ..., Wert_nm};
```

- Als **bessere lesbare Alternative** können auch **geschachtelte Initialisierlisten** verwendet werden:

```
Typ Name[n][m] = { {Wert_11, ..., Wert_1m},
                   {Wert_21, ..., Wert_2m},
                   ...
                   {Wert_n1, ..., Wert_nm} };
```

Speicherverwaltung

Problem: Oft ist die Anzahl der Elemente eines Arrays nicht a priori (d. h. zum Zeitpunkt des Kompilierens) bekannt

$\Rightarrow$ Feld fester Größe schlecht

- Platzverschwendung, falls Anzahl überschätzt wurde
- Überlauf des Feldes falls Unterschätzung

Abhilfe: Dynamische Speicherreservierung

In C reservieren wir mittels der Funktion `malloc` zur Laufzeit des Programms Speicherplatz

```
#include <stdlib.h>
void *malloc (size_t size)
```

$\rightarrow$ reserviert einen `size` Bytes großen Speicherbereich

$\rightarrow$ liefert einen Zeiger auf den reservierten Speicherbereich zurück

Speicherverwaltung – dynamische Speicherreservierung

```
#include <stdlib.h>
void *malloc (size_t size)
```

→ reserviert einen `size` Bytes großen Speicherbereich

- `size_t` ist ein vorzeichenloser Ganzzahl-Datentyp definiert in `stddef.h` (enthalten in `stdlib.h`)
- `sizeof` (Typ) liefert den **Speicherbedarf** eines Typs

→ liefert einen Zeiger auf den reservierten Speicherbereich zurück

- `void *` ist ein Zeiger auf einen **beliebigen Datentyp**
  ⇒ mit explizitem Cast Basis-Typ spezifizieren
- im Fehlerfall wird der `NULL`-Zeiger zurückgeliefert

Dynamische Speicherreservierung für ein `int`-Array `f` der Länge 5:

```
/* Zeiger definieren */
int *f;

/* Speicher reservieren und Zeiger "f" auf Anfang setzen. */
f = (int *) malloc (5*sizeof(int));
```

Dynamische Speicherreservierung – eindimensionale Arrays

```
#include <stdlib.h>
void free (void *ptr)
```

→ gibt den Block, auf den `ptr` zeigt, wieder frei

→ Diese Funktion **nur** auf mit `malloc()` erhaltende Zeiger anwenden, und **nur einmal**!

Für unser `int`-Array `f` der Länge 5 z. B.:

```
/* Zeiger definieren */
int *f;

/* Speicher reservieren und Zeiger "f" auf Anfang setzen. */
f = (int *) malloc (5*sizeof(int));

/* ... */

/* Speicher freigeben */
free (f);
```

Dynamische Speicherreservierung – zweidim. Arrays

Erinnerung: Zweidimensionale Arrays sind Zeiger auf Zeiger

Speicherreservierung für ein `Datentyp`-Array der Größe `n`×`m`:

```
/* Zeiger auf Zeiger definieren */
Datentyp **Name;

/* Der "aeussere" Zeiger soll auf ein n-langes Array
   aus Datentyp-Zeigern zeigen. */
Name = (Datentyp *) malloc (n*sizeof(Datentyp *));

/* Name zeigt auf den ersten von n Datentyp-Zeigern
   Name[0] bis Name[n-1]. Diese sollen auf Datentyp-
   Arrays der Laenge m zeigen. */
for (i = 0; i < n; i++)
   Name[i] = (Datentyp *) malloc (m*sizeof(Datentyp));
```

Hinweis: Bei `free()` läuft's umgekehrt, d. h. erst Speicher auf den die Zeiger `Name[i]` zeigen, freigeben, dann `free (Name)`.

Übung 4: Fibonacci-Zahlen

Schreiben Sie ein C-Programm, das die ersten `n` Fibonacci-Zahlen in einem Array speichert und anschließend ausgibt, wobei `n` ein Eingabeparameter sein soll.

Hinweise

- Den Eingabeparameter `n` können Sie wahlweise mittels `scanf` oder als Parameter der Hauptfunktion implementieren.
- Negative Werte von `n` sollen ausgeschlossen werden; geben Sie in diesem Fall eine Fehlermeldung aus.
- Vergessen Sie nicht, den angeforderten Speicher wieder freizugeben.

Überblick

Nützliche Links

- C-Tutorials:
  - → `http://www.c-howto.de`
  - → `http://cprogramming.com/tutorial.html`
- Übersicht über Bibliotheken und Befehle in C und C++
  - → `http://www.cplusplus.com/reference/clibrary/`