Programmieren in C

B. Kostrzewa, F. Pittler, M. Ueding, C. Urbach

27. März 2020

Inhaltsverzeichnis

1 Grundlagen **5**

1.1 Einleitung . . . 5
1.1.1 Weitere Hilfe . . . 6
1.2 Ein erstes `C` Programm . . . 7
1.2.1 Daten- und Speichertypen . . . 8
1.2.2 Maschinenzahlen . . . 9
1.2.3 `C` Quelltext . . . 10
1.3 Variablen und Operatoren . . . 12
1.3.1 Deklaration und Initialisierung . . . 12
1.3.2 Sichtbarkeitsbereich . . . 14
1.3.3 Datentypen und Wertebereiche . . . 14
1.3.4 `C` Schlüsselwörter und Regeln für Namen . . . 15
1.3.5 Konstanten . . . 16
1.3.6 Operatoren . . . 16
1.3.7 Logische Ausdrücke . . . 18
1.4 Kontrollstrukturen: Verzweigungen und Schleifen . . . 20
1.4.1 Einfache Verzweigung: `if-else` . . . 20
1.4.2 Mehrfache Verzweigung: `switch` . . . 21
1.4.3 Schleifen: `for` . . . 23
1.4.4 Schleifen: `while` und `do-while` . . . 25
1.5 Funktionen . . . 26
1.5.1 Funktionen Prototypen . . . 26
1.5.2 Funktionen Definition . . . 26
1.5.3 Aufruf von Funktionen . . . 27
1.5.4 Typenüberprüfung und sinnvolle Eingabewerte . . . 28
1.5.5 Rückgabetypen und `void` Funktionen . . . 29
1.5.6 Funktionsaufrufe mit vielen Argumenten . . . 29
1.6 Felder (arrays) und Zeichenketten . . . 30
1.6.1 Deklaration und Initialisierung eindimensionaler Felder . . . 30
1.6.2 Eindimensionale Felder als Funktionenargumente . . . 31
1.6.3 Mehrdimensionale Felder . . . 32
1.6.4 Zeichenketten oder Strings . . . 33
1.7 Die Bibliotheken `math.h` und `complex.h` . . . 38
1.7.1 Mathematische Bibliothek `math.h` . . . 38
1.7.2 Komplexe Zahlen mit `complex.h` . . . 39

2 Zeiger und Co **43**
2.1 Zeiger . . . 43
2.1.1 Zeiger und Felder . . . 46
2.1.2 Die Parameter der `main` Funktion . . . 49
2.1.3 Zeigerarithmetik . . . 50
2.2 Ein- und Ausgabe . . . 53
2.2.1 Standard Ein- und Ausgabe . . . 53
2.2.2 Ausgabe in Dateien . . . 54
2.3 Anwendung: Einfügesortieren . . . 59
2.3.1 Header und Quelltextdateien . . . 61
2.4 Dynamische Speicherverwaltung . . . 64
2.4.1 Dynamische Speicherreservierung mit `malloc` . . . 64
2.4.2 Speicherfreigabe mit `free` . . . 65
2.4.3 Mehrdimensionale Arrays . . . 66
2.5 Komplexe Datentypen . . . 70
2.5.1 Eigene Datentypen mit `typedef` . . . 70
2.5.2 Zusammengesetzte Datentypen . . . 70
2.5.3 Beispiel: Datentyp für kartesische Koordinaten . . . 72

Index **75**

1 Grundlagen

1.1 Einleitung

In diesem Skript wird die Programmiersprache `C` eingeführt. `C` ist eine mächtige Programmiersprache, die in vielen Bereichen zum Einsatz kommt. Im Allgemeinen besteht Programmieren aus dem Schreiben von Quelltext (auch *Code*) in einer Programmiersprache, der dann in ausführbaren Maschinencode übersetzt werden muss. Den letzten Schritt nennt man Übersetzen (auch *compilieren*), und das Programm, das diese Aufgabe übernimmt *Compiler*.

`C` gehört zu den Programmiersprachen, für die zunächst der gesamte Quelltext geschrieben werden muss, um ihn dann zu Übersetzen. Der übersetzte Code kann dann ausgeführt werden, man spricht auch vom ausführbaren Programm. Es gibt mehrere Gründe, sich für `C` als Programmiersprache zu entscheiden, unter anderem:

- `C` erzeugt meist effizienten Maschinencode:
  Es gibt sehr gute Compiler und die Sprache `C` lässt sich sehr gut in Maschinencode übersetzen.

- `C` ist eine Hochsprache mit mächtigen Sprachelementen:
  Man muss nicht die Details der benutzten Computerarchitektur kennen, um guten Quelltext zu erzeugen.

- `C` ist sehr maschinennah:
  Wenn man doch einmal die Details beispielsweise der CPU ausnutzen möchte, so ist das möglich.

- `C` wird sehr häufig genutzt:
  Es gibt viele bestehende *Bibliotheken*, in denen nützliche Funktionalität implementiert ist und es ist relativ einfach, schnell Hilfe zu verschiedensten Problemen zu erhalten.

Viel der heute häufig genutzten Software ist ursprünglich in `C` geschrieben.

Der Inhalt dieses Skripts ist der folgende: Im ersten Teil werden die Grundlagen behandelt, mit deren Hilfe man einfache Fragestellungen in `C` umsetzen kann. Dabei beginnen wir mit der grundlegenden Struktur eines `C` Programms und erklären seine Bestandteile. Das Verständniss der folgenden Konzepte ist dabei fundamental:

1. Datentypen

2. Kontrollstrukturen

3. Funktionen

Will man bespielsweise eine Reihe von Zahlen sortieren, so muss man zunächst entscheiden, ob ganze oder reelle Zahlen sortiert werden sollen. Man muss sich also über den Datentyp klar werden. Auf jedem C Datentyp sind bestimmte Operationen definiert. Alle C-Datentypen und die dafür zur Verfügung stehenden Operationen werden eingeführt. Um den Ablauf eines Programms beeinflussen zu können, braucht man allerdings mehr als Operationen auf Daten, nämlich sogenannte C-Kontrollstrukturen wie Schleifen und Verzweigungen.

Ein weiteres wichtiges Konzept sind Funktionen. Im allgemeinen stellen Funktionen (im Idealfall kurze) mit Namen versehene Abschnitte im Programm dar, die dann über den Namen wieder aus dem Quelltext aufgerufen werden können. Das kleinste ausführbare C-Programm besteht aus genau einer Funktion. Funktionen bekommen Daten als Eingabe, führen bestimmte Operationen auf diesen Eingabedaten aus und liefern dann einen Rückgabewert. Die oben bereits erwähnten Bibliotheken stellen im wesentlichen Funktionen zur Verfügung. So wird zum Beispiel die Ein- und Ausgabe von Text in C über Bibliotheksfunktionen realisiert. Wir werden zeigen, wie man Funktionen definiert und aufruft.

Der zweite Teil des Skripts behandelt zwei fortgeschrittene Konzepte von C, nämlich dynamische Speicherverwaltung und zusammengesetzte Datentypen. Das Verständnis von dynamischer Speicherverwaltung in C kann in vielen Situationen unumgänglich sein. Denken Sie an einen Algorithmus zum Sortieren von Zahlen: Wenn nicht von Anfang an klar ist, welche Länge die zu sortierenden Zahlenreihen haben werden, dann sollte man in der Lage sein, diese Länge dynamisch anzupassen

Zusammengesetzte Datenstrukturen sind sehr nützlich, um auf ein Problem angepasste Datentypen nutzen zu können. Das ist unter anderem auch wichtig, weil damit unter Umständen Programme effizienter gemacht werden können. Ferner werden wir sehen, dass zusammengesetzte Datentypen dabei helfen können, den Quelltext für eine gegebene Problemlösung verständlicher zu gestalten.

Ein sehr wichtiger in diese Richtung gehender Aspekt beim Programmieren ist auch das Kommentieren sowie das Codedesign. Letzteres werden wir insbesondere in Form der sogenannten *modularen* Programmierung kennenlernen. Dies erhöht die Wiederverwendbarkeit von Code und erlaubt es auch Code zu verstehen, den man vor einem Jahr geschrieben hat.

Da sich die Sprache C im Laufe der Zeit verändert, existieren verschiedene sogenannte Standards. In diesem Skript werden wir uns an den C99 Standard halten.

1.1.1 Weitere Hilfe

Dieses Skript erhebt nicht den Anspruch, die Programmiersprache C in allen Details und vollständig zu behandeln. Es geht mehr darum, ein grundlegendes Verständnis von Programmierung zu bekommen und dies in C auf einfache Probleme anzuwenden. Weitere Möglichkeiten von C und sogar andere Programmiersprachen kann man sich dann relativ leicht im Selbststudium aneignen. Dazu gibt es eine Vielzahl von Büchern für jeden möglichen Geschmack und Anwendungsbereich. Beispielsweise

- P. Deitel und H. Deitel, "C: How to program", Prentice Hall, 7th edition
- S. P. Harbison, "C: A Reference Manual", Pearson, 5th edition
- J. Goll, U. Bröckl, M. Dausmann,"C als erste Programmiersprache", Teubner, 4., überarbeitete und erweiterte Auflage
- W.H Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, "Numerical Recipes in C book set: Numerical recipes in C. The art of scientific computing", Cambridge, 2th edition
- H. Gould und J. Tobochnik, "An Introduction to Computer Simulation Methods: Applications to Physical Systems (Englisch)", Addison Wesley Longman, 3th edition

Bei konkreten Fragen ist auch das Internet oft sehr hilfreich. Beispielsweise sei hier auf `stackoverflow.com` hingewiesen, wo fast jede Frage schon einmal gestellt und auch beantwortet wurde.

Dieses Dokument wurde mit LaTeX erstellt und kann unter

`https://github.com/urbach/c-kurs`

frei heruntergeladen werden.

1.2 Ein erstes C Programm

Ein wichtiger Schritt hin zu einem Programm ist die Formulierung des Verfahrens für die Lösung eines Problems als Algorithmus.

Definition: Algorithmus

Ein **Algorithmus** ist eine präzise Vorschrift, um aus vorgegebenen Eingaben in endlich vielen Schritten eine bestimmte Ausgabe zu ermitteln.

Hat man dies geschafft, so muss der Algorithmus in die jeweilige Programmiersprache, hier also C, umgesetzt werden. Betrachten wir als Beispiel folgendes Problem: Nehmen wir an wir betrachten Daten

$$x_0, x_1, \ldots, x_{n-1}$$

von einem bestimmten Datentyp, für den wir eine Operation größer-gleich (oder kleiner-gleich) und kleiner (oder größer) definiert haben. Man beachte, dass wir ab jetzt bei der Indizierung der C Konvention folgen und von 0 bis $n-1$ indizieren. Die Aufgabe ist eine Permutation $\sigma(i)$, $i = 0, ..., n-1$ zu finden, so dass wir folgende *sortierte* Liste

$$x_{\sigma(0)} \leq x_{\sigma(1)} \leq \ldots \leq x_{\sigma(n-1)}$$

von Daten erhalten.

Ein einfacher Algorithmus, um dieses Problem zu lösen heißt *Einfügesortieren*. Er kann in folgen Schritten formuliert werden:

1. Beginne mit zwei Listen, einer sortierten Liste S und einer unsortierten U. Am Anfang besteht U aus der zu sortierenden Liste und S ist leer.

2. Verschiebe das jeweils erste Element aus U nach S. Füge das Element dabei so in S ein, dass S immer sortiert ist.

3. Wiederhole 2. so oft, bis U leer ist.

Wahrscheinlich ist jedem klar, dass diese Vorschrift in n Schritten das gewünschte Ergebnis liefern wird. Leider wird der Computer bzw. der `C`-Compiler den Algorithmus so nicht verstehen. Deswegen werden wir den Algorithmus jetzt in `C` übersetzen. Dafür führen wir zunächst Datentypen in `C` ein, damit wir die Liste $\{x_i\}$ auf dem Rechner darstellen können. Anschließend führen wir sogenannte Kontrollstrukturen ein, um obigen Algorithmus abbilden zu können.

1.2.1 Daten- und Speichertypen

Bevor wir `C`-Datentypen vorstellen, ist es hilfreich zu verstehen, wie Daten allgemeine auf einem Rechner gespeichert werden. Speicher, egal ob Hauptspeicher oder Festplatte benutzt als kleinste Speicherzelle ein Element das entweder den Zustand 0 oder 1 annehmen kann. Ein solches Element nennt man Bit. Das heißt mit einem Bit kann man genau zwei Zustände darstellen. Fasst man 8 Bits zu einem Byte zusammen, so kann man $2^8 = 256$ Zustände darstellen. Größere Speicherbereiche nennt man

- Byte: 1 Byte, 2^8 Zustände
- Word: 2 Byte, 2^{16} Zustände
- Dword: 4 Byte, 2^{32} Zustände („*double-word*“)
- Qword: 8 Byte, 2^{64} Zustände („*quad-word*“)

Beispielsweise kann Text in einer Datei im ASCII Format gespeichert werden. Das bedeutet, dass jedes Zeichen genau ein Byte in Anspruch nimmt. Damit kann aber im ASCII Format lediglich ein Zeichenumfang von 256 Zeichen dargestellt werden.

Die verschiedenen Speichertypen haben zwei wichtigen Eigenschaften:

- Die totale Größe
- Die Zugriffszeit

Typischerweise ist die Zugriffszeit invers proportional zur totalen Größe des Speichermediums. Zum Beispiel ist Hauptspeicher ungefähr 10.000 mal schneller zu erreichen, als eine Festplatte, aber 50 mal langsamer als die Register einer CPU. Die Register bestehen aber nur aus wenigen Kilobytes, der Hauptspeicher aus einigen Gigabyte, und die Festplatte heutzutage aus einigen Terabyte. Für uns ist hier aber lediglich der Unterschied Festplattenspeicher und Hauptspeicher von Bedeutung, da die Register im Normalfall vom Compiler angesteuert werden.

1.2.2 Maschinenzahlen

Auf einem Rechner ist lediglich eine Teilmenge $\mathcal{M}$ der reellen Zahlen darstellbar. Nach IEEE Standard wird eine Fließkommazahl wie folgt dargestellt:

$$x = \text{sign}(x) \cdot a \cdot E^{e-k} \tag{1.1}$$

wobei $E \in \mathbb{N}, E > 1$ die Basis ist (meist $E = 2$), $k \in \mathbb{N}$ die Genauigkeit und e im Exponentenbereich $e_{\min} < e < e_{\max}$ liegt mit $e_{\min}, e_{\max} \in \mathbb{Z}$. Die Mantisse $a \in \mathbb{N}_0$ ist definiert als

$$a = a_1 E^{k-1} + a_2 E^{k-2} + ... + a_k E^0 \,, \tag{1.2}$$

wobei k die Mantissenlänge darstellt und a_i die Ziffern im entsprechenden Zahlensystem sind. Auf modernen Rechnern ist üblicherweise $a_i \in \{0, 1\}$ im Dualsystem mit Basis $E = 2$.

Bei der Abbildung der reellen Zahlen auf die Menge der Maschinenzahlen muss fast immer eine Rundungsoperation vorgenommen werden. Dabei geht Information verloren, eine Rückabbildung ist nicht eindeutig möglich.

Zahlendarstellung

1. Die Abbildung der Zahl 0,1 im Dezimalsystem auf das Dualsystem $0{,}1_{10} = 0{,}0001\,1001\,1001\,1001\ldots_2$ ist ein unendlicher periodischer Dualbruch und damit mit endlicher Stellenzahl nicht exakt darstellbar.

2. beim Addieren zweier k-stelliger Zahlen entsteht im Allgemeinen eine $k+1$ stellige Zahl. Überschreitet bei einem solchen Schritt $k+1$ die maximal verfügbare Stellenzahl, so kommt es zu einem sogenannten Überlauf (Englisch: *overflow*), der zum Fehlschlagen numerischer Verfahren führen kann. Dies ist jedoch hauptsächlich bei ganzzahligen Datentypen von Relevanz.

3. die bei der Fließkommadarstellung inhärente Rundung führt dazu, dass ein gegebener Algorithmus Ergebnisse nur bis zu einer bestimmten Genauigkeit berechnen kann. Man spricht hierbei von *Rundungsfehlern*. Es ist Vorsicht geboten: bei manchen numerischen Verfahren kann dies dazu führen, dass unzureichend genaue oder gar völlig falsche Ergebnisse berechnet werden.

Als Maschinengenauigkeit bezeichnet man die größte reelle Zahl δ_M für die der Rechner

$$1 + \delta_M = 1 \tag{1.3}$$

liefert. Für die Abbildung der reellen Zahlen auf Maschinenzahlen gilt dann notwendigerweise

$$-\delta_M \leq \delta x \leq \delta_M \,, \tag{1.4}$$

wobei δx den Zahlenbereich angibt, der zu 0 ausgewertet wird.

1.2.3 C Quelltext

Daten werden im C Quelltext durch sogenannte Variablen repräsentiert. Auf Variablen können wir Operation ausführen, oder sie an Funktionen übergeben. Zunächst stellen wir jetzt vor, wie man Variablen deklariert, ihnen einen Wert zuweist und wie man sie beispielsweise auf dem Monitor ausgeben kann. C kennt Datentypen für ganze Zahlen und für reelle Zahlen. Beispiele sind `int` für ganze und `float` für reelle Zahlen. Diese sind in verschiedenen Größen verfügbar, also mit verschiedener Anzahl an bits. Verschiedene bit-Anzahl erlaubt die Darstellung verschiedener Zahlenbereiche.

Ein erstes C-Programm

Ein C-Programm ist ein Textstück, das entsprechend den Sprachregeln von C formuliert sein muss. Es besteht im Allgemeinen aus Deklarationen, Anweisungen, Kontrollstrukturen und Kommentaren. Das vielleicht einfachste C-Programm hat folgende Form:

```
1 int main()
2 {
3   // dies ist ein Kommentar
4   /*
5     Dies ist ein mehrzeiliger Kommentar
6   */
7   return 0; // Rueckgabe 0
8 }
```

Listing 1.1: Ein erstes C-Programm

An Hand dieses einfachen Programms können wir schon einiges über die C Sprachregeln lernen. Jedes Programm in C muss die Funktion `main` genau einmal definieren. Die `main` Funktion ist auch der Startpunkt für jedes C Programm. Die von uns gerade definiert Funktion `main` hat eine ganze Zahl (`int`) als Rückgabewert. Da der geklammerte Bereich direkt nach `main` leer ist, bekommt die Funktion keine Parameter übergeben. Schlüsselwörter, hier `int` und `main` werden durch ein oder mehrere Leerzeichen voneinander getrennt.

Mit den geschweiften Klammern wird ein Abschnitt oder Block definiert, in diesem Fall der Block der Funktion. Die beiden Schrägstriche `//` lassen den Compiler alles danach folgende bis zum Zeilenende als Kommentar interpretieren. Wir empfehlen, nicht die mehrzeiligen Kommentare `/* */` zu verwenden. Der Grund dafür ist, dass wenn man einen mehrzeiligen Bereich auskommentiert, der schon einen mehrzeiligen Kommentar enthält, dann endet der Kommentarbereich beim ersten `*/`, und man bekommt eine Fehlermeldung.

Die Funktion `return` beendet die Abarbeitung der Funktion `main` und gibt einen Wert an die aufrufende Funktion zurück. Jede Funktion sollte (muss aber nicht) immer explizit `return` aufrufen, auch wenn es keinen Rückgabewert gibt. Jede Deklaration oder Anweisung, in diesem Fall der Aufruf von `return`, muss mit einem Semikolon `;` abgeschlossen werden. Deklarationen oder Anweisungen sind nicht an

Zeilen gebunden und können über mehrere Zeilen verteilt werden. Leere Zeilen werden vom Compiler nicht beachtet. Groß- und Kleinschreibung sind wichtig, `Foo` und `foo` sind also unterschiedlich. Da auch Leerzeichen am Zeilenanfang beliebig sind, werden Zeilen normalerweise eingerückt. Das erhöht die Lesbarkeit des Quelltextes. Gute Editoren stellen Einrückungs Schemata zur Verfügung.

Soweit die Erklärung für das erste Programm. Wie übersetzt man nun dieses Programm? Mit Übersetzen bezeichnet man den Schritt, in dem der `C` Quelltext in sogenannten Maschinentext übersetzt wird. Maschinentext kann dann direkt von einem Rechner interpretiert werden. Wir zeigen das Übersetzen beispielhaft für Linux und den GNU `C` Compiler `gcc`. Angenommen, obiger Quelltext ist in einer Datei `main.c` im momentanen Arbeitsverzeichnis gespeichert. Dann kann man die Datei mit folgendem Aufruf auf der Kommandozeile übersetzen:

```
>$  gcc -std=c99 -Wall -pedantic main.c -o main.exe
```

Das in Maschinentext übersetzte Programm kann man dann mit

```
>$  ./main.exe
```

von der Kommandozeile aus ausgeführt werden. In obigem Aufruf des GNU Compilers sind nicht alle Kommandozeilenparameter strikt notwendig. Die Parameter `-Wall -pedantic` sorgen dafür, dass der Compiler möglichst viele Warnungen ausgibt und alle möglichen Probleme im Quelltext auch mitteilt. Wir denken, dass es gerade für Anfänger sehr wichtig ist, Quelltext so sauber wie möglich zu schreiben. Deshalb möchten wir dringend dazu auffordern, diese Compiler Parameter zu benutzen. Der Parameter `-std=c99` sorgt dafür, dass der `gcc` den C99 Standard verwendet. Aller Quelltext in diesem Dokument ist für diesen Standard geschrieben und übersetzt nicht notwendigerweise für ältere Standards.

Eine Alternative zum `gcc` ist der `clang` Compiler. Er hat einen ganz ähnlichen Aufruf

```
>$  clang -std=c99 -Wall -pedantic main.c -o main.exe
```

`clang` und `gcc` sind beide freie Software. Ihre Installation unter Linux ist sehr leicht, aber auch unter Windows kann man beide – mit einigem Aufwand – ebenfalls installieren.

Bisher tut unser erstes Programm noch nichts, außer Null zurückgeben. Wir können es um eine Ausgabe auf den Bildschirm erweitern:

```c
#include <stdio.h>

int main()
{
  // Ausgabe auf dem Bildschirm
  printf("Hallo Welt\n");
  return 0;
}
```

Listing 1.2: Programm Hallo Welt

Es sind zwei Dinge dazugekommen: Erstens haben wir eine sogenannte Header-Datei, in diesem Fall `stdio.h` eingebunden. Das ist nötig, damit der `C`-Compiler die Funktion `printf` kennt. Denn `printf` ist nicht Teil der `C` Sprache, es ist in einer sogenannten Bibliothek definiert. Mit der Einbindung der Header-Datei wird die Funktion dem Compiler bekannt gemacht. Dies werden wir später noch genauer erörtern.

Zweitens ist der Aufruf von `printf` (*print formatted*) hinzugekommen. Die Funktion `printf` gibt in diesem Fall die Zeichenkette (den *string*) „Hallo Welt“ auf dem Standard Ausgabegerät aus, was normalerweise die Kommandozeile selbst ist, wenn das Programm von der Kommandozeile aufgerufen wird. Auch die Funktion `printf` werden wir später noch im Detail diskutieren. In diesem Beispiel kopiert `printf` die Zeichenkette unverändert zur standard Ausgabe, also wahrscheinlich auf die Konsole. `\n` erzeugt einen Zeilenumbruch. Wieder wird der Aufruf von `printf` mit einem Semikolon `;` abgeschlossen. Man beachte, dass `return` keine Klammern um den Rückgabewert benötigt. `return` stellt keinen Funktionsaufruf dar.

Der Rückgabewert von `main` kann übrigens an der Linux (Unix) Kommandozeile wie folgt abgefragt werden

```
>$  ./main.exe
>$  echo $?
0
```

Man kann der Funktion `main` auch Parameter übergeben. Wie, werden wir später sehen.

1.3 Variablen und Operatoren

Wie schon angedeutet werden Daten in `C` in Variablen bzw. Objekten gespeichert. Variablen haben immer einen Typ, einen Namen und einen Wert. Namen und Typen von Variablen müssen dem Compiler bekannt gemacht werden.

1.3.1 Deklaration und Initialisierung

Dies geschieht bei der Deklaration von Variablen, die allgemein wie folgt aussieht

```
Datatype name;
```

Die Variable muss dann noch initialisiert werden

```
name = value;
```

Es geht auch beides zusammen in einem Schritt

```
Datatype name = value;
```

Dies wird am folgenden Beispiel an Hand des ganzzahligen Datentyps `int` veranschaulicht

```
#include <stido.h>

int main()
{
  int n;
  n = 4;
  printf("Wir werden n=%d zahlen sortieren\n", n);
  return(0);
}
```

Listing 1.3: Erste Variablendeklaration und -zuweisung

Im Einzelnen geschieht das folgende:

- `int n;`
  Diese Anweisung stellt eine Deklaration dar. Sie teilt dem Compiler mit, ab jetzt den entsprechenden Speicherplatz für eine ganze Zahl vom Typ `int` bereitzustellen, also 4 byte. Außerdem kennt der Compiler ab jetzt den Namen `n` innerhalb des Blockes, der durch `{}` begrenzt wird. Streng genommen findet gleichzeitig auch eine Definition statt, weil der entsprechende Speicherplatz reserviert wird. Man unterscheidet Deklaration und Definition, weil es auch Deklarationen ohne Bereitstellung von Speicher gibt. In einem solchen Fall wird nur das Objekt bekannt gemacht.

- `n = 4;`
  Diese Anweisung stellt eine Zuweisung dar. Der Variable `n` wird der Wert 4 zugewiesen. An der entsprechenden Stelle im Speicher wird dieser Wert abgelegt. Bei einer Definition führt `C` keine Initialisierung durch. Nach der reinen Definition (ohne Zuweisung) ist der Wert der Variablen `n` also rein zufällig.

- `printf(...)`
  Diese Anweisung gibt den Text auf der Konsole aus und ersetzt den Platzhalter `%d` mit dem Wert der Variablen `n`, mehr Details dazu in Abschnitten 1.3.6 und 2.2.2.

Es ist wichtig, dass jeder Benutzung einer Variablen, beispielsweise in einer Zuweisung, die Deklaration der Variablen vorangehen muss.

Tabelle 1.1: Elementare Datentypen

Name	Varianten	Größe in Byte	Minimaler Wert	Maximaler Wert
int	int	4	−2 147 483 648	2 147 483 647
	short	2	−32 768	32 767
	unsigned short	2	0	65 535
	unsigned	4	0	+4 294 967 295
	long	4	−2 147 483 648	2 147 483 647
char	signed	1	−128	127
	unsigned	1	0	255
float		4		
double		8		
long double		8		

1.3.2 Sichtbarkeitsbereich

Wie oben ersichtlich, haben Variablen einen Sichtbarkeitsbereicht und damit auch eine Lebensdauer. Innerhalb eines Blockes kann man nicht zwei Variablen mit gleichem Namen deklarieren, dies führt zu einer Fehlermeldung des Compilers. Deklariert man in einem Unterblock eine Variable mit dem Namen einer Variablen aus dem darüberliegenden Block, so ist die Variable aus dem darüberliegenden Block verdeckt. In unserem Beispiel von oben heißt das, dass die Variable `n` nur in der Funktion `main` sichtbar ist.Variablen, die außerhalb aller Funktionen deklariert werden, bezeichnet man als *global*. Sie sind in allen Funktionen, die nach der globalen Deklaration definiert werden, sichtbar und zugreifbar.

1.3.3 Datentypen und Wertebereiche

Variablen haben immer einen Datentyp und einen Wert. Der Datentyp entscheidet, welche Werte eine Variable annehmen kann und wie viel Arbeitsspeicher dafür reserviert wird. In der Tabelle 1.1 sind die elementaren `C`-Datentypen mit ihren Wertebereichen beispielhaft aufgelistet, wobei man bedenken muss, dass sowohl die Wertebereiche, als auch die Größe in Byte architekturabhängig sein können. Ferner gibt es für ganzzahlige Datentypen auch `long long` Varianten, welche meist eine Größe von 8 Byte aufweisen und entsprechend große Wertebereiche haben.

`C` Compiler führen im Prinzip eine strenge Typenkontrolle durch. Das ist eine sehr nützliche Eigenschaft von Compilern, wenn es auch manchmal etwas mühsam ist. Man kann dies durch einen expliziten *cast* umgehen. Beispielsweise wandelt `int n=4; double x = (double)n;` die ganze Zahl `n` in eine Fließkommazahl mit `double` Genauigkeit um. Dafür sollte man aber sehr genau wissen, was man tut. Leider ist die Typenkontrolle vom Compiler abhängig und meist wird bei einer Zuweisung ein impliziter *cast* durchgeführt, wenn nötig und möglich. Beispielsweise wird folgender Code ohne Beanstandung übersetzt

```c
#include <stido.h>

int main()
{
  // so etwas sollte man nicht schreiben!
  int n = 4.5; // implicit cast
  return 0;
}
```

Listing 1.4: Ungenaue Variablenzuweisung, implicit cast

obwohl hier implizit die reelle Zahl 4.5 durch Abschneiden in eine ganze Zahl umgewandelt wird. `n` hat den Wert 4.

Im allgemeinen sollte man versuchen immer den Datentyp zu verwenden, der der Nutzung der entsprechenden Variablen am ehesten entspricht. Wenn man also beispielsweise weiß, dass eine ganze Zahl nicht negativ werden kann, sollte man `unsigend int` verwenden. Dies macht es möglich, dass der Compiler Fehler in der Nutzung von Datentypen finden kann.

1.3.4 `C` Schlüsselwörter und Regeln für Namen

Es gibt einige Regeln für die Namen von Objekten in `C`. Schlüsselworte der Sprache `C`, wie z.B. `main` dürfen nicht verwendet werden. Auch dürfen die Namen nicht mit einer Zahl beginnen, auch wenn Zahlen generell erlaubt sind. Operatornamen, wie z.B. `+`, `-` oder `=`, dürfen ebenfalls nicht verwendet werden. Es ist ratsam, Variablen mit sinnvollen Namen zu versehen. Das macht den Quelltext lesbarer und erhöht die Verständlichkeit des Programms. Für den Algorithmus Einfügesortieren könnte man beispielsweise die beiden Liste mit `sortiert` und `unsortiert` benennen. Im folgenden Quelltext sind einige Beispiele für richtige und falsche Variablendeklarationen zu finden:

```c
int main()
{
  int m1 = 4, n1 = 5, l1 = 6;                // Richtig
  int m2 = 4, char n2 = 'a', float m2 = 4.; // Falsch
  char m3 = 'a';                             // Richtig
  double n3 = 18.9;                          // Richtig
  float 4m = 1.;                             // Falsch
  return (0);
}
```

Wie man sieht, kann man mehrere Variable in einer Anweisung deklarieren, definieren und initialisieren, wenn sie den gleichen Typ haben. Die Variablen werden dabei durch ein Komma getrennt. Wie schon oben erwähnt, können mehrere Anweisung in der gleichen Zeile stehen, solange sie mit dem Semikolon abgeschlossen werden.

Bei folgenden Wörtern handelt es sich um C99 Schlüsselworte:
auto, double, int, struct, break, else, long, switch, case, enum, register, typedef, char, extern, inline, return, union, const, float, short, unsigned, continue, for, restrict, signed, void, default, got, sizeof, volatile, do, if,

static, while, _Bool, _Complex, _Imaginary Der überwiegende Teil dieser Schlüsselwörter wird in diesem Dokument vorgestellt.

1.3.5 Konstanten

`C` erlaubt auch, Variablen als konstant zu deklarieren. Dies bedeutet, dass sich der einmal zugewiesene Wert einer solchen als `const` deklarierten Variablen nicht ändern darf. Im Quelltext sieht das wie folgt aus

```
1 const int n = 5;
```

Notwedigerweise müssen als `const` deklarierte Variablen immer initialisiert werden, denn eine spätere Zuweisung ist nicht erlaubt. Die Benutzung von `const` kann große Vorteile haben. Erstens, wenn wir wissen, dass sich eine Variable nicht mehr ändern wird und wir sie als `const` deklariert haben, dann kann der Compiler das überprüfen und eine Fehlermeldung ausgeben, wenn wir versehentlich den Wert der Variablen doch ändern. Zweitens ist der Wert von `const` Variablen zur Zeit der Übersetzung bekannt und erlaubt dem Compiler einige Optimierungen. Im Allgemeinen sollte man `const` immer verwenden, wenn die Variable sich nicht mehr ändern soll.

1.3.6 Operatoren

Variablen können mit Hilfe von Operationen manipuliert werden. Natürlich hängt es vom Variablentyp ab, welche Operationen dafür definiert sind. Man unterscheidet drei verschiedene Typen von Operationen:

- Infix:
  Der Operator steht zwischen den Variablen. Zum Beispiel: `a+b`. Dieser Ausdruck nimmt die jeweiligen Werte von `a` und `b`, summiert sie und gibt das Ergebnis zurück.

- Präfix:
  Der Operator steht vor der Variablen. Zum Beispiel: `++a`. Dieser Ausdruck erhöht den Wert von `a` um 1 und gibt danach den neuen Wert von `a` zurück.

- Postfix:
  Der Operator steht nach der Variablen. Zum Beispiel: `a--`. Dieser Ausdruck reduziert den Wert von `a` um 1, aber gibt den originalen Wert von `a` zurück.

Wieder sieht man es am einfachsten an einem Beispiel:

```c
#include <stdio.h>

int main()
{
  int a = 2;
  printf("%d\n", a++); // gibt 2 aus
  printf("%d\n", a);   // gibt 3 aus
  printf("%d\n", ++a); // gibt 4 aus
  printf("%d\n", a);   // gibt 4 aus
  return 0;
}
```

In diesem Beispiel wird zunächst eine Variable `a` mit dem Wert 2 initialisiert. Dann nutzen wir die Funktion `printf`, um den Wert der Variablen bzw. von Ausdrücken auszugeben. Das erste Argument von `printf` ist immer ein String, also eine Zeichenkette. Diese Zeichenkette wird unverändert in den standard output kopiert. Die Ausnahme sind Zeichenfolgen, die mit einem Prozentzeichen % beginnen. Die auf das % folgenden Zeichen werden von `printf` in bestimmter Weise interpretiert. `%d` beispielsweise steht für eine ganze Zahl vom Typ `int`. Bei genau einem % in der Zeichenkette erwartet `printf` dann genau eine Variable als zweiten Parameter nach der Zeichenkette vom entsprechenden Typ, hier also vom Typ `int`. Man kann sich leicht überlegen, dass obiges Programm die Zahlenfolge $2, 3, 4, 4$ ausgibt. Die Bedeutung der mit % markierten Platzhalter wird in Teil 2.2 noch ausführlich behandelt werden.

In Bezug auf die Anzahl ihrer Argumente gibt es drei verschiedene Typen von Operatoren

- binäre Operatoren: Operatoren mit zwei Argumenten, wie z.B. `+`.
- unäre Operatoren: Die Operatoren haben nur ein Argument, wie z.B. `++`.
- trinäre Operatoren: Operatoren mit drei Argumenten. In `C` gibt es davon nur einen, nämlich `?:`.

Neben arithmetischen Operatoren gibt es auch noch solche, die bit–weise wirken. Außerdem gibt es logische Operatoren. Bit–weise Operatoren sind binäre Operatoren und wirken auf jedes bit des Arguments. In den Tabellen 1.2, 1.3 und 1.4 sind die wichtigsten arithmetischen Operatoren und logischen Operatoren zusammen gefasst.

Es gibt einige Dinge, die man sich bei der Benutzung von Operatoren bewusst machen sollte. An dieser Stelle weisen wir auf eine davon explizit hin:

- Die Division ist sowohl für ganze, als auch für reelle Zahlen definiert. Eine ganzzahlige Division von 7 durch 2 ergibt 3. Dagegen liefert eine Division von reellen Zahlen 7,0 und 2,0 das Ergebnis 3,5. Dementsprechend liefert

```c
float a = 7 / 3;
```

2.0 als Ergebnis. Man kann C mitteilen, dass man eine reellwertige Division durchführen möchte, indem man den Dezimalpunkt mit angibt

```
float a = 7. / 3;
```

Es ist empfehlenswert, dies immer explizit zu tun, damit es nicht durch spätere Änderungen zu schwer aufzufindenden Fehlern eines Programms kommt.

1.3.7 Logische Ausdrücke

Vergleichsoperatoren (siehe Tabelle 1.3) und logische Operatoren (siehe Tabelle 1.4) werden zum Bilden sogenannter logischer Ausdrücke verwendet. Logische Ausdrücke werden entweder zu wahr (*true*) oder falsch (*false*) ausgewertet. Intern repräsentiert C jeden logischen Ausdruck als ganze Zahl. Dabei entspricht $\neq 0$ wahr (*true*) und 0 falsch (*false*). Man kann anstelle des logischen Ausdrucks also auch einen ganzzahligen Ausdruck verwenden. Ein sehr einfacher logischer Ausdruck ist also beispielsweise

```
1; // -> wahr
```

Etwas komplexer geht es mit Vergleichsoperatoren

```
double x = 5.;
int n = 3;
(x > 1.);   // wahr
(x < 0.);   // falsch
(n == 3);   // wahr
(x == 4.9); // falsch, aber mit Fliesskommanzahlen nicht
 empfehlenswert
```

Das Prüfen auf Gleichheit ist für reelle Maschinenzahlen nicht wohl definiert. Der Grund dafür ist, wie oben diskutiert, die Maschinengenauigkeit. Zwei reelle Zahlen werden vom Rechner als gleich ausgewertet, falls sie sich ihr Betrag um weniger als $|\delta_M|$ unterscheidet. Oder anders formuliert, $|\delta_M|$ ist als die größte reelle Zahl definiert, für die

```
(1 == 1 + delta_M);
```

zu wahr ausgewertet wird. Deshalb sollte man wenn irgend möglich zwei reelle Zahlen nicht auf Gleichheit prüfen. Stattdessen sollte man wenn möglich größer oder kleiner verwenden.

Logische Ausdrücke können natürlich auch verkettet werden. Also beispielsweise

```
double x = 5.;
int n = 3;
((x > 3) && !n); // falsch
```

Hierbei werden die einzelnen logischen Ausdrücke von links nach rechts ausgewertet. Hätten wir also geschrieben

```
double x = 5.;
int n = 3;
(!n && (x > 3)); // falsch
```

Operator	Ausdruck	Auswertung
Zuweisung	`a = b`	Wert von `b`
Addition	`a + b`	Summe von `a` und `b`
Subtraktion	`a - b`	Differenz von `a` und `b`
Multiplikation	`a * b`	Produkt von `a` und `b`
Division	`a / b`	Quotient von `a` und `b`
Zuweisung und Addition	`a += b`	Wert von `a+b`
Zuweisung und Subtraktion	`a -= b`	Wert von `a-b`
Zuweisung und Multiplikation	`a *= b`	Wert von `a*b`
Zuweisung und Division	`a /= b`	Wert von `a/b`
Modulo	`a % b`	`a` modulo `b`
Inkrement	`++a, a++`	Präfix: `a`+1, Postfix: `a`
Dekrement	`--a, a--`	Präfix: `a`-1, Postfix: `a`
Positiver Vorzeichenoperator	`+a`	Wert von `a`
Negativer Vorzeichenoperator	`-a`	Wert von `-a`

Tabelle 1.2: Arithmetische Operatoren

Operator	Ausdruck
Prüft auf Gleichheit	`a == b`
Prüft auf Ungleichheit	`a != b`
Prüft, ob `a` echt größer als `b` ist	`a > b`
Prüft, ob `a` echt kleiner als `b` ist	`a < b`
Prüft, ob `a` größer gleich `b` ist	`a >= b`
Prüft, ob `a` kleiner gleich `b` ist	`a <= b`

Tabelle 1.3: Vergleichsoperatoren

wird `x > 3` nicht mehr ausgewertet, da `!n` bereits zu falsch ausgewertet wurde. Dies kann dann von Bedeutung sein, wenn das Auswerten eine teure Operation ist. Durch geschickte Ordnung der logischen Ausdrücke kann man dann unter Umständen Zeit sparen.

Regeln zum Bilden von Ausdrücken

Der `C` Compiler geht bei der Auswertung von Operatoren in einer bestimmten Reihenfolge vor. Es gelten im Allgemeinen die Vorrangregeln der Algebra beim Auswerten eines Ausdrucks, inklusive der Klammerregeln. So werden z.B. `*`, `/` und `%` vor `+` und

Operator	Ausdruck	Wert
Logisches UND	`a && b`	`a` und `b`
Logisches ODER	`a || b`	`a` oder `b`
Negation	`!a`	nicht `a`

Tabelle 1.4: Logischen Operatoren

Rang	Operatoren
0	`.`, `->`, `[]`, `()`
1	`&` (Adressoperator), `*` (Dereferenzierung)
2	`*`, `/` `%`
3	`<`, `>`, `<=`, `>=`
4	`==`, `!=`
5	`&&`
6	`||`
7	alle Zuweisungen `=`, `+=`, `-=`, ...

Tabelle 1.5: Priorität von Operatoren

- ausgewertet. Ein unvollständiger Auszug aus der Prioritätenliste ist in Tabelle 1.5 zusammengefasst. Kleinerer Rang bedeutet dabei höhere Priorität.

1.4 Kontrollstrukturen: Verzweigungen und Schleifen

Bisher haben wir einfache Anweisungen kennen gelernt, die vom Rechner nacheinander ausgeführt werden. Wir werden nun Kontrollstrukturen einführen, die es erlauben, den Fluss eines Programmes zu beeinflussen. Beispielsweise kann man, in Abhängigkeit von Werten von Variablen entweder einen, oder einen anderen Block von Anweisungen ausführen. Solche Kontrollstrukturen nennt man Verzweigungen. Das kann beispielsweise bedeuten, dass der Programmteil `A` ausgeführt wird, falls eine Variable x größer als Null ist, und sonst der Programmteil `B`. Die Entscheidung wird auf Grundlage logischer Ausdrücke gefällt, siehe den vorangegangenen Abschnitt.

1.4.1 Einfache Verzweigung: `if-else`

Im Allgemeinen hat das *if-else* Konstrukt folgendes Aussehen:

```
1  if (logischer Ausdruck)
2    {
3      // falls wahr
4      Anweisung1;
5      Anweisung2;
6      ...;
7    }
8  else // optional
9    {
10     // sonst
11     Anweisung3;
12     Anweisung4;
13     ...;
14   }
```

Listing 1.5: if-else Statement

In Abhängigkeit vom logischen Ausdruck wird entweder der erste (nach *if*) oder

der zweite (nach *else*) Codeblock ausgeführt. Diese beiden Codeblöcke sind durch geschweifte Klammern definiert. Die Klammern können auch weggelassen werden, wenn ein Block nur aus genau einer Anweisung besteht. Wir empfehlen trotzdem auch in diesem Fall Klammern zu setzen. Dies verhindert spätere Fehler, und macht den Quelltext lesbarer. Der logische Ausdruck kann prinzipiell alle Operatoren enthalten, bzw. auch verkettete Ausdrücke von Operatoren, die wir in Tabellen 1.3 und 1.4 zusammengestellt haben.

`Anweisung1` kann explizit auch wieder ein *if-else* Ausdruck sein. Man kann *if-else* also beliebig schachteln. Außerdem ist der *else* Block optional. Wird der *else* Block ausgelassen, so wird in dem Fall, in dem der logische Ausdruck zu *falsch* ausgewertet wird, der Block mit Anweisungen nicht ausgeführt.

Nehmen wir beispielsweise an, wir wollen den Absolutwert einer Variablen `a` ausgeben. Dafür kann man einen `if-else` Ausdruck verwenden. Der entsprechende Quelltext könnte so aussehen:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a = 4;
6    unsigned int absolutevalue = 0;
7    if (a > 0) // a ist positiv
8      {
9        absolutevalue = a;  // Betrag ist direkt gleich a
10     }
11   else       // a ist negativ
12     {
13       // falls a <= 0
14       absolutvalue = -a;  // Betrag ist gleich -a
15     }
16   printf("Der Absolutwert von a ist %u\n", absolutevalue);
17   return (0);
18 }
```

In Abhängigkeit davon, ob `a > 0` zu wahr ausgewertet wird oder nicht, wird in diesem Beispiel entweder der Codeblock nach dem Schlüsselwort `if` oder der nach dem Schlüsselwort `else` ausgeführt. Die Variable `absolutevalue` ist außerhalb beider Blöcke deklariert, und damit in beiden sichtbar. Die Zuweisung, die `absolutevalue` innerhalb der Blöcke bekommt, ist damit auch nach Ende der Blöcke weiterhin erhalten.

1.4.2 Mehrfache Verzweigung: `switch`

Ein dem *if-else* Konstrukt verwandtes Konstrukt ist das *switch* Konstrukt. Es erlaubt eine ganze Liste von *ganzzahligen* Alternativen abzuarbeiten. Im Allgemeinen sieht das also so aus:

```
switch (int)
  {
    case Wert1:
      Anweisung1;
      ....;
      break; // optional
    case Wert2:
      Anweisung2;
      ....;
      break; // optional
      ....
    default: // optional
      Anweisung3;
      break; // optional
  }
```

Listing 1.6: switch statement

Das Konstrukt wird mit dem Schlüsselwort `switch` eingeleitet. Jede der Möglichkeiten beginnt mit dem Schlüsselwort `case`, gefolgt vom möglichen Wert des Ausdruckes und einem Doppelpunkt. Falls die entsprechende Möglichkeit realisiert ist, werden alle noch folgenden Anweisungen ausgeführt, bis ein `break` Schlüsselwort kommt, oder der `switch` Block zu Ende ist. Der `default` Zweig wird dann ausgeführt, wenn keiner der anderen Fälle gepasst hat.

Im folgende Bespiel wird eine ganze Zahl von der Standardeingabe eingelesen, und zwar mit Hilfe der `scanf` Funktion, die eine `printf` sehr ähnliche Syntax hat. Wir diskutieren die Details zu `scanf` später. Dann wird, in Abhängigkeit vom eingegebenen Wert eine Ausgabe auf dem Bildschirm gemacht.

Beispiel: `switch` statement

```
#include <stdio.h>

int main()
{
  int n = 0;
  scanf("%d", &n);
  switch (n)
    {
      case 0:
        printf("Der Eingabewert ist 0\n");
        break;
      case 1:
        printf("Der Eingabewert ist 1\n");
      case 2:
        printf("Der Eingabewert ist 1 oder 2\n");
        break;
      case 3:
        printf("Der Eingabewert ist 3\n");
        break;
      default:
```

```
21          printf("Der Eingabewert ist groesser als 3 oder kleiner
      als 0\n");
22          break;
23       }
24    return (0);
25 }
```

D.h., wenn 0 eingegeben wird, wird Zeile 10 ausgeführt. Wenn 1 eingegeben wird, wird Zeile 13 und 15 ausgeführt. Und, wenn keines von $0, 1, 2, 3$ eingegeben wird, also keine der Möglichkeiten passt, so wird der `default` Zweig ausgeführt, und damit Zeile 21.

1.4.3 Schleifen: `for`

Eine wesentlich wichtigere Kontrollstruktur sind sogenannte *for* Schleifen. Allgemein sieht die *for* Schleife wie folgt aus:

```
1 for (Ausdruck1; Ausdruck2; Ausdruck3)
2   {
3     Anweisung1;
4     Anweisung2;
5     ...;
6   }
```

Listing 1.7: for Schleife

Im Einzelnen bedeutet das:

1. Zuerst wird `Ausdruck1` ausgewertet.
   In diesem ersten Ausdruck wird typischerweise die Schleifenvaraible deklariert und / oder initialisiert. (Es können auch mehrere Variablen deklariert und initialisiert werden.)

2. Dann wird `Ausdruck2` ausgewertet.
   Dieser zweite Ausdruck wird zu Beginn jeder Ausführung der Schleife ausgewertet und wird als logischer Ausdruck interpretiert. Er stellt die Abbruchbedingung dar.

   Wenn `Ausdruck2` zu wahr ausgewertet wird, werden die Anweisungen im Körper der Schleife ausgeführt.

3. Nach Ausführung des Schleifenkörpers wird `Ausdruck3` ausgewertet.
   Hier werden typischerweise Schleifenvariablen modifiziert.

Als Beispiel nehmen wir an, wir möchten die ganzen Zahlen von 0 bis $n-1$ aufsummieren und schreiben dazu folgenden Quelltext:

```c
#include <stdio.h>

int main()
{
  int n = 173;
  int summe = 0;
  for (int i = 0; i < n; i++) {
    summe += i;
  }
  printf("Die Summe der Zahlen von 0 bis %d ist %d\n", n - 1, summe);
  return (0);
}
```

Dabei ist `Ausdruck1` die Deklaration und Initialisierung von `i`

```c
  int i = 0;
```

Das Abbruchkrieterium ist der logische Ausdruck

```c
  i < n;
```

Der Körper der Schleife besteht in diesem Fall nur aus der Zeile

```c
summe += i;
```

die für `i=0,1,... n-1` nacheinander ausgeführt wird. Diese Zeile wird also $n-1$–Mal ausgeführt, mit jeweils anderen Werten für `summe` und `i`. Da `summe` außerhalb der Schleife deklariert und initialisiert wurde, ist der entsprechende Wert auch nach der Schleife verfügbar. `i` dagegen ist in diesem Beispiel nur innerhalb der Schleife verfügbar! `Ausdruck3` erhöht bei jedem Aufruf die Schleifenvariable `i` um eins.

```c
  i++
```

Interessanterweise dürfen auch alle drei Ausdrücke leer sein. Dann wird die Schleife im Prinzip unendlich oft ausgeführt. In einem solchen Fall kann man die Schleife mit Hilfe von `break` explizit abbrechen, wie man im folgenden modifizierten Beispiel sieht:

```c
#include <stdio.h>

int main()
{
  int n = 173;
  int summe = 0;
  int i = 0;
  for (;;) { // korrekt, aber schlechter Stil!
    if (i == n) break;
    summe += i;
    i++;
  }
  printf("Die Summe der Zahlen von 0 bis %d ist %d\n", n - 1, summe);
  return (0);
}
```

Der Quelltext führt immer noch die gleiche Aufgabe aus. Allerdings stellt das Ver-

wenden einer `for` Schleife in dieser Weise sehr schlechten Stil dar. Der Grund ist, dass man am Schleifenkopf nicht mehr erkennen kann, wann die Schleife abgebrochen wird und was die Schleifenvariablen sind. Der Quelltext wird also wesentlich unleserlicher und fehleranfälliger.

Die Benutzung von `break` kann und ist aber an vielen Stellen sinnvoll. Soll beispielsweise innerhalb einer Schleife eine externe Funktion aufgerufen werden, so sollte man immer überprüfen, ob diese Funktion auch ohne Fehler ausgeführt wurde. Falls die Funktion einen Fehler meldet, kann man mit Hilfe von `break` die Ausführung der Schleife unterbrechen:

```
1  for (int i = 0; i < n; i++) {
2    int errorcode = myFunction(i);
3    if( errorcode != 0 ) {
4      // take appropriate action
5      break;
6    }
7  }
```

Dies lässt sich für Fälle verallgemeinern, in denen unerwartete Bedingungen innerhalb der Ausführung einer Schleife auftreten.

1.4.4 Schleifen: `while` und `do-while`

C kennt zwei weitere Schleifenkonstrukte, nämlich die `while` und die `do-while` Schleife. `for` Schleifen haben in der Regel eine fest vorgegebene Anzahl von Durchläufen. Die `while` Schleifen Familie ist da flexibler: Die Anzahl an Durchläufen hängt nur von einem logischen Ausdruck ab. Während die `while` Schleife die Abbruchbedingung vor dem Ausführen des Schleifenkörpers überprüft und im Zweifel abbricht, wird bei der `do-while` Schleife der Körper immer mindestens einmal ausgeführt. Das macht den großen Unterschied zwischen den beiden aus. Allgemein hat die `while` Schleife folgende Form:

```
1 while (logischer Ausdruck)
2   {
3     Anweisungen;
4   }
```

Listing 1.8: while Schleife

Hier ist der `logische Ausdruck` die Abbruchbedingung. So lange der logische Ausdruck zu wahr ausgewertet wird, werden die Anweisungen im Schleifenkörper ausgeführt. Die Abbruckbedingung steht bei der `do-while` Schleife am Ende, wie man an der allgemeinen Form sieht:

```
do
  {
    Anweisungen;
  }
while (logischer Ausdruck);
```

Listing 1.9: do-while Schleife

Die Anweisungen werden also mindestens einmal ausgeführt bis der `logische Ausdruck` das erste Mal ausgewertet wird.

Ausführungen von `while` und `do-while` Schleifen können ebenfalls mit `break` abgebrochen werden. Natürlich können `while`, `do-while` und `for` Schleifen immer ineinander überführt werden. Allerdings ist dies manchmal mit Aufwand verbunden und es erscheint fast immer natürlich die eine oder andere Schleifenform zu verwenden.

1.5 Funktionen

Es gibt viele Quelltextabschnitte, die wiederholt benutzt werden. Es ist sinnvoll, diese Abschnitte im Quelltext nicht ständig zu wiederholen. Das erhöht einerseits die Lesbarkeit des Quelltextes und macht andererseits Code weniger fehleranfällig, da der Abschnitt nur einmal getestet werden muss. Dafür existiert das Konzept von Funktionen. Eine Funktion haben wir schon kennen gelernt, nämlich die Funktion `main`.

1.5.1 Funktionen Prototypen

In `C` sind Funktionen Variablen sehr ähnlich. Eine Funktion kann wie folgt deklariert werden

```
Rueckgabetyp Funktionsname(Parameterliste);
```

Man spricht von einem sogenannten *Funktionsprototypen*. Die Parameterliste besteht aus durch Kommata getrennten Variablendeklarationen

```
Typ1 parameter1, Typ2 parameter2, ...
```

Die Parameterliste kann auch leer sein. Der Rückgabetyp kann jeder `C` Typ und jeder selbst definierte Typ sein. Bei streng strukturierten Programmiersprachen wie `C` werden der Funktionsname, die Parameterliste und der Rückgabetyp zusammen als *Signatur* der Funktion bezeichnet.

1.5.2 Funktionen Definition

Die Definition einer Funktion muss dann natürlich einen Block von Anweisungen enthalten, also

```
Rueckgabetyp Funktionsname(Typ1 parameter1, Typ2 parameter2,
                           Typ3 parameter3, ...)
{
  Rueckgabetyp x;
  Anweisung1;
  Anweisung2;
  ...;
  return x;
}
```

Listing 1.10: Funktionen Prototyp

Die in der Parameterliste deklarierten Variablen sind dann innerhalb dieses Blocks definiert und unter ihrem Namen sichtbar. Außerdem sind alle *global* deklarierten Variablen im Funktionsblock sichtbar. Für den Funktionsnamen gelten die gleichen Regeln, wie für Variablennamen.

Als Beispiel betrachten wir eine Funktion, die die Fakultät einer ganzen Zahl berechnet:

```
unsigned long int Fakultaet(const unsigned int zahl)
{
  unsigned long int fak = 1;
  for (unsigned long int i = 2; i <= zahl; i++)
    {
      fak *= i;
    }
  return fak;
}
```

Der Rückgabetyp ist als `unsigned long int` gewählt, da die Fakultaet immer positiv ist, aber auch sehr groß werden kann.

Funktionen sollten wohldefinierte Unterprobleme lösen. Ihr Umfang hängt natürlich von der Komplexität dieser Unterprobleme ab. Man sollte trotzdem versuchen, dass Funktionen nicht zu viel Quelltext enthalten. Sonst kann man mit sehr großer Wahrscheinlichkeit den Quelltext noch weiter aufspalten.

1.5.3 Aufruf von Funktionen

Aufgerufen werden Funktionen dann im Prinzip wie folgt

```
Ergebnistyp Ergebnis = Funktionsname(Uebergabeliste);
```

Die `Uebergabeliste` enthält dabei Variablen aus dem aufrufenden Codeabschnitt oder konstante Ausdrücke. Dabei muss die Reihenfolge in der `Uebergabeliste` genau mit der in der `Parameterliste` in der Funktionsdeklaration übereinstimmen. Auch die Typen in `Uebergabeliste` müssen mit denen in der `Parameterliste` übereinstimmen. Der `C` Compiler überprüft dies strickt und bricht die Übersetzung ab, falls es Abweichungen gibt. Obige Funktion zur Berechnung der Fakultät kann wie folgt im Hauptprogramm aufgerufen werden:

```c
#include <stdio.h>

int main()
{
  int n = 20;
  unsigned long int result; // Variable fuer das Ergebnis
  result = Fakultaet(n);    // Funktionenaufruf
  // oder
  result = Fakultaet(3);    // Funktionenaufruf alternativ
  printf("%lud", result);   // Ausgabe des Ergebnisses
  return 0;
}
```

1.5.4 Typenüberprüfung und sinnvolle Eingabewerte

Wie schon oben erwähnt, führt der `C` Compiler für Aufrufe einer Funktion eine Typenüberpüfung durch, man muss jedoch bei elementaren Datentypen vorsichtig sein, da implizite casts zu unerwarteten Ergebnissen führen können. Der Aufruf der Funktion in der Form

```c
  unsigned long int result;
  double x = -2.0;
  result = Fakultaet(x);
```

wird leider vom Compiler[1] problemlos übersetzt. Die implizite Umwandlung macht aus `double` den einfachsten, ganzzahligen und vorzeichenbehafteten Datentyp, `int`. Dies führt wiederum dazu, dass nach einer weiteren Umwandlung zu `unsigned int`, die Zahl 4294967294 übergeben wird und das Programm in einer Endlosschleife endet.

Der Compiler kann allerdings nicht überprüfen, ob die Eingabe sinnvoll ist. Im Fall `Fakultaet` bekommt die Funktion nur einen Parameter vom Typ `unsigned int` übergeben. An dieser Stelle sollte man sich als Programmierer fragen, ob jede mögliche Eingabe zu einer sinnvollen Ausgabe führt. Das ist offensichtlich dann nicht mehr der Fall, wenn die funktionsinterne Variable `fak` aus dem Wertebereich für `long unsigned int` herausläuft. Man kann sich leicht überlegen, für welche Werte von `zahl` dies geschieht. Eine solche Überprüfung würde auch den oben aufgeführten Fehler zumindest abfangen und die Endlosscheife beenden.

Optimalerweise sollte in der Funktion überprüft werden, ob `zahl` im sinnvollen Wertebereich liegt. Wenn dies nicht der Fall ist, sollte die Bearbeitung abbrechen, zum Beispiel so

```c
  // pruefe, ob zahl im sinnvollen Wertebereich liegt
  if(zahl > N) {
    // falls nein, bricht die Ausfuehrung ab
    exit(EXIT_FAILURE);
  }
```

Die Funktion `exit` ist in der Headerdatei `stdlib.h` deklariert. `EXIT_FAILURE` (und `EXIT_SUCCESS`) ist ebenfalls in `stdlib.h` definiert.

[1] getestet unter `gcc 5.4.0` mit `-Wall -Wpedantic`

1.5.5 Rückgabetypen und `void` Funktionen

Funktionen haben in `C` immer einen Rückgabetyp. Hierbei sind alle `C` Typen und auch selbstdefinierte Typen möglich. Auf den ersten Blick scheint dies zu bedeuten, dass immer nur skalare Größen zurückgegeben werden können. Das stimmt aber nur für `C` Typen, und auch nur, wenn man Zeiger außen vor lässt. Letzteres wird im nächsten Kapitel diskutiert werden.

Es gibt auch den Fall, dass Funktionen keine Rückgabewert liefern. Für diesen Fall gibt es in `C` den `void` Datentyp. Eine `void` Funktion kann beispielsweise wie folgt aussehen:

```
1  void myFunction(...) {
2    // do something
3    return;
4  }
```

Diese Funktion kann dann wie folgt aufgerufen werden

```
1  // some code
2  myFunction(...);
3  // more code
```

also ohne eine Zuweisung des Funktionswertes an eine entsprechende Variable.

1.5.6 Funktionsaufrufe mit vielen Argumenten

Wenn einer Funktion viele Argumente übergeben werden, so kann man mithilfe von Kommentaren verdeutlichen, welcher Wert, welchem Parameter entspricht.

mehrzeiliger Funktionsaufruf

Der Aufruf einer Funktion mit vielen Argumenten kann recht schnell schwer verständlich werden.

```
1      void func(const int par1, const double par2,
2                const char par3, const int par4,
3                const double par5, const double par6,
4                const unsigned long int par7)
5      {
6        [...]
7      }
8      double x = 2.0;
9      double y = 4.5;
10     [...]
11     func(1, x, 'c', 3, y, 7.3, 8908123908);
12
```

Wir können diesen Aufruf übersichtlicher gestalten, indem wir den Aufruf ersten auf mehrere Zeilen aufspalten und zweitens, Kommentare einfügen, die den Parameter entweder beschreiben oder benennen.

```
1      double x = 2.0;
2      double y = 4.5;
```

```
    [...]
    func(/* par1 */           1,
         /* par2 */           x,
         /* par3 */         'c',
         /* par4 */           3,
         /* par5 */           y,
         /* par6 */         7.3,
         /* par7 */  8908123908);

```

Blickt man jetzt auf die Funktionssignatur und vergleicht mit dem Aufruf, sieht man viel schneller, ob man für die einzelnen Argumente die richtigen Datentypen sowie Werte übergibt.

1.6 Felder (arrays) und Zeichenketten

Bisher haben wir uns ausschließlich mit skalaren Datentypen beschäftigt, d.h. also einzelne Elemente eines bestimmten Typs. C kennt allerdings auch Tupel eines bestimmten Typs, sogenannte Felder oder im Englischen *arrays*. Allerdings muss die Länge des Tupels zum Zeitpunkt der Deklaration bekannt sein. Und diese Länge kann auch dann nicht mehr verändert werden.

1.6.1 Deklaration und Initialisierung eindimensionaler Felder

Die Deklaration eines Feldes ist ganz analog zu anderen Variablen

```
int a[5]; // array declaration
```

was in diesem Fall ein Feld der Länge 5 mit ganzen Zahlen erzeugt.

Für die Initialisierung gibt es zwei Möglichkeiten. Man kann erstens Deklaration und Initialisierung in einem Schritt vornehmen

```
int a1[5] = {1, 2, 3, 4, 5};
int a2[] = {1, 2, 3, 4, 5};
```

was jeweils ein Feld der Länge 5 erzeugt, aber diesmal initialisiert mit den Werten $1, 2, 3, 4, 5$. Bei der zweiten Zeile bestimmt der Compiler aus der Initialisierungsliste automatisch die Länge des Feldes. Die zweite Möglichkeit besteht darin, jedes Element einzeln zuzuweisen:

```
int a[5];
for(int i = 0; i < 5; i++) {
  a[i] = i+1;
}
```

was zum exakt gleichen Ergebnis führt. An diesem Beispiel sieht man bereits, dass in C die Indizierung von 0 bis $n-1$ läuft. Außerdem kann auf die einzelnen Elemente

des Feldes mit dem Indexoperator `[ ]` zugegriffen werden. Erlaubte Argumente für den Indexoperator `[ ]` sind positive ganze Zahlen.

Es sei nocheinmal darauf hingewiesen, dass Felder wie wir sie bisher eingeführt haben, konstante Länge haben. So führt folgender Quelltext zu einer Fehlermeldung

```
1  int n = 5;
2  int a[n];
```

da die Variable `n` nicht konstant ist. Dagegen ist folgendes korrekt

```
1  const int n = 5;
2  int a[n];
3  for(int i = 0; i < n; i++) {
4    a[i] = i+1;
5  }
```

was es erlaubt die Feldlänge in einer Variablen zu speichern. Wenn sich nun im Quelltext diese Länge ändert, so muss man diese Änderung nur noch an einer Stelle vornehmen.

1.6.2 Eindimensionale Felder als Funktionenargumente

Wie schon im Abschnitt über Funktionen angedeutet, kann man auch Felder als Funktionenargument verwenden. Die Schreibweise wird am folgenden Beispiel für eine Funktion erklärt, die die Quadratsumme aller Elemente eines Feldes erzeugt (das Quadrat der Norm)

```
1  double sqrsum(double a[], const int n) {
2    double sum = 0;
3    for(int i = 0; i < n; i++) {
4      sum += a[i]*a[i];
5    }
6    return sum;
7  }
```

Die Schreibweise `a[]` teilt dem Compiler mit, dass es sich um ein Feld handelt. Es ist sehr wichtig hier zu verstehen, dass der Compiler aber keine Möglichkeit hat zu überprüfen, ob innerhalb der Funktion nur auf die vorhandenen Elemente von `a` zugegriffen wird. Man muss als Programmierer also darauf achten, dass diese Information verfügbar gemacht wird und nicht verlorengeht. In diesem Fall bedeutet das, dass `n` die richtige Länge als Wert enthalten muss. Folgender Quelltext ist korrekter `C` Quelltext

```
1 int list[5];
2 int i = 5;
3 list[i] = 3;
```

und wird vom Compiler anstandslos übersetzt. Im besten Fall erhält man dann bei der Ausführung dieses Codes einen *segmentation fault*. Im schlechtesten Fall ist `list[5]` Speicher, auf den das Programm zugreifen kann. Dann erhält man keinen Laufzeitfehler und modifiziert ungewollt Speicher, den man nicht modifizieren will.

Dies kann zu sehr seltsamem Verhalten des Programms führen, und das Finden eines solchen Fehlers ist sehr schwierig. Deshalb sollte man Indizierungen immer mit großer Sorgfalt überprüfen.

Es gibt noch einen weiteren wichtigen Punkt, der später noch genauer diskutiert wird. Für Felder, die man wie oben beschrieben an Funktionen übergibt, wird keine Kopie angelegt. Das heißt insbesondere, dass sich das Originalfeld ändert, wenn man innerhalb der Funktion das Feld modifiziert. Genauer wird das im Abschnitt über Zeiger diskutiert.

1.6.3 Mehrdimensionale Felder

Ganz analog zu eindimensionalen Feldern lassen sich auch zweidimensionale Felder (Matrizen) erzeugen

```
1  int A[3][5]; // Deklaration
2  for(int i = 0; i < 3; i++) {
3    for(int j = 0; j < 5; j++) {
4      A[i][j] = i+j;  // Initialisierung
5    }
6  }
```

Und wie im eindimensionalen Fall kann man auch Deklaration und Initialisierung kombinieren

```
1  int A1[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
2  int A2[][3]  = {{1,2,3}, {4,5,6}, {7,8,9}};
3  int A3[][3] = {1,2,3,4,5,6,7,8,9}};
```

Die Anzahl der Zeilen kann der Compiler aus der Initialisierungsliste bestimmen. Die Anzahl der Spalten muss aber immer angegeben sein. Die Dimensionalität eines Feldes kann dann in analoger Weise zu drei oder mehr erweitert werden, wobei der C-Compiler die Feldelemente immer linear so im Speicher ablegt, wie für das Beispiel `A1` in Abbildung 1.1 gezeigt. Das heißt, dass die Dimensionen des Arrays von rechts nach links zählend linear im Speicher abgelegt werden.

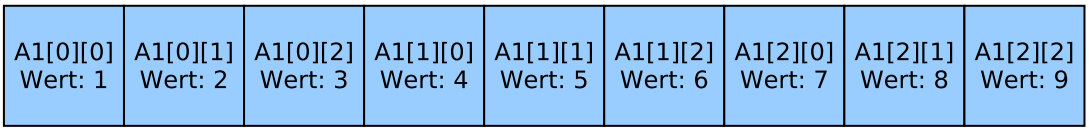


Abbildung 1.1: Das zweidimensionale Array `A1` im Speicher.

Die Benutzung von Feldern wird in folgendem Beispiel illustriert:

Beispiel: Berechnung des Mittelwerts und der Varianz

Die Berechnung des Mittelwerts einer Datenreihe ist ein gutes Beispiel für die Benutzung von Arrays. Nehmen wir an, wir haben die folgenden Daten

gegeben:

```
double data[] = {1.3, 2.4, 5.3, 2.4, 6.7, 3.5, 6.9,
                 1.3, 1.4, 4.5, 5.5, 5.3, 6.7, 2.1,
                 2.4, 3.3, 7.9, 0.3, 3.3, 1.5};
```

und wir wollen den Mittelwert dieser Daten berechnen. Folgendes Programm übernimmt diese Aufgabe:

```
#include <stdio.h>

int main()
{
  const int size = 20;
  double data[] = {1.3, 2.4, 5.3, 2.4, 6.7, 3.5, 6.9,
                   1.3, 1.4, 4.5, 5.5, 5.3, 6.7, 2.1,
                   2.4, 3.3, 7.9, 0.3, 3.3, 1.5};

  // initialisiere mean zu 0
  double mean = 0.;
  for (int i = 0; i < size; i++)
    {
      mean += data[i];
    }
  mean /= (double)size;
  printf("Der Mittelwert ist %e\n", mean);
  return (0);
}
```

Für die Varianz müssen wir auch noch die Quadrate aufsummieren. Wir modifizieren dafür die Schleife wie folgt:

```
double mean = 0., xsq = 0.;
for (int i = 0; i < size; i++)
  {
    mean += data[i];
    xsq += data[i] * data[i];
  }
mean /= (double)size;
```

Die Varianz können wir dann wie folgt berechnen und ausgeben:

```
double var = xsq / (double)size - mean * mean;
printf("Die Varianz ist %e\n", var);
```

1.6.4 Zeichenketten oder Strings

In `C` gibt es keinen elementaren Datentyp für Zeichenketten, sogenannte *strings*. Zeichenketten werden mit Hilfe von Arrays abgebildet. Ein Zeichen kann in einer Variablen vom Typ `char` gespeichert werden. Die Größe des Datentyps `char` ist

immer ein Byte, also 8 Bit.[2]

Eine Zeichenkette kann also durch eine Array von Elementen vom Typ `char` erzeugt werden. Das Ende einer Zeichenkette wird durch das Zeichen `\0` angegeben. Man spricht auch von *null-terminiert* oder *0-terminiert* und das Zeichen `\0` wird auch *Nullterminierungszeichen* genannt. Im nächsten Beispiel überprüfen wir, ob eine Zeichenkette eine Zahl enthält:

Beispiel: Durchsuchen von Zeichenketten

```
#include <stdio.h>

int main()
{
  char string[] = "sdfk99225kljsdfs\0";
  int i = 0;
  int ergebnis = 0;

  while(1) {
    if(string[i] == '\0') {
      break;
    }
    if((string[i]) >= '0' && (string[i] <= '9')) {
      ergebnis = 1;
      break;
    }
    i++;
  }
  if (ergebnis){
    printf("Der String %s enthaelt mindestens eine Zahl\n", string
    );
  } else {
    printf("Der String %s enthaelt keine Zahlen\n", string);
  }
  return 0;
}
```

[2] **Bemerkung:** Im ASCII Zeichensatz können $2^8 = 256$ verschiedene Zeichen gespeichert werden. Dies reicht für Englisch und ein paar Steuerzeichen, jedoch nicht für alle Sprachen dieser Welt. Der ASCII Zeichensatz nutzt die ersten 128 Zustände (also die ersten 7 Bit) für das im Englischen genutzte Alphabet. Die restlichen 128 Zustände werden abhängig vom *Encoding* interpretiert. Für Deutsch kann man `latin-1` nutzen. Je nach Encoding wird, z.B., ein Zeichen mit Wert 204 als solches oder jenes interpretiert. Ist das Encoding nicht das richtige, erscheinen z.B. Zeichen mit Umlauten nicht korrekt und scheinbar willkürliche Zeichen stehen an ihrer Stelle. Sprachen, die mehr als 128 verschiedene Zeichen benötigen, können im oberen Teil von ASCII überhaupt nicht dargestellt werden. Die Einsicht, dass es mehr als 256 verschiedene Zeichen gibt, wurde im Unicode Standard manifestiert. Die Konsequenz ist jedoch, dass jetzt mehr als ein Byte pro Zeichen benötigt wird. UTF-8 ist inzwischen vielerorts das Standard-Encoding, sodass beliebig viele verschiedene Zeichen in einer Textdatei genutzt werden können. Der Preis ist jedoch, dass ein Buchstabe jetzt beliebig viele Byte (meist eins) nutzt. Da hier der Fokus allerdings auf Numerischen Methoden liegt, wird nicht weiter auf die vielfältigen Probleme mit Encodings eingegangen.

Wir deklarieren und initialisieren zunächst die Variable `string` mit einer Zeichenkette. Dann nutzen wir eine `while` Schleife mit konstant wahrem logischem Ausdruck. Die Schleife wird dann mit `break` abgebrochen, wenn das Endstring-Zeichen gefunden wurde. In der Schleife wird dann jedes Element der Kette darauf überprüft, ob es eine Zahl ist. Dementsprechend wird die Variable `ergebnis` gesetzt.

Formatierte Erstellung von Zeichenketten mit `sprintf` und `snprintf`

Die Manipulation von Zeichenketten ist oft ein wichtiger Bestandteil eines Programms, zum Beispiel um Ausgabedateinamen abhängig vom Wert einer Variable zu machen.

Wie haben die Funktion `printf` schon öfter zur Textausgabe verwendet und werden im Detail in Teil 2.2 auf die *formatierte* Ausgabe eingehen. Möchte man jedoch formatierte Textdarstellungen von Variablen nicht auf die Konsole, sondern direkt in eine Zeichenkette ausgeben, so stehen in `C` die beiden Funktionen `sprintf` und `snprintf` zur Verfügung. Diese ähneln in ihrer Benutzung der bekannten `printf` Funktion und werden mit der Headerdatei `stdio.h` eingebunden. Im Gegensatz zu `printf`, schreiben diese Funktionen jedoch direkt in eine sich im Speicher befindliche Zeichenkette.

Diese Funktionen haben die Signaturen

```
1  int sprintf(char *str, const char *format, ...);
2  int snprintf(char *str, size_t size, const char *format, ...);
```

wobei das Argument „`str`“ für den Speicherbereich der Zeichenkette steht, in die geschrieben werden soll und „`format`“ eine konstante Zeichenkette ist, in der wir beschreiben werden, wie dies zu erfolgen hat. Die Bedeutung von „`char *`“ und „`const char *`“ wird in Teil 2.1 noch genau erklärt werden. Die sogenannte *variable Argumentenliste*, „`...`“, dient bei den Funktionen aus `stdio.h` als Platzhalter für eine beliebige Anzahl von Variablen oder Konstanten, die man ausgeben möchte. Bei `snprintf` steht das Argument „`size`“ für die maximale Anzahl an Zeichen (inklusive des Nullterminierungszeichens), welche maximal in die Zeichenkette `str` geschrieben werden sollen. Die Stelle in der Zeichenkette, an denen diese Ausgabe stattfinden soll, sowie das Format, in der diese Ausgabe erfolgen soll, geben wir anhand von Platzhaltern an, genau so, wie wir es auch schon bei `printf` getan haben.

Da wir jedoch nicht auf die Konsole ausgeben, sondern direkt in Speicher schreiben, muss man bei der Verwendung insbesondere von `sprintf` sehr vorsichtig sein, wie in folgendem Beispiel gezeigt wird. Grundsätzlich ist `snprintf` zu bevorzugen.

Beispiel: Formatierte Erstellung von Zeichenketten mit `sprintf` und `snprintf`

```
1  #include <stdio.h>
2  // konstante Längen für lange und kurze Zeichenketten
3  const int MAX_LENGTH = 1000;
4  const int SHORT_LENGTH = 50;
5
6  int main()
```

```
7  {
8    char string[MAX_LENGTH];
9    const int i = 42;
10   sprintf(string,
11           "The Answer to the Ultimate Question of Life,"
12           "The Universe, and Everything is %d.\n\n",
13           i);
14   printf("string: %s", string);
15
16   int rval;
17   char short_string[SHORT_LENGTH];
18
19   // kurze Zeichenkette wird in short_string geschrieben
20   // Rueckgaberwert wird ueberprueft
21   rval = snprintf(short_string,
22                   SHORT_LENGTH-1,
23                   "This is a test string.\n\n");
24   if(rval >= SHORT_LENGTH) {
25     printf("snprintf: The string was not completely written!\n");
26   } else {
27     printf("snprintf: wrote %d characters\n", rval);
28   }
29   printf("short_string: %s",short_string);
30
31   // ueberlange Zeichenkette wird in short_string geschrieben
32   // Rueckgaberwert wird ueberprueft
33   rval = snprintf(short_string,
34                   SHORT_LENGTH-1,
35                   "The Answer to the Ultimate Question of Life, "
36                   "The Universe, and Everything is %d.\n",
37                   i);
38
39   if(rval >= SHORT_LENGTH) {
40     printf("snprintf: The string was not completely written!\n");
41   }
42
43   printf("short_string: %s\n\n",short_string);
44
45   // ACHTUNG: hier wird fremder Speicher ueberschrieben
46   sprintf(short_string,
47           "The Answer to the Ultimate Question of Life"
48           ", The Universe, and Everything is %d.\n",
49           i);
50   printf("short_string: %s",short_string);
51
52   return 0;
53 }
```

Ausgabe des Beispielprogramms:

```
1 $ ./test
2 string: The Answer to the Ultimate Question of Life, The Universe,
```

```
     and Everything is 42.

snprintf: wrote 24 characters
short_string: This is a test string.

snprintf: The string was not completely written!
short_string: The Answer to the Ultimate Question of Life, The

short_string: The Answer to the Ultimate Question of Life, The
    Universe, and Everything is 42.
```

Ebenso wie bei `scanf`, besteht die Gefahr eines buffer overflows, wenn mit `sprintf` mehr Zeichen geschrieben werden, als eigentlich allokiert wurden. Dies resultiert im schlimmsten Fall *nicht* in einem Programmabsturz sondern darin, dass irgendeine Speicherstelle überschrieben wird.

Die Verwendung von `snprintf` hat drei wesentliche Vorteile:

- Solange man weiß, wie lang das zu beschreibende Feld ist, kann man sicherstellen, dass die Länge des Feldes nicht überschritten wird.
- Versucht man trotzdem eine längere Zeichenkette zu schreiben, wird diese nicht vollständig geschrieben, wobei als letztes Zeichen der Nullterminierer geschrieben wird. Die reservierte Speicherstelle zeigt also immer auf einen gültigen String mit Nullterminierer.
- Durch Überprüfung des Rückgabewertes kann sichergestellt werden, ob die Zeichenkette in den vorgesehenen Speicher passte und falls nicht, kann darauf reagiert werden.

Wie man an der folgenden Ausgabe sehen kann, schreibt das Programm in der Anweisung in den Zeilen 46 bis 49 in fremden Speicher. `printf` liest bei Angabe von `%s` die angegebene Speicherstelle immer bis zum nächsten Nullterminierungszeichen aus.

Anmerkung: konstante Zeichenketten teilen

Im vorherigen Beispiel haben wir uns der Eigenschaft konstanter Zeichenketten in `C` bedient, dass mehrere, aufeinander folgende und nicht mit Kommata getrennte Zeichenketten zu einer einzigen Zeichenkette Zusammengefasst werden. Der Compiler interpretiert

```
  "Zeichenkette mit Platzhalter: %d "
  "Zweite Zeichenkette"
```

also als

```
  "Zeichenkette mit Platzhalter: %d Zweite Zeichenkette"
```

was es uns erlaubt, den Quelltext für das Beispielprogramm durch Zeilenumbrüche sauberer und verständlicher zu gestalten.

1.7 Die Bibliotheken `math.h` und `complex.h`

Die Sprache `C` hat neben den Sprachbestandteilen auch eine umfangreiche Liste von Standardbibliotheken. Sie sind im `ANSI C` Standard festgelegt. Diejenige für die Ein- und Ausgabe haben wir schon kennengelernt. Wir werden jetzt noch zwei weitere kurz einführen.

1.7.1 Mathematische Bibliothek `math.h`

Erstens die Bibliothek `math.h`. Sie stellt mathematische Funktionen, wie beispielsweise trigonometrische Funktionen, aber auch eine Funktion für den Betrag zur Verfügung. Natürlich berechnen diese Funktionen das Ergebnis nur zur gewünschten Genauigkeit. Wir illustrieren dies am Bespiel vom Cosinus. Die Cosinus Funktion ist auf ganz $\mathbb{R}$ durch eine Taylorreihe um 0 definiert.

$$\cos(x) = \sum_{l=0}^{\infty} (-1)^l \frac{x^{2l}}{(2l)!}. \tag{1.5}$$

Die Funktion

```
1  double cos(double x);
```

berechnet den Cosinus in `double` Genauigkeit. Im folgenden Quelltext berechnen wir den Cosinus einmal mit Hilfe der Funktion aus `math.h` und einmal über die Reihenentwicklung, die wir zu einer in der Kommandozeile angegebenen Ordnung abbrechen:

```
1  #include <math.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(int argc, char *argv[])
6  {
7    double winkel;
8    double coswinkel;
9    // genuegend Kommandozeilenparameter?
10   if (argc < 2) {
11     printf("Usage: ./print_cos Winkel Naeherungsordnung[optional]\n");
12     return(-1);
13   }
14   winkel = atof(argv[1]);  // Konvertiert eine Zeichenkette in eine
       Fliesskommazahl
15   // cos aus math.h
16   printf("Winkel=%e \t cos(Winkel) = %e\n", winkel, cos(winkel));
17   // nun selbst, wenn die Naeherungsordnung gegeben wurde
18   if (argc > 2) {
19     double coswinkelapprox = 1.; // Das Ergebnis zu fuehrender Ordnung
20     double winkelsqr = winkel * winkel;
21     double partialsum = 1.;      // Die Partialsummen
22     const int n = atoi(argv[2]);           // atoi konvertiert eine
       Zeichenkette in eine ganze Zahl
```

```
23      coswinkelapprox = 1.;
24      int sign=1;
25      for (int i = 1; i < n; ++i) {
26        sign *= -1;              // (-1)^l ist entweder 1 oder -1, wenn l
        gerade oder ungerade ist
27        partialsum *= winkelsqr / ((2 * i - 1) * 2 * i);
28        coswinkelapprox += sign * partialsum;
29      }
30      printf("cos(Winkel) approx %e\n", coswinkelapprox);
31    }
32    return(0);
33  }
```

Listing 1.11: Beispiel zur Verwendung des Cosinus

Wenn auf der Kommandozeile die Ordnung der Näherung als zweiter Kommandozeilenparameter angegeben wurde[3], berechnen wir auch die Näherung selbst nach der Formel Gleichung 1.5. Wie schon vorher erklärt, sind die Kommandozeilenparameter in der `main` Funktion nur als Zeichenkette verfügbar. Die Konvertierung der Ordnung in eine ganze Zahl kann man mit der standard Funktion `atoi` (ASCII to integer) durchführen. Genauso verwenden wir `atof` (ASCII to float) um den eingegebenen Winkel in the Fließkommazahl umzuwandeln. Man beachte, dass die standard Cosinus Funktion das Argument im Bogenmaß erwartet.

Bei der Berechnung der Näherung müssen wir nicht in jedem Schritt die Potenz x^{2i} und Fakultät $(2i)!$ neu berechnen, sonder müssen lediglich die Variablen `zaehler, nenner` entsprechend auffrischen. Dies ist ein generelles Konzept: Man verwendet mehr Speicher um Rechnung zu sparen.

Übersetzen und Linken mit `math.h`

Das Programm kann man wie folgt übersetzen

```
>$  gcc -std=c99 -Wall -pedantic cosineexample.c -o cos.exe -lm
```

Man beachte, dass man nun explizit die Bibliothek `math.h` dazulinken muss. Dies geschieht mit dem Argument `-lm`. In Tabelle 1.6 listen wir einige wichtige Funktionen auf, die `math.h` bereitstellt.

1.7.2 Komplexe Zahlen mit `complex.h`

Als zweite Standardbibliothek weisen wir noch auf `complex.h` hin. Sie stellt einen komplexen Datentyp zur Verfügung, der leider nicht Teil von C selbst ist. Die Benutzung ist dann aber gleich zu `C` Datentypen, wie man an folgendem Beispiel sieht:

[3] die Bedeutung von `argc` und `argv` wird in Abschnitt 2.1.2 erläutert

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(int argc, char *argv[])
{
  double complex z,v,w;

  // I ist die rein komplexe Zahl i
  z = 1 + I*3;
  v = 0.3*I;
  // komplexe Multiplikation
  w = z*v;
  // Zugriff auf Real- und Imaginaerteile
  printf(``w = %e + i*%e\n'', creal(w), cimag(w));
  // Anwendung der komplexen Exponetialfunktion
  z = cexp(w);
  printf(``w = %e + i*%e\n'', creal(z), cimag(z));
  return 0;
}
```

Listing 1.12: Beispiel für komplexe Zahlen

Mathematische Funktionen für komplexe Zahlen

In `math.h` sind wiederum die Funktionen aus Tabelle 1.6 für den Typ `double complex` definiert. Meistens unterscheiden sie sich von den reellwertigen Funktionen nur duch ein vorgestelltes `'c'`, wie zum Beispiel bei `cexp`. Auf Real- und Imaginärteil kann mit `creal` und `cimag` zugegriffen werden. `complex.h` stellt auch Typen `float complex` und `long double complex` bereit. Die entsprechenden Exponentialfunktionen heißen dann `cexpf` und `cexpl`. Das komplex konjugierte erhält man mit `conj`, bzw. `conjf` und `conjl`.

`Deklaration`	`Rückgabewert`
`double acos(double x);`	Arcuscosinus von `x`
`double asin(double x);`	Arcussinus von `x`
`double atan(double x);`	Arcustangenz von `x`
`double atan2(double y,double x);`	Arcustangenz von `x/y`
`double ceil(double x);`	Kleinste ganze Zahl, welche $\geq$ `x` ist
`double cos(double x);`	Cosinus von `x`
`double cosh(double x);`	Cosinus hyperbolicus von `x`
`double exp(double x);`	e^x wobei $e = 2.7182..$
`double fabs(double x);`	$\|x\|$
`double floor(double x);`	Größte ganze Zahl, welche $\leq$ `x` ist
`double ldexp(double x, double y);`	xe^y
`double log(double x);`	Natürlicher Logarithmus von `x`
`double log10(double x);`	Zehnerlogarithmus von `x`
`double pow(double x, double y);`	x^y
`double sin(double x);`	Sinus von `x`
`double sinh(double x);`	Sinus hyperbolicus von `x`
`double sqrt(double x);`	$\sqrt{x}$
`double tan(double x);`	Tangens von `x`
`double tanh(double x);`	Tangens hyperbolikus von `x`

Tabelle 1.6: Einige Funktionen, die in `math.h` deklariert sind.

2 Zeiger und Co

Im folgenden Kapitel werden zum Abschluss noch die beiden wichtigen Themen Zeiger und die damit verbundene dynamische Speicherverwaltung und zusammengesetzte Datentypen eingeführt. Dabei handelt es sich um etwas fortgeschrittenere Konzepte, die aber auch die wahre Stärke von `C` ausmachen. Mit Hilfe von Zeigern wird auch eine Implementierung vom Algorithmus *Einfügesortieren* gegeben.

2.1 Zeiger

Kommen wir nun zu der Frage, wie man aus Funktionen mehr als nur skalare Größen zurück gibt. Genauso, wie wir Felder an Funktionen uebergeben können, wollen wir auch Felder als Rückgabewert benutzen. Dies geht in `C` nicht direkt, sondern nur über den Umweg von sogenannten Zeigern. Zeiger sind mit das mächtigste Sprachelement in `C`. Sie sind allerdings auch für jede Menge Verwirrung und Programmierfehler verantworlich.

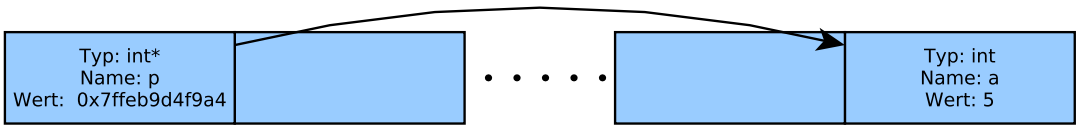


Abbildung 2.1: Illustration Zeiger.

Um Zeiger verstehen zu können, muss man sich zunächst klar machen, dass eine Variable im Prinzip aus zwei Dingen besteht: Einerseits dem Typ der Variablen und andererseits der Speicheradresse, unter der ihr Wert abgelegt wird. Über den Typ der Variablen weiß der Compiler, wie viele Bits im Speicher belegt sind, und wenn die Anfangsadresse für das erste Bit bekannt ist, dann kann die gesamte Anzahl an Bits ausgelesen und dem Typ entsprechend interpretiert werden. `C` erlaubt Zugriff sowohl auf den Wert einer Variablen, als auch auf die Adresse, an der der Wert abgelegt ist. Unterschieden wird zwischen den beiden mit dem `&` Operator. Für eine Variable `n` repräsentiert `n` den Wert und `&n` die Speicheradresse. Bei `&n` spricht man auch von Zeiger (*pointer*) und bei `&` vom Adressoperator. Mit Hilfe von `printf` kann man den Unterschied ausgeben

```
1  int a = 5;
2  int * p = &a;
3  printf("Der Wert %d ist an der Adresse %p abgelegt.\n", a, p);
```

was eine Ausgabe wie die folgende erzeugt

```
Der Wert 5 ist an der Adresse 0x7ffeb9d4f9a4 abgelegt.
```

Die Adresse wird hier als *Hexadezimalzahl* `0x7ffeb9d4f9a4` ausgegeben. Die Beziehung von `p` und `a` ist in Abbildung 2.1 illustriert.

Da Zeiger in `C` eine sehr wichtige Rolle spielen, gibt es für sie spezielle Datentypen, wie im letzten Beispiel schon gesehen:

```
1 int n = 3; // eine Variable vom Typ int
2 int *address; // eine Variable vom Typ Zeiger auf int
3 address = &n; // address zeigt auf n
4 *address = 5; // *address representiert den Wert, der unter der Adresse
      address
5                 // gespeichert ist. man spricht von dereferenzieren
6 n == 5; // ist jetzt wahr.
```

Genauso gibt es einen Typ Zeiger auf `double`, nämlich `double*` und so weiter. Allgemein ist ein Zeiger eine Variable, die eine Adresse als Wert zusammen mit einem Datentyp speichert. Wir demonstrieren die Nützlichkeit von Zeigern zunächst an einem Beispiel. Wir definieren eine Funktion, die zwei Parameter `a, b` übergeben bekommt und diese jeweils um eins erhöht. Anschließend sollen diese beiden neuen Werte zurückgegeben werden.

Ein Weg, um dies zu tun, ist das sogenannte *call by reference*. Was **nicht** funktioniert ist das folgende:

```
1  #include <stdio.h>
2  // einfache Funktion um a,b um 1 zu erhoehen
3  void increment(int a, int b)
4  {
5    a++;
6    b++;
7    return;
8  }
9
10 int main()
11 {
12   int a = 2, b = 3;
13   printf("Wert vor der Funktion a=%d, b=%d\n", a, b);
14   increment(a, b); // die Variablen in unserem Block werden
15                            // nicht veraendert!
16   printf("Wert nach der Funktion a=%d, b=%d\n", a, b);
17   return 0;
18 }
```

Der Grund ist, dass in der Funktion `increment` neue Variablen `a`, `b` angelegt werden, die nur in der Funktion selbst sichtbar sind. Sie haben also nichts mit den Variablen `a`, `b` in der Funktion `main` gemein, außer dem Namen. Man spricht hier von *call by value*, da die Variablen `a, b` in der Funktion `increment` mit den Werten der Variablen aus `main` initialisiert werden. Deswegen liefern die `printf` Aufrufe in den Zeilen 12 und 15 das gleiche Ergebnis. Denn die Variablen `a`, `b` in `main` wurden nicht verändert.

Bei *call by reference* wird an Stelle des Wertes die Adresse übergeben.

```
1 #include <stdio.h>
2 // einfache Funktion um a,b um 1 zu erhoehen
3 void increment(int *a, int *b)
4 {
5   (*a)++;
6   (*b)++;
7   return;
8 }
9
10 int main()
11 {
12   int a = 2, b = 3;
13   printf("Wert vor der Funktion a=%d, b=%d\n", a, b);
14   increment(&a, &b); // die Variablen in unserem Block werden
15                              // direkt veraendert!
16   printf("Wert nach der Funktion a=%d, b=%d\n", a, b);
17   return 0;
18 }
```

Die Funktion bekommt also zwei Parameter vom Typ `int*`, also Zeiger auf `int`. Dann wird mit dem Dereferenzierungsoperator `*` der Wert, der unter den beiden Adressen gespeichert ist, um eins erhöht. Das geschieht unter der Annahme, dass dort eine Variable vom Typ `int` abgelegt ist. Damit werden also direkt die Werte der Variable `a, b` in `main` verändert.

Zeiger kann man genau wie andere Variablen nutzen (womit auch klar ist, dass man auch Zeiger auf Zeiger definieren kann). Folgendes Beispiel illustriert die Nutzung noch einmal, wobei `NULL` der Nullzeiger ist, welcher auf eine reservierte Adresse zeigt, die nicht dereferenziert werden kann:

```
1 #include <stdio.h>
2
3 int main()
4 {
5   int q = 10;
6   int *p = NULL;
7   p = &q;
8   printf("Der Wert an der Adresse %p ist:%d\n", p, *p);
9   return 0;
10 }
```

In der fünften Zeile haben wir eine Variable vom Typ `int` deklariert und mit dem Wert 10 initialisiert, in der sechsten Zeile eine Variable vom Typ `int*`, die mit dem `NULL` Zeiger initialisiert wurde. In der siebten Zeile wird dann `p` auf die Adresse von `q` gesetzt. Damit liefert die Dereferenzierung von `p`, also `*p`, den Wert von `q`. Zeigern dürfen nur gültige Adressen zugewiesen werden. Dies kann allerdings, bis auf Ausnahmen, nicht vom Compiler überprüft werden. Wenn doch keine gültige Adresse zugewiesen wurde, bekommt man bei Dereferenzierung den *segmentation fault* als Laufzeitfehler. Dieser Quelltext weist sehr wahrscheinlich eine ungültige Adresse zu:

```
1 int *p = 42;
```

Allerdings wird in diesem Fall der Compiler sehr wahrscheinlich[1] eine Warnung geben, denn 42 ist vom Typ `int`, und nicht vom Typ `int*`. GCC 6.3.1 gibt folgendes aus:

```
int-ptr.c:7:14: Warnung: Initialisierung erzeugt Zeiger
           von Ganzzahl ohne Typkonvertierung [-Wint-conversion]
     int *p = 42;
              ^~
```

Wenn der Compiler sich hier nicht beschwert, sollte man die Dokumentation studieren, um heraus zu finden, wie man diesen Typ von Warnung anschalten kann, oder, wenn das nicht möglich ist, den Compiler wechseln.

2.1.1 Zeiger und Felder

Wie schon im Abschnitt über Felder angedeutet sind Felder und Zeiger in `C` eng verwandt. Nehmen wir an, es wurde ein Feld wie oben eingeführt deklariert

```
1   int n[] = {1, 2, 3, 4, 5};
```

Dann ist die Variable `n` eng verwandt mit dem Typ `int *`, also Zeiger auf `int`. Den subtilen Unterschied kann man am ehesten damit erklären, dass `n` konstant ist, also die Adresse nicht verändert werden darf. Das bedeutet, dass folgender `C` Quelltext korrekt ist

```
1   int n[] = {1, 2, 3, 4, 5};
2   int * p = n;
```

und auch die folgenden zwei Funktionsdeklarationen äquivalent sind

```
1   void f1(int * a, const int n);
2   void f2(int a[], const int n);
```

Daraus folgt auch, dass ein Feld in `C` immer *by reference* übergeben wird. Wird in Funktion `f1` oder `f2` in das Feld `a` geschrieben, so wird das ursprüngliche Feld aus dem aufrufenden Quelltextblock modifiziert. Obiges Beispiel kann man also auch wie folgt schreiben:

[1] Hängt leider vom Compiler ab.

```c
#include <stdio.h>
// einfache Funktion um a,b um 1 zu erhoehen
void increment(int *a, const int n)
{
  for(int i = 0; i < n; i++) {
    a[i]++;
  }
  return;
}

int main()
{
  int a[] = {2, 3};
  printf("Wert vor der Funktion a[0]=%d, a[1]=%d\n", a[0], a[1]);
  increment(a, 2); // die Variablen in unserem Block werden
                           // direkt veraendert!
  printf("Wert nach der Funktion a[0]=%d, a[1]=%d\n", a[0], a[1]);
  return 0;
}
```

Wenn das übergebene Feld innerhalb einer Funktion nicht verändert werden soll, so kann man es als `const` deklarieren

```c
void f(const int * a, const int n);
```

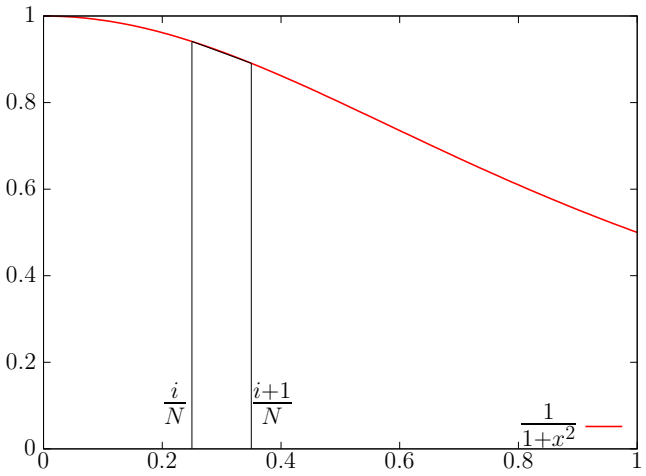
Als etwas ausführlicheres Beispiel für die Verwendung von Feldern, dient uns folgende numerische Abschätzung der Zahl π:

Beispiel: `Näherung von` π

In diesem Beispiel berechnen wir π näherungsweise. Dafür verwenden wir folgende Integraldarstellung von π:

$$\pi = 4 \cdot \int_0^1 \mathrm{d}x \frac{1}{1+x^2} \tag{2.1}$$

Das Integral berechnen wir numerisch, indem wir die Fläche unterhalb der Kurve als eine Summe abschätzen. Dabei bedienen wir uns der sogenannten Trapez-Regel. Wir teilen das Interval $[0, 1]$ in N gleichlange Unterintervalle auf. Den jeweilige linke Punkt des Intervals nennen wir x_i, $i = 0, ..., N$, wobei $x_n - x_{n-1} = \Delta = \text{const}$. Auf jedem Unterintervall approximieren wir die Funktion linear, wie in folgender Abbildung dargestellt ist:



In unserem Fall sind die Punkte wie folgt gegeben

$$x_i = i/N\,, \qquad i = 0, ..., N-1\,.$$

Definieren wir folgende Funktion

$$f(x) = \frac{1}{1+x^2}\,,$$

so können wir wie folgt über die Teilergebnisse summieren:

$$\int_0^1 \mathrm{d}x \frac{1}{1+x^2} \approx \sum_{i=0}^{N-1} \frac{1}{2N}[f(x_i) + f(x_{i+1})] \tag{2.2}$$

Hier ist der entsprechende `C` Quelltext, der Arrays benutzt:

```c
#include <stdio.h>
const int MAX = 10000;

int main()
{
  int N = 0;
  double f[MAX], x[MAX]; // double arrays der Laenge MAX
  scanf("%d", &N);
  // Pruefe die Eingabe
  if (N >= MAX)
    {
      printf("Fehler, zu vielen Stuetzstellen\n");
      return (-1);
    }
  if (N < 0)
    {
      printf("N muss groesser als 0 sein!\n");
      return (-2);
    }
```

```
20    for (int i = 0; i < N; ++i)
21      {
22        x[i] = (double)i / (double)N;
23        f[i] = 1. / (1 + x[i] * x[i]);
24      }
25    double summe = 0.;
26    for (int i = 0; i < N - 1; ++i)
27      {
28        summe += (f[i] + f[i + 1]);
29      }
30    summe += f[N - 1] + 0.5; // Randterm
31    printf("Die Naeherung von pi ist =%e\n", 2. / N * summe);
32    return (0);
33  }
```

Neben der Benutzung von Arrays, haben wir noch ein weiteres neues Konzept eingeführt. Bei der Division von i/N, beide vom Typ `int`, in Zeile 22 haben wir einen expliziten *cast* nach `double` durchgeführt, damit keine Division ganzer Zahlen durchgeführt wird.
Dieses Beispiel hätten wir natürlich auch ohne Arrays durchführen können, aber es illustriert deren Benutzung.

2.1.2 Die Parameter der `main` Funktion

An dieser Stelle können wir jetzt auch die möglichen Parameter der Funktion `main` einführen. Die ist nämlich allgemein wie folgt deklariert:

```
1 int main(int argc, char *argv[]);
```

An der Kommandozeile kann man mit Hilfe der Funktionsargumente von `main` dem Programm Parameter übergeben. Die Funktion `main` erhält diese in Form einer Zeichkette. Leerzeichen in dieser Zeichenkette werden als Trennzeichen interpretiert, so dass die Zeichenkette `argc` Wörter enthält. Das erste Wort ist immer der Name der Programms. Die Wörter werden in `argv` gespeichert. Also, zum Bespiel

```
./main.exe 3 hallo
```

liefert

- `argc=3`
- `argv[0] = "./main.exe"`
- `argv[1] = "3"`
- `argv[2] = "hallo"`

2.1.3 Zeigerarithmetik

Man kann auch eine Zeichenkette initialisieren:

```
#include <stdio.h>
int main()
{
  char string[] = {'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd
    ', '\0'};
  char *pointer = NULL;
  pointer = &string[6];
  printf("Das siebte Zeichen im String ist %c\n", *pointer);
  return (0);
}
```

und dann mit `pointer` auf einzelne Elemente der Zeichenkette zugreifen. Das ist nicht nur ein alternativer Weg, um auf die Elemente eines Arrays zuzugreifen. `C` stellt Zeiger intern als ganze Zahlen dar und auf jedem Zeigertyp sind auch arithmetische Operationen definiert. Beispielsweise ist folgendes in `C` korrekter Quelltext:

```
int list[5];
int *plist = NULL;
plist = list; // aequivalent zu plist = &list[0];
for (int i = 0; i < 5; i++)
  {
    *plist = i;
    plist++; // aequivalent zu plist = plist + 1; oder plist += 1;
  }
```

Mit unserer bisherigen Kenntnis des Operators `++` würden wir erwarten, dass der Wert von `plist` um eins erhöht wird. Das ist im Prinzip auch richtig, allerdings findet die Erhöhung in Einheiten der Länge des Typs, auf den der Zeiger zeigt. Und damit zeigt `plist++` auf das nächste Element in der Liste `list`. Denn `C` reserviert für `int list[5];` einen zusammenhängenden Speicherbereich der fünffachen Länge von `int`. `list`, ohne den Indexoperator, ist selbst vom Typ `int*`, und zeigt auf den Anfang dieses Speicherbereichs. `list[3]` ist dann äquivalent zu folgender Dereferenzierung: `*(list + 3)`. Und damit weißt obiger Beispielcode dem $i$ten Element von `list` den Wert i zu. Genauso kann man den Indexoperator für Zeiger verwenden. Im obigen Beispiel hätten wir auch

```
int list[5];
int *plist = NULL;
plist = list; // equivalent zu plist = &list[0];
for (int i = 0; i < 5; i++)
  {
    plist[i] = i;
  }
```

schreiben können. Im folgenden Beispiel finden sich einige der möglichen arithmetischen Operationen für Zeiger:

```
#include <stdio.h>

```

```
3  int main()
4  {
5    int sqnum[] = {1, 4, 9, 16, 25, 36, 49};
6    int *pointer;
7    int *pointer2;
8    pointer = sqnum;
9    pointer++;
10   printf("Nach der Inkrementierung des Zeigers %d\n", *pointer);
11   pointer--;
12   printf("Nach der Dekrementierung des Zeigers %d\n", *pointer);
13   pointer += 2;
14   printf("Nach dem Hinzufuegen von 2 zum Zeiger %d\n", *pointer);
15   pointer -= 2;
16   printf("Nach dem Subtrahieren von 2 vom Zeiger %d\n", *pointer);
17   ++pointer;
18   pointer2 = &sqnum[4];
19   printf("Zwischen pointer2 und pointer gibt es %ld Elemente\n",
20          pointer2 - pointer);
21   return (0);
22 }
```

Überlegen Sie sich, was obiges Programm als Ausgabe erzeugen wird, bevor Sie diesen Quelltext übersetzen und ausführen lassen.

An dieser Stelle müssen wir auf einen möglichen Fehler hinweisen. Folgender Quelltext wird vom Compiler anstandslos übersetzt

```
1  int main()
2  {
3    int list[5]; // int array
4    double *plist = (double *)list; // explizit cast
5    *(plist + 1) = 3;
6    return 0;
7  }
```

Wenn man Zeile 3 durch

```
1  double *plist = list; // without cast
```

übersetzen das die meisten Compiler auch noch, hoffentlich wenigstens mit einer Warnung. Was ist problematisch an obigen Code? `plist + 1` zeigt nicht auf das zweite Element in `list`. Denn die Länge von `double` und `int` ist nicht identisch. Da `plist` vom Typ Zeiger auf `double` ist, bedeutet `plist + 1` dass die entsprechende Adresse um die Länge von `double` erhöht wird. Damit ist aber der Speicherbereicht von `list` nicht mehr als `int` interpretierbar. Obiger Code wird also undefiniertes Verhalten nach sich ziehen!

Zusammenfassend können wir also die Elemente eines Arrays auf zwei verschiedene Arten und Weisen indizieren

1. mit dem Indexoperator `[]`

2. mit arithmetischen Operationen auf Zeignern

Wie schon gesagt, intern behandelt `C` ein Array quasi als einen Zeiger. Es gibt aber wichtige Unterschiede. Wenn `array` als Array deklariert ist, so darf man dessen Adresse nicht verändern. Folgendes Beispiel erläutert dies:

```
1 #include <stdio.h>
2 int main()
3 {
4   int *pointer;
5   int array[] = {1, 4, 9, 16, 25, 36, 49, 64};
6   int i = 2;
7   pointer = array; // pointer zeigt auf &array[0]
8   printf("Die Werte sind identisch: %d %d\n", array[i], *(pointer + i))
      ;
9   pointer++; // sinnvoll
10  array++; // nicht erlaubt!
11  array = pointer; // nicht erlaubt!
12  return 0;
13 }
```

Zeiger auf konstante Zeichenketten

Abschließend diskutieren wir noch eine Subtilität von Zeigern auf konstante `char` Arrays. Dafür betrachten wir folgenden Beispielcode:

```
1 int main()
2 {
3   char *pointer = "Hello world";
4   pointer[1] = 'a'; // uebersetzt, fuehrt aber zu einem segmentation
      fault
5   return 0;
6 }
```

In der vierten Zeile wird versucht, das zweite Element von `pointer` auf das Zeichen `a` zu setzen. Obwohl dieser Quelltext vom Compiler übersetzt wird, wird es bei der Ausführung zu einem *segmentation fault* kommen. Das liegt daran, dass `pointer` auf einen Bereich im Speicher zeigt, der nicht verändert werden darf. Die Zeichenkette `"Hello world"` wird zur Zeit der Übersetzung in einem Speicherbereich abgelegt, der nur gelesen werden darf und somit nicht verändert werden darf. Hier könnte man mithilfe eines Zeiger auf `const` Abhilfe schaffen, da folgender Code nicht übersetzt und der Fehler somit früh erkannt werden kann:

```
1 int main()
2 {
3   char const * pointer = "Hello world"; // Zeiger auf einen konstanten
      Speicherbereich
4   pointer[1] = 'a'; // Fehler beim Übersetzen
5   return 0;
6 }
```

Zeiger auf konstante Zeichenketten sollten also immmer mithilfe von `const` als solche deklariert werden!

2.2 Ein- und Ausgabe

Ein- und Ausgabe in Dateien, oder allgemein Ein- und Ausgabe auf Geräten ist nicht Teil von `C` selbst. Aber die Standardbibliothek stellt diese Funktionalität zur Verfügung. Sie basiert auf dem Konzept von sogenannten *streams*. Der Name kommt daher, dass Ein- und Ausgabe immer ein serieller Prozess ist. D.h., in Einheiten bestimmter kleinster Elemente werden die Daten sukzessive eingelesen oder ausgegeben. Man kann es sich also als einen Datenstrom vorstellen. Ein stream in `stdio.h` ist immer vom Typ `FILE*`. Die Syntax ist die gleiche, wie bei jedem Datentyp

```
1  FILE * mystream;
```

2.2.1 Standard Ein- und Ausgabe

Es gibt einige vordefinierte streams, so genannte standard streams:

1. `stdin` : Standardeingabe
2. `stdout` : Standardausgabe
3. `stderr` : Standardfehler

Standard Ein- und Ausgabe haben wir schon implizit mit `scanf` und `printf` benutzt. Funktionen, die ebenfalls direkt auf `stdin` und `stdout` arbeiten, sind `getchar` and `putchar`. Sie lesen bzw. schreiben genau ein Zeichen von `stdin` bzw. nach `stdout`. Ihre Deklaration in `stdio.h` sieht wie folgt aus:

```
1 int getchar();
2 int putchar(int c);
```

In `putchar` wird ein impliziter *cast* von `c` nach `unsigned char` durchgeführt. `getchar` liest ein Zeichen als `unsigned char` vom `stdin` und führt dann einen impliziten *cast* nach `int` durch. Damit kann man beispielsweise ein Programm schreiben, dass alle über die Standardeingabe eingegenen Zeichen direkt wieder auf die Standardausgabe `stdout` ausgibt:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6    int c;
7    while ((c = getchar()) != EOF)
8      {
9        putchar(c);
10     }
11   return (0);
12 }
```

Wiederum wird solange eingelesen, bis das Sonderzeichen `EOF` gefunden wird.

2.2.2 Ausgabe in Dateien

Wenn wir direkt aus einer Datei lesen wollen, so müssen wir einen *stream* bekanntmachen, der mit der Datei verbunden ist. Dies geht mit Hilfe der Funktion `fopen`. Man spricht vom Öffnen der Datei. Die Definition von `fopen` sieht wie folgt aus:

```
1 FILE *fopen(const char *filename, char mode);
```

Dabei wird mit `filename` der Name der Datei als string übergeben. `mode` gibt an, für welchen Zweck die Datei geöffnet werden soll:

- `"w"`: Datei zum Schreiben öffnen. Wenn sie schon existiert, wird sie überschrieben. Wenn sie nicht existiert, wird die Datei erzeugt.
- `"r"`: Öffnen einer Datei ausschließlich zum lesen. Der *stream* zeigt auf den Anfang der Datei.
- `"a"`: Öffnen einer Datei zum Schreiben. Wenn sie schon existiert, wird am Ende der Datei hinzugefügt.
- `"w+"`: Öffnen zum Schreiben und Lesen. Wenn die Datei schon existiert, wird sie überschrieben. Der *stream* zeigt auf den Anfang der Datei.
- `"r+"`: Öffnen einer Datei zum Schreiben und Lesen. Der *stream* zeigt auf den Anfang der Datei.
- `"a+"`: Öffnen einer Datei zum Lesen und Hinzufügen. Zum Lesen zeigt der *stream* auf den Anfang der Datei. Geschriebenes wird immer hinzugefügt.

Eine Datei, die mit `fopen` geöffnet wurde, muss mit der Funktion `fclose` wieder geschlossen werden, die folgende Definition hat:

```
1 int fclose(FILE *stream);
```

Der Rückgabewert ist `0`, wenn die Datei erfolgreich geschlossen werden konnte und `EOF`, wenn nicht. Nur, wenn `fclose` aufgerufen wurde ist sichergestellt, dass die Daten auch in die Datei geschrieben wurden.

Formatierte Ein- und Ausgabe

Hat man einmal eine Datei geöffnet, muss man sich entscheiden, wie man sie ausliest oder in sie schreibt. Wir führen hier zunächst die sogenannte formatierte Ein- und Ausgabe ein. Dabei wird jedes gelesene Byte als ein ASCII Zeichen interpretiert. Wir haben diese Art von Ein- und Ausgabe bereits mit `printf` und `scanf` kennengelertn. Die Verallgemeinerung von `scanf` für beliebige *streams* ist `fscanf`

```
1 int fscanf(FILE *stream, char *format, ...);
```

`fscanf` bekommt als ersten Parameter den *stream* übergeben. `format` ist eine Zeichekette, wie wir sie schon von `scanf` kennen. Daher kennen wir auch schon, dass man Variablen über spezielle Zeichfolgen, die mit % beginnen, aus dem *stream* zuweisen kann. Dabei stehen unter anderem die folgenden speziellen Zeichen zur Verfügung

Platzhalter	Bedeutung
%d	int
%ld	long int
%ud	unsigned int
%f,%e	float
%lf,%le	double
%c	char
%s	eine Zeichenkette

Alle anderen Zeichen im Formatierungstext werden eingelesen, aber nicht gespeichert. Nehmen wir an, wir müssen die Zahlen aus folgender ASCII Datei einlesen:

```
1212
        1222
999             12212
888
```

Dies kann mit folgendem Code gemacht werden

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

int main(int argc, char *argv[])
{
  if (argc > 2)
    { // wurde ein Dateiname uebergeben?
      fprintf(stderr, "Usage: ./read_int_ascii filename\n");
      return (-3);
    }
  FILE *in = fopen(argv[1], "r"); // Oeffne die Datei
  if (in == NULL)
    { // Ueberpruefe auf Fehler
      fprintf(stderr, "Error opening the file\n");
      return (-2);
    }
  printf("Lese aus der Datei %s\n", argv[1]);
  int d;
  while (fscanf(in, "%d", &d) == 1)
    { // lese ein int nach dem anderen ein
      printf("%d\n", d);
    }
  if (fclose(in) != 0)
    { // Schliesse die Datei wieder
      printf("Error closing the File\n");
      return (-3);
    }
  return (0);
}
```

Es fällt auf, dass wir als Formatierungszeichenkette lediglich `%d` verwenden, und keine Leerzeichen. Der Grund dafür ist, dass Leerzeichen (und Zeilenumbrüche, Tabs etc.) als Trennzeichen interpretiert werden. Wie viele Leerzeichen und Zeilenumbrüche

zwischen den Zahlen eingefügt sind, spielt für die formatierte Eingabe keine Rolle. Die Ausgabe des Programms sieht dann wie folgt aus:

```
Lese aus der Datei numbers.txt
1212
1222
999
12212
88
```

Die formatierte Ausgabe funktioniert ähnlich, nur mit der Funktion `fprintf`

```
int fprintf(FILE *stream, char *format, ...);
```

Die Spezifizierungszeichen sind die gleichen wie für `fscanf`. Mit der Ausname von `%lf,%le`, die für `fprintf` nicht zur Verfügung stehen. `%e` gibt eine Fließkommazahl auf `double` Genauigkeit gerundet im wissenschaftlichen Format aus, `%f` ebenfalls auf `double` Genauigkeit gerundet als Dezimalzahl. Weiterhin kann man die Anzahl von Stellen etc beeinflussen, zum Beispiel

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

int main()
{
  FILE *out;
  out = fopen("quadrats.txt", "w"); // Oeffne die Datei zum Schreiben
  if (out == NULL)
    {
      fprintf(stderr, "Error opening the file\n");
      return (-1);
    }
  for (int i = 1; i <= 10; ++i)
    {
      fprintf(out, "%.6d\n", i * i); // Schreibe i^2 in die Datei
    }
  if (fclose(out) != 0)
    { // Schliese die Datei wieder
      printf("Error in closing the File\n");
      return (-2);
    }
  return (0);
}
```

Nach dem Aufruf enthält die Datei `quadrats.txt` folgendes

```
000001
000004
000009
000016
000025
000036
```

```
000049
000064
000081
000100
```

Die Spezifizierung `%.6d` bedeutet, dass die ganze Zahl mit sechs Stellen ausgegeben werden soll. Dementsprechend werden führende Nullen hinzugefügt.

Nichtformatierte Ein- und Ausgabe

Formatierte Ausgabe ist sinnvoll, wenn die Dateien für Menschen lesbar sein sollen und wenn nicht viele Daten gespeichert werden müssen. Für den Fall von vielen Daten sollte man allerdings auf nichtformatierte Ausgabe zurückgreifen (manchmal auch als binäre Ausgabe bezeichnet). Dies kann man in C beispielsweise mit den Funktionen `fread` und `fwrite` bewerkstelligen:

```
size_t fread(void *ptr, size_t size, size_t count, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)
    ;
```

Dabei werden `count*size` bytes direkt gelesen und im Speicherbereich abgelegt, auf den `ptr` zeigt, bzw. `count*size` aus dem Speicherbereich direkt geschrieben, ohne, dass dazwischen eine Konvertierung und Interpretation stattfinden würde. Beide Funktionen liefern die Anzahl der gelesen bzw. geschriebenen Elemente der Größe `size` zurück. Um den Erfolg der Operationen zu testen, sollte man also prüfen, ob der Rückgabewert dem Wert der Variablen `count` entspricht.

Um mehrfach aus einer Datei lesen zu können, kann man mit

```
void rewind(FILE *stream)
```

zum Anfang der Datei zurückkehren, oder mit

```
int fseek(FILE *stream, long int offset, int whence)
```

zu einem bestimmten Punkt in der Datei gehen. Bei letzterer Funktion wir der `offset` in Bytes relativ zu `whence` gesetzt. `whence` kann auf `SEEK_SET`, `SEEK_CUR` or `SEEK_END` gesetzt werden, was Anfang, momentane Position oder Ende in der Datei bedeutet. Im folgenden ein Beispiel:

Beispiel: Summe der Quadrate

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

int main()
{
  FILE *outin;
  int *squares;
  const int n = 10;
  // Oeffne die Datei
  outin = fopen("temporarystorage", "w+");
```

```
11    if (outin == NULL)
12      {
13        fprintf(stderr, "Error in opening the file\n");
14        return (-1);
15      }
16    // Speicher fuer squares reservieren
17    squares = (int *)malloc(sizeof(int) * n);
18    if (squares == NULL)
19      {
20        fprintf(stderr, "Error in allocating memory\n");
21        return (-2);
22      }
23    // Berechne Quadrate und speichere sie in squares
24    for (int i = 0; i < n; ++i)
25      {
26        squares[i] = (i + 1) * (i + 1);
27      }
28    // Schreibe squares in die Datei
29    if (fwrite((void *)squares, sizeof(int), n, outin) != n)
30      {
31        printf("Error in writing\n");
32        return (-3);
33      }
34    for (int i = 0; i < n; ++i)
35      {
36        squares[i] = 0;
37      }
38    rewind(outin); // Geh wieder zum Anfang der Datei
39    // Lese die Datei wieder ein
40    if (fread((void *)squares, sizeof(int), n, outin) != n)
41      {
42        printf("Error in read\n");
43        return (-4);
44      }
45    for (int i = 0; i < n; ++i)
46      {
47        printf("%d\n", squares[i]);
48      }
49    // Datei wieder schliessen
50    if (fclose(outin) != 0)
51      {
52        printf("Error in closing file\n");
53        return (-5);
54      }
55    return (0);
56  }
```

Die Datei wird zum Schreiben und Lesen geöffnet. Dabei wird eine eventuell existierende Datei gleichen Namens überschrieben. Danach werden die Quadrate von $i = 1, ..., n$ zuerst in die Datei geschrieben. Anschließend wird die Datei wieder von Anfang gelesen und die eingelesenen Zahlen auf die Standardausgabe ausgegeben.

2.3 Anwendung: Einfügesortieren

Als abschließendes und zusammenfassendes Beispiel für den ersten Teil dieses Skripts diskutieren wir nun noch ein etwas komplizierteres Beispiel. Dabei wenden wir das bisher gelernte auf das schon erwähnte Sortierproblem an: Ganze Zahlen $x_1, \ldots, x_n$ sollen eingelesen und dann sortiert ausgegeben werden. Dafür unterteilen wir das Problem zunächst in kleinere Unterprobleme, das Einlesen der Zahlen und das Sortieren. Dafür werden wir jeweils eine Funktion schreiben.

Wir beginnen mit dem Einlesen der zu sortierenden Zahlen. Dafür nutzen wir wieder die Funktion `scanf`. Die Funktion, die wir `read` nennen, sieht wie folgt aus

```
#include <stdio.h>

int read(int array[], const int n)
{
  int i;
  for (i = 0; i < n; i++)
    {
      if (scanf("%d\n", &array[i]) == EOF)
        {
          break;
        }
    }
  if (i != n) {
    printf("Sorry, es konnten nicht alle %d Zeichen eingelesen werden.\
   n", n);
  }
  return i;
}
```

Die Funktion liest bis zu n Zeichen ein, bzw. bis `scanf` den Rückgabewert `EOF` liefert. Dieser Rückgabewert, der in `stdio.h` definiert ist, bedeutet, dass `scanf` das Ende des Eingabestroms erreicht hat. Die eingelesenen Zahlen werde in `array` gespeichert, das by reference übergeben wird. Der Speicher für `array` muss also von der aufrufenden Funktion bereitgestellt sein. Der Rückgabewert unserer Funktion `read` ist die Anzahl der eingelesenen Zahlen.

Als nächstes schreiben wir die Funktion zum Sortieren:

```
void sort(int sortiert[], int unsortiert[], const int n)
{
  for (int i = 0; i < n; ++i)
    {
      sortiert[i] = unsortiert[i];
      for (int j = i; j > 0; --j)
        {
          if (sortiert[j] < sortiert[j - 1])
            {
              int swap = sortiert[j];
              sortiert[j] = sortiert[j - 1];
              sortiert[j - 1] = swap;
            }
          else
```

```
15              break;
16          }
17      }
18  }
```

Diese Funktion hat drei Eingabeparameter: Das Array, das die sortierten Zahlen enthalten soll, das Array, dass die zu sortierenden Zahlen enthält, und die Anzahl an zu sortierenden Zahlen. Wieder muss die aufrufende Funktion dafür Sorge tragen, dass für beide Arrays ausreichend Speicher bereitgestellt wurde. Überzeugen Sie sich bei den übrigen Zeilen, dass diese wirklich die n Zahlen durch Einfügen in das Array `sortiert` sortiert.

Bleibt noch die `main` Funktion, die wie folgt aussieht:

```
1  #include <stdio.h>
2
3  int read(int array[], const int n);
4  void sort(int sortiert[], int unsortiert[], const int n);
5
6  int main()
7  {
8    const int MAXNUM = 10000;
9    int array[MAXNUM], sortiert[MAXNUM];
10
11   // read the numbers
12   int actual_length = read(array, MAXNUM);
13
14   // sort the numbers
15   sort(sortiert, array, actual_length);
16
17   // print them on the screen
18   for (int i = 0; i < actual_length; ++i)
19     {
20       printf("%d ", sortiert[i]);
21     }
22   printf("\n");
23 }
```

Hier haben wir schon von der Möglichkeit Gebrauch gemacht, dass man Funktionen deklarieren kann, und die Definition später stattfinden kann. Man kann also beispielsweise in eine Datei `sort.c` erst den Code für die `main` Funktion kopieren, und im Anschluss daran den Code für die beiden Funktionen. Übersetzen kann man dann mit

```
gcc --std=c99 -o sort sort.c
```

Der Name der ausführbaren Datei ist dann `sort`. Diese kann im Prinzip mit

```
$> ./sort
```

von der Konsole ausgeführt werden. Allerdings brauchen wir erst noch Eingabedaten. Damit nicht alles von Hand eingegeben werden muss, nutzen wir die Funktionalität von Linux und leiten den Eingabestrom aus einer Datei um. Wir erzeugen erst eine Textdatei mit Namen `data.dat` mit folgenden Zeilen

```
12
77
25
43
4
```

Damit kann man das Programm von der Kommandozeile wie folgt ausführen:

```
$> ./sort < data.dat
4 12 25 43 77
```

Man kann, natürlich, wie im letzten Abschintt gezeigt, auch direkt aus `C` aus der Datei lesen.

2.3.1 Header und Quelltextdateien

Das gerade besprochene Beispiel eignet sich gut auf eine Möglichkeit hinzuweisen, die das Entwickeln von Programmen deutlich erleichtern kann. Es ist nämlich möglich, den Quelltext für ein Programm auf verschiedene Dateien aufzuteilen. Im obigen Beispiel könnte man beispielsweise die zwei Funktionen `sort` und `read` in eigenen Dateien speichern. Zweckdienlicherweise speichert man `read` in einer Datei `read.c`

```c
#include "read.h"
#include <stdio.h>

int read(int array[], const int n)
{
  // Funktionendefinition wie oben
  return i;
}
```

Listing 2.1: Datei `read.c`

und die Funktion `sort` in einer Datei `sort.c`

```c
#include "sort.h"

void sort(int sortiert[], int unsortiert[], const int n)
{
  // Funktionendefinition wie oben
  return;
}
```

Listing 2.2: Datei `sort.c`

Das neue Element in diesen beiden Dateien ist das Einbinden von zwei sogenannten *header* Dateien, `read.h` und `sort.h`. Das Einbinden erfolgt mit Hilfe von `include`, was im allgemeinen wie folgt aussieht

```c
#include "relative/path/header.h"
```

wobei der Pfad relativ zu dem Verzeichnis ist, in dem Übersetzt wird. Diese beiden header Dateien benötigt man, um die Funktionendeklarationen in der `main` Funktion bekannt zu machen. Und man bindet sie auch in den beiden Dateien `sort.c`

und `read.c`, um die Deklarationen konsistent zu halten. Die beiden header Dateien enthalten also lediglich die Funktionendeklarationen, also für die Funktion `read`

```
#pragma once

int read(int array[], const int n);
```

Listing 2.3: Datei `read.h`

und für die Funktion `sort`

```
#pragma once

void sort(int sortiert[], int unsortiert[], const int n);
```

Listing 2.4: Datei `sort.h`

In diesen beiden header Dateien ist das sogenannte Pragma `#pragma once` hinzugekommen. Es signalisiert dem Compiler, dass diese header Datei lediglich einmal einzubinden ist. Auf diese Weise verhindert man fehlerhaftes, wechselseitiges Einbinden von header Dateien, was zu unendlichen Schleifen führen kann. Die beiden header Dateien kann man nun nutzen, um die beiden Funktionen auch in der `main` Funktion bekannt zu machen. Die `main` Funktion können wir auch in eine eigene Datei schreiben, sagen wir `main.c`

```
#include<stdio.h>
#include "read.h"
#include "sort.h"

int main() {
  // Funktionendefinition wie oben
  return 0;
}
```

Listing 2.5: Datei `main.c`

Bleibt noch zu erklären, wie man das Programm übersetzt, wenn es in verschiedenen Dateien gespeichert ist. Dafür kennt der Compiler die Möglichkeit, eine Datei nur zu übersetzen und nicht zu verlinken. Zum Beispiel für die drei Dateien geht das wie folgt

```
>$  gcc -std=c99 -Wpedantic -c main.c -o main.o
>$  gcc -std=c99 -Wpedantic -c sort.c -o sort.o
>$  gcc -std=c99 -Wpedantic -c read.c -o read.o
```

Das flag `-c` zeigt dem Compiler an, dass nur übersetzt werden soll. Es wird eine sogenannte Objektdatei erzeugt, die typischerweise `.o` als Endung hat. Das heißt, dass die beiden Funktionen jetzt zwar deklariert, aber noch nicht definiert. Dementsprechend kann man eine Objektdatei auch nicht ausführen, man muss vorher noch linken. Dies geschieht mit

```
>$  gcc main.o sort.o read.o -o main.exe
```

In diesem Schritt wird dann die Deklaration in `main.o` mit den Definitionen in `read.o` und `sort.o` verknüpft. Ein weiterer Vorteil von mehreren Dateien ist, dass man immer nur den Teil neu übersetzen muss, den man geändert hat. Dies ist unter Umständen ein immenser Zeitvorteil, wenn die Projekte etwas anspruchsvoller werden. Man kann dies mit Hilfe des Programms `make` auch automatisieren.

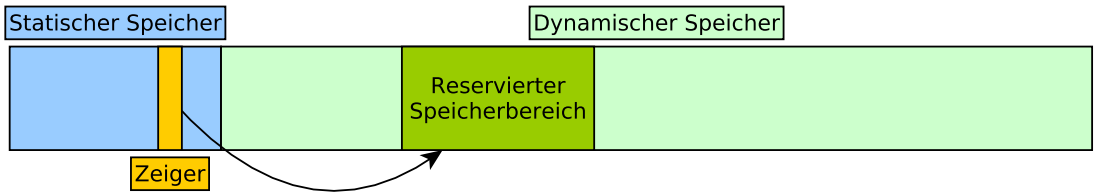


Abbildung 2.2: Dynamischer und statischer Speicher eines Programms.

2.4 Dynamische Speicherverwaltung

Mit den bisher eingeführten Grundlagen der Programmiersprache `C` kann man einfache Probleme lösen. Das haben wir am Beispiel des Einfügesortierens gesehen. Aber erst mit Hilfe dynamischer Speicherverwaltung und zusammengesetzter Datentypen kann man die volle Stärke von `C` ausnutzen.

Bei der Benutzung von Arrays muss man immer schon zur Zeit der Übersetzung wissen, wie groß das Array maximal werden darf. Das ist eine relativ starke Einschränkung, die sehr ineffizient sein kann. Entweder, man hat viel zu viel Speicher bereitgestellt, oder zu wenig. Genau passen wird es selten.

Der Speicher, der einem Programm zugeteilt ist, kann in zwei Kategorien unterteilt werden.

1. Statischer Speicher
   Dieses Teil haben wir schon kennengelernt. Im statischen Speicher befindet sich der Programmcode selbst, und der Speicher aller Variablen, die wir bisher deklariert haben. Seine Größe ist zum Zeitpunkt des Übersetzens festgelegt.

2. Dynamische Speicher
   Diesen Teil haben wir bisher noch nicht benutzt, aber jedes Programm kann eine bestimmte Menge an dynamischen Speicherplatz nutzen. Allerdings muss man sich zur Laufzeit des Programms selbst um das Reservieren und Freigeben dieses Speichers kümmern.

Wenn wir also in einem Programm zum Beispiel einen Zeiger

```
1 int *list;
```

deklarieren, so liegt dieser Zeiger selbst im statischen Speicherbereich. Er kann aber auf Speicher zeigen, der im dynamischen Speicherbereich liegt. Dies ist in Abbildung 2.2 illustriert.

2.4.1 Dynamische Speicherreservierung mit `malloc`

Um Speicher dynamisch zu reservieren, nutzt man Funktionen aus der Standardbibliothek `stdlib.h`. Das Reservieren erledigt die Funktion `malloc`, was für *memory allocation* steht. `malloc` hat die folgenden Syntax:

Definition `malloc`

```
1    void *malloc(size_t size);
2
```

Reserviert Speicherzellen von der Größe `size` bytes.

- Rückgabewert: Wenn die Reservierung erfolgreich war, einen Zeiger auf den Anfang des reservierten Speicherbereiches, andernfalls `NULL`.
- Eingabeparameter `size`: die Größe des zu reservierenden Speicherbereiches in bytes.
- Beispielcode:

```
1        int *array = (int *)malloc(sizeof(int) * 10);
2
```

Man beachte, dass der reservierte Speicherbereich mit der Funktion `free` (siehe unten) wieder freigegeben werden muss, wenn er nicht mehr benötigt wird. Ansonsten steht er für das Programm nicht mehr zur Verfügung. Am Ende des Programms wird aller vom Programm reservierte Speicher automatisch freigegeben.

Wir haben schon den Rückgabewert `void` kennengelernt, der für keinen Rückgabewert steht. Ein Zeiger auf `void` repräsentiert einen Zeiger ohne Typ. Er kann bzw. muss später mit einem *cast* in jeden anderen Zeigertyp umgewandelt werden. Deshalb wird im Beispielcode ein expliziter *cast* nach `int *` durchgeführt, den man immer bei der Benutzung von `malloc` machen sollte.

Den Zeiger `NULL` haben wir schon kennengelernt. `malloc` gibt `NULL` zurück, falls das Reservieren aus irgendeinem Grund nicht erfolgreich war. Daher ist es sehr wichtig bei jedem Aufruf von `malloc` den Rückgabewert auf `NULL` zu testen! Die Größe eines Typs in byte liefert die Funktion `sizeof`.

2.4.2 Speicherfreigabe mit `free`

Wie schon angedeutet, dynamisch reservierter Speicher muss wieder freigegeben werden, wenn er nicht mehr benötigt wird. Dafür gibt es die Funktion `free`:

Funktion Definition `free`

```
1    void free(void *memory);
2
```

Gibt einen mit `malloc` reservierten Speicherbereich wieder frei.

- Eingabeparameter: Zeiger auf einen Speicherbereich, der mit `malloc` reserviert wurde.

- Beispiel:

```
int *array = (int *)malloc(sizeof(int) * 10);
free(array);
```

Wendet man `free` auf einen Zeiger an, der nicht auf den Anfang eines mit `malloc` reservierten Bereiches zeigt, so erhält man einen Laufzeitfehler.

2.4.3 Mehrdimensionale Arrays

Bisher haben wir uns auf eindimensionale Arrays beschränkt. Als Beispiel für ein mehrdimensionales Array betrachten wir eine Drehmatrix in zwei Dimensionen um den Winkel α. Es handelt sich um eine lineare Abbildung, die einen zweidimensionalen Vektor $(x, y)^t$ auf den Vektor $(x', y')^t$ wie folgt abbildet:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.3}$$

Die entsprechende Matrix kann man in `C` mit Hilfe von zweidimensionalen Arrays darstellen. Die Deklaration eines statische 2×2 Arrays sieht wie folgt aus:

```
double rotate2d[2][2];
```

An Hand dieser Deklaration kann man mit dem jetzigen Vorwissen ahnen, dass ein zweidimensionales Array ein Array von Arrays ist. Daher entspricht ein solches zweidimensionales Array auch einem Zeiger auf einen Zeiger auf einen Typ. Diskutieren wir wieder ein Beispiel:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
  const int SPACEDIM = 2;
  const double M_PI = 3.1415;

  // Speicher reservieren
  double **array2d;
  if ((array2d = (double **)malloc(sizeof(double *) * SPACEDIM)) ==
   NULL)
    {
      printf("Fehler in malloc\n");
      return (-1);
    }
  for (int i = 0; i < SPACEDIM; ++i)
    {
      if ((array2d[i] = (double *)malloc(sizeof(double) * SPACEDIM)) ==
     NULL)
        {
```

```
20            printf("Fehler in malloc\n");
21            return (-2);
22          }
23      }
24    // Werte zuweisen
25    array2d[0][0] = cos(M_PI / 4.);
26    array2d[0][1] = sin(M_PI / 4.);
27    array2d[1][0] = -sin(M_PI / 4.);
28    array2d[1][1] = cos(M_PI / 4.);
29    // Ausgabe
30    for (int i = 0; i < SPACEDIM; ++i)
31      {
32        for (int j = 0; j < SPACEDIM; ++j)
33          {
34            printf("%e ", array2d[i][j]);
35          }
36        printf("\n");
37      }
38    // Speicher freigeben
39    for (int i = 0; i < SPACEDIM; ++i)
40      {
41        free(array2d[i]);
42      }
43    free(array2d);
44    return (0);
45  }
```

`array2d` ist jetzt als `double**` deklariert, also als Zeiger auf einen `double` Zeiger. Dann wird zunächst Speicher für ein Array von `double*` Zeigern reserviert. Das geschieht in Zeile 10. Man beachte, wie an dieser Stelle der Rückgabewert von `malloc` auf `NULL` getestet wird. Eine Zuweisung der Form `(x = y)` hat selbst den Wert von `x`. Anschließend wird in der Schleife, die in Zeile 14 anfängt, für jedes Element des Arrays von `double*` Zeigern selbst wieder Speicher reserviert. Das muss nun Speicher für `double` selbst sein. Wieder testen wir den Rückgabewert von `malloc` auf `NULL`.

Es ist wichtig, sich klar zu machen, dass `array2d` vom Typ `double**` ist. Dementsprechend zeigt es auf Elemente vom Typ `double*`, die dann selbst auf Elemente vom Typ `double` zeigen. Dies ist in Abbildung 2.3 dargestellt.

Es bietet sich an für die Speicherreservierung einer zweidimensionalen Matrix eine eigene Funktion zu schreiben. Es ist schließlich ziemlich wahrscheinlich, dass mehr als eine solche Matrix benötigt wird. Der entsprechende Quelltext könnte beispielsweise so aussehen:

```
1  #include <math.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  double **create2darray(const int dim)
6  {
7    double **p1 = NULL;
8    if ((p1 = (double **)malloc(sizeof(double *) * dim)) == NULL)
9      {
```

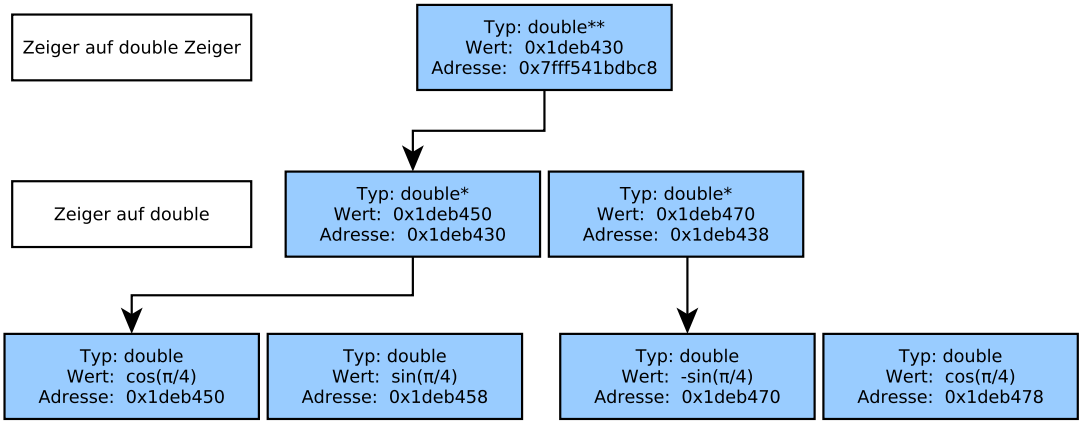


Abbildung 2.3: Illustration Zeiger auf Zeiger.

```
10        return (NULL);
11      }
12    for (int i = 0; i < dim; ++i)
13      {
14        if ((p1[i] = (double *)malloc(sizeof(double) * dim)) == NULL)
15          {
16            return (NULL);
17          }
18      }
19    return (p1);
20  }
21
22  int main()
23  {
24    const int SPACEDIM = 2;
25    const double M_PI = 3.1415;
26
27    double **array2d;
28    if ((array2d = create2darray(SPACEDIM)) == NULL)
29      {
30        printf("Fehler in malloc\n");
31        return (-1);
32      }
33    // Zuweisen
34    array2d[0][0] = cos(M_PI / 4.);
35    array2d[0][1] = sin(M_PI / 4.);
36    array2d[1][0] = -sin(M_PI / 4.);
37    array2d[1][1] = cos(M_PI / 4.);
38    for (int i = 0; i < SPACEDIM; ++i)
39      {
40        for (int j = 0; j < SPACEDIM; ++j)
41          {
42            printf("%e ", array2d[i][j]);
43          }
```

```
44        printf("\n");
45      }
46    // hier sollte noch der Speicher freigegeben werden!
47    return (0);
48  }
```

Die Funktion `create2darray` liefert als Rückgabewert eine Variable vom Typ `double**`, also genau das, was wir benötigen. In der Funktion selbst geschieht dann das, was wir im obigen Beispiel schon gezeigt hatten. Mit `return` wird dann das Objekt zurückgegeben, für das wir Speicher reserviert haben.

An dieser Stelle sieht man einen wichtigen Unterschied zwischen der Deklaration einer Variablen und dem Reservieren von Speicher mit `malloc`. Währen beispielsweise die Variable `p1` in der Funktion `create2darray` natürlich nur in der Funktion selbst sichtbar ist, bleibt der mit `malloc` reservierte Speicher auch außerhalb der Funktion reserviert. Und indem wir die Adresse zu dieser Speicherstelle von der Funktion an die aufrufende Funktion übergeben, können wir den reservierten Speicherbereich auch außerhalb der Funktion nutzen.

Das bedeutet aber auch, dass Speicher, der mit `malloc` reserviert wurde, und dessen Adresse wir nicht mehr kennen, nicht mehr nutzbar ist. Auf diese Art und Weise kann man recht einfach versehentlich Programme schreiben, die den gesamten Hauptspeicher eines Rechners reservieren, ohne ihn zu nutzen. Das führt relativ schnell zur Unbenutzbarkeit des entsprechenden Rechners. Ein Beispiel für einen solchen Fehler ist der folgende (nicht besonders sinnvolle) Code Abschnitt

```
int list[50];
doubke *p;
for (int i = 0; i < 50; i++)
  {
    p = malloc(sizeof(double));
    *p = i * i;
    list[i] = *p + (*p) * (*p);
  }
free(p); // falsche Stelle!
```

In jedem Durchlauf der Schleife wird Speicher für `p` reserviert, ohne ihn wieder frei zu geben. Nach Ende der Schleife ist keine der Adressen all dieser Reservierungen (bzw. nur noch die letzte) mehr verfügbar. D.h. man kann den Speicher weder weiter benutzen, noch kann man ihn nachträglich frei geben, ein Umstand der als Speicherleck oder *memory leak* bezeichnet wird.

2.5 Komplexe Datentypen

2.5.1 Eigene Datentypen mit `typedef`

Je nach Problemstellung ist es manchmal sehr hilfreich, Datenstrukturen selbst erzeugen zu können. `C` bietet dafür die Möglichkeit, und zwar auf verschiedene Weisen. Zunächst kann man in `C` einen neuen Typ definieren, z.B. einen eigenen Typ für reelle Zahlen:

```
typedef float real;
```

Dieser kann im Quelltext danach wie native `C` Typen verwendet werden

```
typedef float real;
int main()
{
  real x, y = 3.718;
  return 0;
}
```

Auf den ersten Blick mag eine eigene Definition für eine Datentyp für reelle Zahlen nicht besonders sinnvoll erscheinen. Allerdings kann man mit dieser Typdefinition zu einem späteren Zeitpunkt sehr einfach die verwendete Genauigkeit für reelle Zahlen ändern. Nämlich einfach, indem man die Zeile für die Typdefinition durch

```
typedef double real;
```

ersetzt.

2.5.2 Zusammengesetzte Datentypen

Eine weitere Art, einen neuen Datentyp zu definieren, ist zusammengesetzte Datentypen zu definieren. Ein einfaches mathematisches Beispiel sind Vektoren, die n Elemente haben. Mithilfe des Schlüsselwrts `struct` kann man in `C` zusammengesetzten Datentypen definieren, welche man sich wie Kontainer vorstellen kann. Die allgemeine Benutzung von `struct` ist wie folgt:

```
  struct name
  {
    Type1 name1;
    Type2 name2;
    Type3 name3;
    ...
  };
```

Die Verwendung als Typ für eine Variable:

```
  struct name variablename;
```

Als Beispiel betrachten wir einen Datentyp für kartesiche Koordinaten mit x, y und z Komponente. Der Datentyp wird wie folgt definiert:

```
struct coord
{
  double x, y, z;
};
```

mit drei `double` Elementen, jeweils für die x, y und z Komponente. Der Zugriff auf die einzelnen Elemente erfolgt über den Namen des `struct` und den Namen des Elements, getrennt durch einen Punkt. Für den allgemeinen Fall also wie folgt:

```
  struct name
  {
    Type1 name1;
    Type2 name2;
    Type3 name3;
    ...
  };
  name.name1 = value;
```

In unserem Beispiel für den Koordinatentyp sieht dies wie folgt aus

```
#include <stdio.h>

struct coord
{
  double x, y, z;
};
int main()
{
  struct coord r;
  r.x = 1.0;
  r.y = 3.4;
  r.z = -0.55;
  printf("r hat Koordinaten (%e, %e, %e)\n", r.x, r.y, r.z);
  return 0;
}
```

Also, allgemein gesprochen greift man auf die Elemente über

```
structname.elementname = wert;
```

zu. Elemente können selbst wieder ein `struct` sein. Einen Unterschied gibt es beim Zugriff auf Elemente, wenn man einen Zeiger auf einen `struct` benutzt. Dort kann man direkt mit dem Pfeil Operator `->` auf die Elemente zugreifen:

```
#include <stdio.h>

struct corrd
{
  double x, y, z;
};
int main()
{
  struct coord r;
  struct coord *v = &r;
  v->x = 1.0;          // aequivalent zu (*v).x = 1.0;
```

```
12   v->y = 3.4;           // aequivalent zu (*v).y = 3.4;
13   v->z = -0.55          // aequivalent zu (*v).z = -0.55;
14   printf("r hat Koordinaten (%e, %e, %e)\n", v->x, v->y, v->z);
15   return 0;
16 }
```

Elemente eines `struct` können natürlich auch Zeiger sein. Allerdings muss dann Speicher außerhalb der Deklaration des `struct` reserviert werden.

2.5.3 Beispiel: Datentyp für kartesische Koordinaten

Die Verwendung von `struct` ist sehr hilfreich, um logisch zusammengehörige Daten gesammelt zu übergeben und zu bearbeiten. Es muss dann nicht mehr jedes Element einzeln übergeben werden, sondern nur noch einmal der Kontainer. Der Zusammenhang der einzelnen Elemente bleibt damit erhalten. Man könnte nun beispielsweise eine Funktion für die Quadratnorm einer Koordinate wie folgt schreiben:

```
1  #include <stdio.h>
2
3  struct coord
4  {
5    double x, y, z;
6  };
7  double sqnorm(struct coord r) {
8    return(r.x*r.x + r.y*r.y + r.z*r.z);
9  }
10
11 int main()
12 {
13   struct coord r;
14   r.x = 1.0;
15   r.y = 3.4;
16   r.z = -0.55;
17   printf("r hat Quadratnorm %e\n'', sqnorm(r));
18   return 0;
19 }
```

Mit einem `struct` wird ebenfalls einer neuer Typ bereitgestellt. Der entsprechende Name enthält aber immer das Schlüsselwort `struct`, in unserem Fall `struct coord`. Man kann einen `struct` mit einem `typedef` kombinieren, um nicht immer `struct` schreiben zu müssen. Im Quelltext sähe das so aus:

```
1  struct coord
2  {
3    double x, y, z;
4  };
5  typedef struct coord coord;
```

Das sieht etwas verwirrend aus, weil `coord` zweimal auftaucht. Aber der Ausdruck wird klar, wenn man sich erinnert, dass `struct coord` im Prinzip ein Typ ist. Und damit ist der Name `coord` noch nicht vergeben. Die etwas gebräuchlichere Art obiger

Typdefinition geht über die Möglichekeit eines anonymen `struct`. Man kann nämlich auch schreiben:

```
typedef struct
{
  double x, y, z;
} coord;
```

Damit ist natürlich `struct coord` nicht definiert, aber man kann `coord` wie einen `C` Typ verwenden.

Index

&, 43
++, 17
EOF, 53
FILE, 53
I, 39
NULL, 45
SEEK_CUR, 57
SEEK_END, 57
SEEK_SET, 57
[ ], 31
#pragma once, 62
argc, 49
argv, 49
break, 22, 24, 25
cexp, 39, 40
cimag, 40
clang, 11
complex.h, 39, 40
complex, 39
conj, 40
const, 16, 47
creal, 40
default, 22
do-while, 25
fclose, 54
fopen, 54
for, 23
fread, 57
free, 65
fscanf, 54
fseek, 57
fwrite, 57
gcc, 11
getchar, 53
if-else, 21
main, 49
    Parameter, 49
malloc, 64
math.h, 40
prinft, 54
printf, 12, 17, 22, 53
putchar, 53
rewind, 57
scanf, 22, 53, 54
snprintf, 35
sprintf, 35
stderr, 53
stdin, 53
stdio.h, 12, 53
stdout, 53
struct, 70
switch, 21, 22
typedef, 70
void, 29, 65
while, 25

Adressoperator &, 43
Algorithmus, 7
    Einfügesortieren, 7
array, 30, 46
ASCII Format, 8

bit, 8
byte, 8

C99 Standard, 6
call by reference, 44–46
call by value, 44
cast, 14
    explizit, 14
    implicit, 14
    implizit, 53

Datentypen, 14
    complex, 39
    double complex, 39

`ind`, 13

Feld, 30, 46
  eindimensional, 31
  mehrdimensional, 32, 66
Fließkommazahl, 9
Funktion, 26
  Aufruf, 27
  Definition, 26
  Prototyp, 26

Header Dateien, 61

Indexoperator `[ ]`, 31

Linken, 62

make, 63
Mantisse, 9

Objektdatei, 62
Operator, 16
  arithmentisch, 17
  Infix, 16
  logisch, 17
  Postfix, 16
  Präfix, 16
  Vergleich, 18

Schleife
  `do-while`, 25
  `for`, 23
  `while`, 25
Schlüsselwörter, 15
segmentation fault, 45
Speicher
  dynamisch, 64
  statisch, 64
string, 33

Variable
  Definition, 13
  Deklaration, 12, 13
  Initialisierung, 13

word, 8

Zeichenkette, 33
Zeiger, 43
  `*`, 50
  `++`, 50
  `NULL`, 65
  `void`, 65
Zeiger auf `const`, 52