Institut für Wissenschaftliches Rechnen
Technische Universität Braunschweig
Prof. Hermann G. Matthies, Ph. D.
Dr. Elmar Zander

Sommersemester 2010
14. November 2014

Weiterführendes Programmieren

Programmierwerkzeuge und Sortierungsalgorithmen
Aufgabenblatt 1

Übung 1: *Erste Schritte*

Aufgabe a) Um zu verstehen worum es in diesem Kurs geht, welche Lernziele es gibt und wie Sie an Lernmaterial kommen, lesen Sie das erste Kapitel "Allgemeine Hinweise" aus dem Begleittext. Den Begleittext finden Sie auf unserer Webseite unter `http://www.wire.tu-bs.de/ADV`. □

Aufgabe b) Lesen Sie Kapitel zwei "Die Arbeitsumgebung von UNIX" aus dem Begleittext. Machen Sie sich mit den Unix-Kommandos zum Auflisten, Umbenennen und Löschen von Dateien und Verzeichnissen, zum Erzeugen von Verzeichnissen, und dem `man`-System der Hilfeseiten vertraut. Wichtige Kommandos, die Sie kennen müssen: `man, ls, cp, mkdir, rm, mv, cd`. □

Aufgabe c) Gehen Sie auf die Internetseite `http://www.digilife.be/quickreferences/quickrefs.htm` und drucken Sie die "C Reference Card". Diese wird Ihnen im laufe des Kurses helfen. □

Aufgabe d) Arbeiten Sie das Kapitel 3 des Begleittextes durch. Tippen Sie die Programme selbst ein, um zu lernen, welche Zeichen beim C-Programmieren häufig verwendet werden und wo sie auf der Tastatur zu finden sind. □

Aufgabe e) Erstellen Sie einen Programmkopf, der später jedes Ihrer Programme einleiten soll, und speichern Sie ihn. Der zu erstellende Programmkopf ist ein Kommentartext, in dem stehen soll, wer Sie sind. Er könnte beispielsweise so aussehen wie in Listing 1 angegeben. □

```
1 /*****************************************************************
2  *
3  *  Claus Muster, Elektrotechnik, 4. Sem., Mat-Nr 2222345
4  *  Sabine Beispiel, Wirtschafts-ET, 4. Sem., Mat-Nr 2555678
5  *
6  *  3.5.2001
7  *
8  *****************************************************************/
```

Listing 1: Beispiel-Programmkopf

Bemerkung: Sie sollten für jede Aufgabe, die Sie im Rahmen dieses Praktikums lösen, ein neues Verzeichnis in Ihrem Dateisystem anlegen. Die Struktur wird ihnen dabei helfen, ihre Lösung voneinander zu trennen und die passenden Lösungen zu ihren Aufgaben wieder zu finden.
Wir empfehlen Ihnen, bei Programmierprojekten schon früh auf die Dokumentation in Englisch umzusteigen. Wenn sie deutsche Variablennamen verwenden, und später einmal ein Kollege aus dem Ausland, der vermutlich kein Deutsch spricht, in dem Projekt mitarbeitet, dann ist es sehr schwer und fehleranfällig, auf englische Variablennamen umzusteigen. Daher ist es besser, gleich Englisch zu verwenden. Im übrigen sieht die Mischung von englischen Programmiersprachenkonstrukten (if, while, return) mit deutschen Variablennamen seltsam aus.

Übung 2: *Kleine Programmänderung*
In dieser Aufgabe geht es um das Programm `helloWorld.c` aus dem Begleittext, an dem ein paar kleine Änderungen durchgeführt werden sollen.
Zum Einlesen einer Zeichenkette muss in einem Programm Speicherplatz für diese reserviert werden (siehe Kapitel 6.3 (6 Seiten) im Buch [Dausmann,Bröckl,Goll:2008]). Dies geschieht mit einer Zeile der Form

```
char name[10];
```

welche eine Variable `name` deklariert, die eine Zeichenkette (String) von bis zu 9 Zeichen enthalten kann. Man kann deshalb nur 9 Zeichen in der Zeichenkette speichern, da jede Zeichenkette in C durch ein `'\0'`-Symbol abgeschlossen werden muss. In den meisten Fällen, z.B. wenn man eine Zeichenkette im Code zuweist, setzt der Compiler dieses Symbol automatisch. Wenn Sie aber speicher für Zeichenketten explizit reservieren, so müssen Sie darauf achten, immer ein symbol “zu viel” zu reservieren.
Das Einlesen eines Namens geschieht mit der Zeile

```
scanf("%s",&name);
```

und die Ausgabe, indem der Aufruf von `printf` noch ein zusätzliches Argument bekommt:

```
printf("Hallo %s!\n",name);
```

Bemerkung: Beide C-Funktionen `printf` und `scanf` sind "formated I/O" (I=Input,O=Output) Funktionen; daher das `f` am Ende ihrer Namen. Der erste Parameter dieser Funktionen ist ein "*format string*", welcher das Format der Ein-/Ausgabe angibt. Die Zeichenfolge `%s` im *format string* gibt an, dass die ein-/auszugebende Variable als *string* (also als Zeichenkette `char*`) interpretiert werden soll. Eine Übersicht solcher *Formatzeichen* finden Sie auf ihrer "C Reference Card" im Abschnitt "Codes for Formatted I/O" in der Rubrik "Input/Output".

Wendet man die Funktion `scanf` an, so muss der Parameter der Funktion (hier `name`) mit einem "`&`" angegeben werden. Das "`&`" ist ein wichtiger Operator in C, der die Adresse der Variable rechts von ihm zurück liefert. Die Funktion `scanf` braucht diese Adresse und nicht den Wert der Variable (anders als z.B. `printf`), denn `scanf` soll einen neuen Wert dort speichern und nicht den alten Wert lesen.

Mit der Position der *Formatzeichen* gibt man an, wo die Werte der Variablen, die auf den *format string* folgen, in der Ausgabe erscheinen sollen, bzw. in der Eingabe erscheinen. Erscheinen mehrere Variablen in einer solchen formatierten Ein-/Ausgabe, so ist auf die Reihenfolge der Variablen im *format string* zu achten. Die auf den *format string* folgenden Variablen werden in der Reihenfolge interpretiert, welche die Formatsymbole im *format string* haben.
Details zu diesem Thema finden Sie in die Kapitel 14.1 bis 14.6 (5 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] .

Aufgabe a) Erweitern Sie nun das Programm `helloWorld.c` aus dem Begleittext im Kapitel 3 um eine persönliche Begrüßung. Dazu soll das Programm einen Namen einlesen, und die Person persönlich begrüßen. Benutzen Sie die oben angegebenen Funktionsaufrufe und Deklarationen. □

Aufgabe b) Geben Sie in das Programm einen Namen ein, der länger ist als 9 Zeichen. □

Bemerkung: Sie sehen, dass durch die Eingabe ein Fehler verursacht wird, der das Programm abstürzen lässt. Die Funktion `scanf` hat keine Information über die Größe des Speichers, der für `name` reserviert ist, daher überschreibt die Funktion Speicher, der nichts mit der Variable zu tun hat. Solche Fehler werden als Buffer overflows (`http://en.wikipedia.org/wiki/Buffer_overflow`) bezeichnet und können von *Hackern* für verschiedene Dinge ausgenutzt werden. Sie müssen daher immer darauf achten, was sie tun, wenn Sie sicherheitsrelevante Programme entwickeln. Das einfache Beispiel zeigt, wie wichtig es bei der Programmentwicklung ist, systematisch vorzugehen und Bibliotheken zu benutzen, die gut getestet und vertrauenswürdig sind.

In diesem Beispiel kann der Fehler behoben werden, indem der Funktion `scanf` die maximale Länge bekannt gemacht wird. Dies geschieht mit einer Begrenzung des Strings auf 9 Zeichen.

```
scanf("%9s",name);
```

Aufgabe c) Machen Sie die Änderung in ihrem Programm, und geben Sie in das Programm einen Namen ein, der länger ist als 9 Zeichen. Jetzt gibt es keine Probleme mehr. □

Aufgabe d) Erweitern Sie das Programm um eine Schleife, die weitere Namen einliest und die Perso-

nen begrüßt. Überlegen Sie sich ein Abbruchkriterium, zum Beispiel ein bestimmtes Zeichen zu Beginn eines Wortes. Erklärungen zu Schleifen finden Sie in Kapitel 8.3 (11 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] . □

Aufgabe e) Verwenden Sie die Eingabeumlenkung (Kapitel "Ein- und Ausgabeumlenkung" im Begleittext), um Ihr Begrüßungsprogramm anhand einer vorher in einer Datei gespeicherten Liste von Namen zu testen. □

Übung 3: *Werkzeuge der Softwareentwicklung*
Um systematisch Programme entwickeln zu können, müssen Sie einige Werkzeuge kennenlernen.
Aufgabe a) Arbeiten Sie das Kapitel 5 des Begleittextes "Werkzeuge der Softwareentwicklung" durch. □

Die in diesem Text beschriebenen Werkzeuge sollten Sie in den folgenden Aufgaben und Aufgabenblättern immer einsetzen.

Bemerkung: In Vorlesungen über Algorithmen sollten Sie verschiedene Sortierverfahren kennen gelernt haben. Hier können Sie diese Algorithmen nochmals implementieren und testen, die Laufzeiten mit verschiedenen Eingaben messen, und die angegebenen Komplexitäten O(n^2) und O($n \log n$) verifizieren. Wir werden in dieser Aufgabe Arrays benutzen, daher sollten Sie die Kapitel 6, 10.1 und 10.2 (16+17 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] gelesen haben.
Wir stellen ein Programm bereit, das Eingaben für ein Sortierprogramm erzeugt: Das Programm ist unter `http://www.wire.tu-bs.de/ADV/files/sortdata/sortdata.c` zu finden. Kompilieren Sie das Programm mit `make sortdata`.
Bei Aufruf von

```
./sortdata -f 10
```

sollten Sie eine Ausgabe von der Zahl 10 bekommen, gefolgt von 10 Zahlen, die annähernd (aber nicht vollständig) sortiert sind. Bei dem Aufruf

```
./sortdata 100000 testfile
```

wird eine Datei `testfile` erzeugt, die zunächst die Zahl 100000, gefolgt von 100000 zufälligen Zahlen enthält.
Das Programm hat weitere Optionen, die man erfährt, wenn man `./sortdata` ohne Argumente aufruft.

Übung 4: *Sortieren*
Aufgabe a) Lesen Sie Kapitel 15 (5 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] und die Kapitel 14.1 bis 14.6 (5 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] und rufen Sie sich die Kapitel 6, 10.1 und 10.2 (16+17 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] und Kapitel 9.3 bis 9.5 (13 Seiten) im Buch [Dausmann,Bröckl,Goll:2008] in Erinnerung.
Schauen Sie sich den Code `sortdata.c` in Ruhe an und versuchen Sie jede Zeile zu verstehen. Schauen Sie sich vor Allem an, wie in dem Code die Kommandozeilenparameter interpretiert werden. □

Aufgabe b) Laden Sie die Dateien unter `http://www.wire.tu-bs.de/ADV/files/data_io` herunter und versuchen Sie den Code zu verstehen.

```c
#include<stdio.h>
/************************************************/
/* function to write an integer array into a file */
/* args: 1. filepointer to write to               */
/*       2. integer array                         */
/*       3. length of the array                   */
/************************************************/
void write(FILE*, int*, int);
/************************************************/
/* function to read an integer array from file    */
/* args: 1. filepointer to read from              */
/*       2. pointer to integer array              */
/*          (has to point to NULL)                */
/*       3. pointer to integer will be filled     */
/*          with the length of the read array     */
/* return: 0 on success, -1 on error              */
/************************************************/
int read (FILE*, int**, int*);
```

Listing 2: io.h

Bemerkung: Wichtig sind hier vor Allem die Funktionen `write` und `read`, die in Listing 2 deklariert sind. Die Implementierung in der Datei `io.c` sieht auf Grund der Parametrisierung relativ kompliziert aus. Das liegt daran, dass erst in der Funktion der Speicher für das Array reserviert werden kann, da die Größe aus der Datei eingelesen werden muss.
Die Funktionen erlauben es Ihnen, Arrays von Ganzzahlen auf standardisierte Weise in eine Datei zu schreiben und aus einer Datei zu lesen.
Benutzen Sie die gegebenen Funktionen in der Lösung der folgenden Aufgaben.

Bemerkung: Zeigerarithmetik ist eine wichtige Technik in C. Man kann mit Hilfe der Zeigerarithmetik mit Zeigern rechnen.

```
1  int read(FILE* fp, int** data, int* n)
2  {
3  ...
4     for(i=0;i<(*n);i++)                 // loop over the entries
5    {
6     if(!is_eof(fp))
7     {
8       fscanf(fp,"%i",(*data)+i);        // read entry i
9     }
10 ...
11 }
```

Listing 3: io.c mit Zeigerarithmetik

Das Codefragment aus Listing 3 ist der Datei `io.c` entnommen und zeigt in Zeile 8 ein Beispiel für die Nutzung der Zeigerarithmetik. Hier wird mit `(*data)` zunächst der *Zeiger auf einen Zeiger auf Integer* mit dem Namen `data` dereferenziert; aus dem *Zeiger auf einen Zeiger* wird also ein *Zeiger*. Dieser Zeiger zeigt auf den ersten Eintrag eines Arrays, das mit `calloc` angelegt wurde. Durch die Addition von i, also `(*data)+i` wird der Zeiger um `i` Einträge erhöht und zeigt damit auf die Speicherstelle mit der Adresse `(*data)+i`. Diese Adresse wird in diesem Code an `fscanf` übergeben, welches dort den gelesenen Wert speichert. In Abbildung 1 auf Seite 11 sind nochmals ein paar Beispiele für den Umgang mit Zeigern illustriert. Zeigerarithmetik sollte nicht exzessive benutzt werden, sondern nur dort, wo es unbedingt nötig ist. Bei der Sortierung müssen Sie mit dieser Technik umgehen können.

□

Aufgabe c) Schreiben Sie ein Programm `sort`, das die von `sortdata` erzeugten Daten einliest, sie sortiert und in sortierter Reihenfolge in eine Ausgabedatei schreibt. Die Ausgabedatei soll das gleiche Format haben, wie die von `sortdata` erzeugten Dateien (benutzen Sie `write`).
sort und stellen Sie Kommandozeilenparameter zur Verfügung, mit denen ein Verfahren ausgewählt werden kann (`-i` für Insertionsort,`-q` für Quicksort). Diese Verfahren werden auf den nächsten Seiten präsentiert.

□

Bemerkung: Hier eine kurze Zusammenfassung der Algorithmen. Die Darstellung folgt dem Buch *Introduction to Algorithms* von *T.H.Cormen et al.*:

Der Insertionsort Algorithmus

INSERTION-SORT(A)

```
1  for j ← 2 to length[A]
2      do key ← A[j]
3         //Insert A[j] into the sorted sequence A[1 . . j − 1].
4         i ← j − 1
5         while i > 0 and A[i] > key
6             do A[i + 1] ← A[i]
7                i ← i − 1
8         A[i + 1] ← key
```

Warnung: Die Algorithmen in *T.H.Cormen et al.* benutzen Arrays nicht (wie in C üblich) mit Element 0 startend, sondern mit Element 1. Diese Indexverschiebung findet man sehr häufig in Pseudocode.

Der Quicksort Algorithmus

QUICKSORT(A, p, r)

```
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q)
4          QUICKSORT(A, q + 1, r)
```

PARTITION(A, p, r)

```
1   x ← A[p]
2   i ← p − 1
3   j ← r + 1
4   while TRUE
5       do repeat j ← j − 1
6            until A[j] ≤ x
7          repeat i ← i + 1
8            until A[i] ≥ x
9          if i < j
10           then exchange A[i] ↔ A[j]
11           else return j
```

Bemerkung: Näheres zu den Algorithmen findet sich unter den angegebenen Adressen auf Wikipedia:

Insertionsort `http://de.wikipedia.org/wiki/Insertionsort`

Quicksort `http://de.wikipedia.org/wiki/Quicksort`

Aufgabe d) Schreiben Sie ein Programm `test_sorted`, das eine Datei in dem gemeinsamen Dateiformat von `sortdata` und `sort` einliest und testet, ob die Zahlensequenz sortiert ist. □

Aufgabe e) Testen Sie Ihre Sortierroutinen mit null, einer, zwei, und dann mit drei beliebigen Zahlen. Dies überprüft die einfachen Grenzfälle des Algorithmus. □

Aufgabe f) Ein weiterer wichtiger Grenzfall ist die Sortierung einer bereits sortierten Liste. Testen Sie Ihre Sortierroutinen daher mit den Zahlen 1 bis 10 schon aufsteigend bzw. absteigend sortiert, sowie mit

10 gleichen Zahlen. □

Aufgabe g) Testen Sie die Sortierroutinen mit jeweils 3 verschiedenen Datensätzen aus dem Programm `sortdata`. □

Übung 5: *Zeiten messen*

Aufgabe a) Bauen Sie in ihr Sortierprogramm einen weiteren Kommandozeilenparameter `-t` ein, der bewirken soll, dass die Ausführungszeit der Sortieralgorithmen anstelle des Ergebnisses ausgegeben wird. **Messen Sie nur die Zeiten zum Sortieren, ohne Ein- und Ausgabe.** □

Bemerkung: Zur Zeitmessung benötigen Sie eine Stoppuhr, diese stellen wir Ihnen unter `http://www.wire.tu-bs.de/ADV/files/stopwatch` zur Verfügung. Nach dem Entpacken des Paketes sollten Sie zunächst mit `make test` überprüfen, ob die Stoppuhr funktioniert. Dieser Aufruf erzeugt auch schon eine Bibliothek `libstopwatch.a`, welche Sie in ihrem Programm benutzen sollen. Die Bedienung der Stoppuhr ist sehr einfach:

`start_timer()` startet die Stoppuhr

`pause_timer()` pausiert die Stoppuhr

`read_timer()` liest die Zeit ab

```
1 #include <stopwatch.h>
2 int main(){
3   start_timer();
4   printf("Already %fs gone!\n",read_timer());
5 }
```

Listing 4: Stopwatch Benutzung

In Listing 4 sehen Sie ein Beispiel für die korrekte Benutzung. Das Beispiel befindet sich ebenfalls in dem Paket `stopwatch.zip` im Unterverzeichnis `usage_demo`. In diesem Verzeichnis befindet sich auch ein Makefile, das die korrekten Parameter für Compiler und Linker zur Nutzung der Bibliothek enthält.

Aufgabe b) Nun testen Sie die Laufzeiten der Sortierroutinen mit den verschiedenen Sortierverfahren und verschiedenen Größen der zu sortierenden Zahlenmenge. □

Bemerkung: Im Folgenden sollen Sie Messreihen für die Dauer der Sortierung mit den unterschiedlichen Verfahren durchführen. Es ergeben sich mehrere Messreihen:

- Quicksort und Insertionsort
- mit Option -v, mit Option -r, ohne Option bei `sortdata`.
- n=100, n=500, n=1000, n=1500, n=2000, n=2500, etc. bis die Laufzeiten zu lang werden.

Um einfacher jede Kombination ausprobieren zu können, bietet es sich an, die Daten gar nicht zwischenzuspeichern, sondern gleich die Ausgabe von `./sortdata` als Eingabe ihres Sortierprogramms zu verwenden, also etwa

```
./sortdata -r 10000 | ./sort -q
```

Aufgabe c) Erstellen Sie eine graphische Darstellung Ihrer Zeitmessungen. Diskutieren Sie die Gründe, warum sich die verschiedenen Sortierverfahren bei unsortierten und vorsortierten Daten so unterschiedlich verhalten. Warum ist Quicksort bei falschrum sortierten Daten schneller als bei unsortierten? Spielt das $\log(n)$ in $O(n \log n)$ eine Rolle?

Bemerkung: Die graphische Darstellung können Sie am Besten mit dem Werkzeug `gnuplot` machen. Dazu benutzen Sie den Aufruf

```
echo plot \"Messreihe.txt\" | gnuplot -persist
```

Wobei die Datei `Messreihe.txt` folgendes Format haben sollte:

```
10  0.001
20  0.002
50  0.015
100 0.03
```

Listing 5: Beispiel einer Eingabedatei für Gnuplot

Mit dem Aufruf

```
echo plot \"Reihe1\",\"Reihe2\" | gnuplot -persist
```

können sie zwei Messreihen im gleichen Bild darstellen lassen; analog geht das natürlich auch mit mehreren. Falls Sie mehr zu GNUplot wissen wollen, schauen Sie auf `http://www.gnuplot.info/`.

Bemerkung: Bei der Interpretation der Messreihen werden Sie sehen, dass die Ausführungszeit für die Algorithmus stark von deren Eingabe abhängen. Man unterscheidet die folgenden Varianten zur Laufzeitabschätzung:

- Die worst-case-Laufzeit (engl. schlechtester Fall) gibt an, wie lange der Algorithmus maximal braucht. Für viele Algorithmen gibt es nur wenige Eingaben, die diese worst-case-Laufzeit erreichen, weshalb sie nicht unbedingt eine realistische Abschätzung ist. Handelt es sich aber um Echtzeitsysteme, so muss die worst-case-Laufzeit natürlich berücksichtigt werden.
- Die average-case-Laufzeit (engl. durchschnittlicher Fall) gibt die erwartete Laufzeit bei einer gegebenen Verteilung der Eingaben an. Da allerdings die Verteilung der Eingaben bei Programmen nicht immer bekannt ist, ist die Berechnung der average-case-Laufzeit in diesen Fällen nur unter einschränkenden Annahmen möglich. Siehe auch: Amortisierte Laufzeitanalyse
- Die best-case-Laufzeit (engl. bester Fall) gibt an, wie lange der Algorithmus in jedem Fall braucht, also selbst für die Eingaben, auf denen er ideal arbeitet. Diese untere Schranke zur Lösung des Problems wird nur selten angegeben, da sie nur für wenige Fälle zutrifft und die best-case-Laufzeit in der für die schlechteren Fälle enthalten ist.

Für mehr Informationen zu diesem Thema schauen Sie bitte auf `http://de.wikipedia.org/wiki/Zeitkomplexit\protect\unhbox\voidb@x\bgroup\U@D1ex{\setbox\z@\hbox{\char127}\dimen@-.45ex\advance\dimen@\ht\z@}\accent127\fontdimen5\font\U@Da\egroupt`.

□

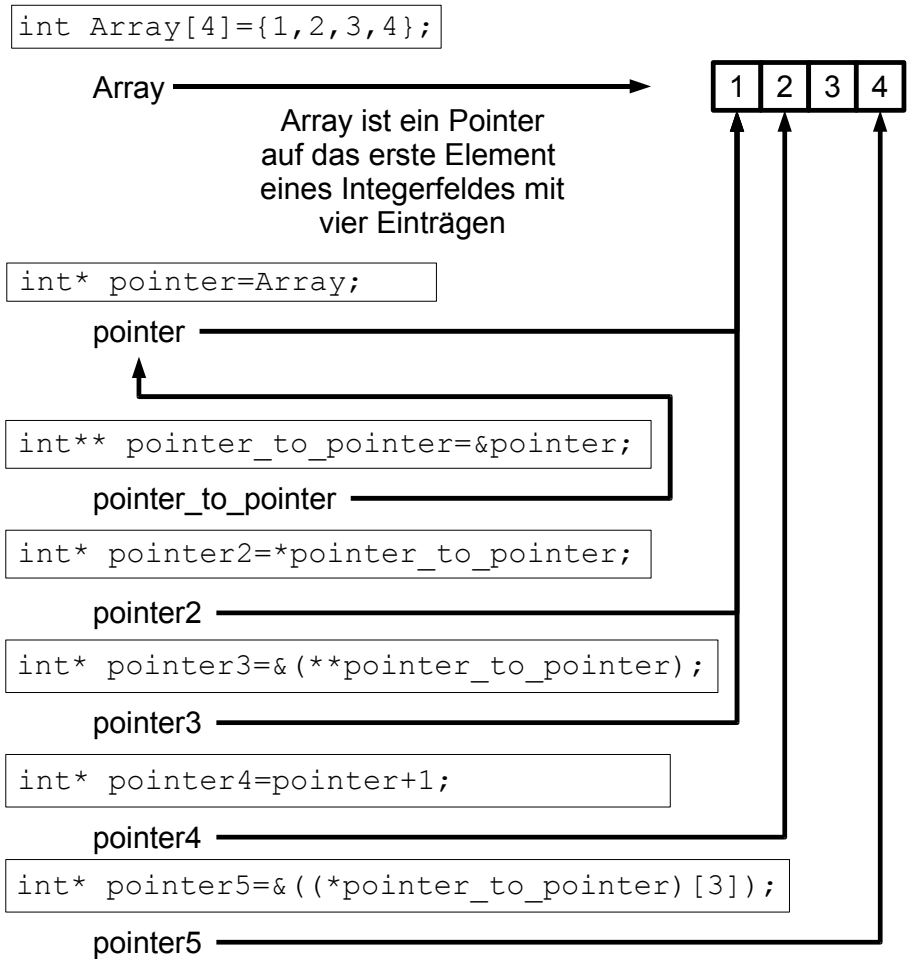

Abbildung 1: Das Bild illustriert einige komplexere Beispiele für den Umgang mit Zeigern.