Begleitskript: C-Kurs 2020

Bartosz Kostrzewa

2. April 2020

Inhaltsverzeichnis

Allgemeine Informationen

0.1 Administrativa

- es findet *keine* interaktive Vorlesung statt, Streams wird es vorerst auch nicht geben
- Skript, Vorlesungsfolien, Vorlesugnsbegleitskript, sowie Übungen sollen in Heimarbeit bearbeitet werden.
- Bartosz Kostrzewa, bartosz_kostrzewa@fastmail.com, Raum 3.009 HISKP (bis auf unbestimmte Zeit im Homeoffice)
- Tutoren: Marcel Hohn, Marcel Nitsch, Simon Schlepphorst, Florian Taubert

Ein Zoom Meeting, erreichbar unter
`https://zoom.us/j/121361434?pwd=Q3NtYlFjSDI4QmtLY3k3bG14c3plQT09`
, ist eingerichtet worden. Das Passwort lautet **311077**. Dem Meeting kann man jederzeit beitreten und sich im Chat austauschen. Audio und Video sind natürlich auch verfügbar (wenn per Zoom-client verbunden). Dazu muss lediglich der Zoom Client heruntergeladen werden → Meeting ID ist **121-361-434**, wenn man den oberen Link nicht nutzt.

Das Meeting ist (ohne Account) auch über die Telefonnummern (+49 69 7104 9922, +49 30 5679 5800, +49 695 050 2596) zu erreichen.
Wie genau das mit dem Meeting funktionieren wird, klärt sich erst nach ein paar Tagen. Der Plan ist vorerst:

- 23.03 - 03.04: 10-12 Uhr, Dozent ist verfügbar, um Fragen zum Kurs, zum Skript und zu den Vorlesungen zu beantworten.
- 23.03 - 03.04: 14-16 Uhr, die Tutoren sind verfügbar, um Fragen zu den Übungen zu beantworten.

Da wir potentiell ca. 100 Studierende im Kurs haben, müssen wir schauen, ob das in dieser Form nicht zu chaotisch wird. Unter Umständen wird es eine Aufteilung in zwei oder vier Gruppen geben, für die dann jeweils zwei oder ein Tutor(en) verantwortlich sind/ist.
Zusätzlich existiert ein Diskussionsforum.

0.2 Umfrage

Unter
`https://ecampus.uni-bonn.de/goto_ecampus_svy_1664941.html`
besteht eine (anonyme) Umfrage zur Erhebung der vorhandenen Programmierkenntnisse. Ich würde mich über rege Teilnahme sehr freuen!

0.3 Lernziele

Ziele dieser Vorlesung sind, unter Anderem:

- Verständnis der immer weiter steigenden Signifikanz der Programmierung in der Physik
- Algorithmen entwickeln und verstehen
- Algorithmen in Quelltext übertragen
- Erstes Kennenlernen der sogennanten *imperativen* Programmierung
- Kennenlernen der Daten- und Kontrollstrukturen von C99
- Praktischer Einsatz des C-Compilers zur Übersetzung des Quelltextes in ausführbaren Maschinencode
- Erstellen eigener C-Programme in den Übungen
  - einfache Beispielprogramme bis zu komplizierteren Programmen aus merheren Quelltextdateien
  - Verwendung externer Bilbiotheken
- Vorbereitung auf *physik441: Computerphysik* (SoSe 2020), *physics760: Computational Physics* (WiSe 2020/2021), etwaige Bachelor- und Masterarbeiten

1 Vorlesung 1

1.1 Motivation

Programmierung ist in den letzten 40 Jahren eine essenzielle Fähigkeit für Physiker geworden. Sowohl in der Theorie, als auch im Experiment sind gut entwickelte Programmierkenntnisse nicht mehr wegzudenken. PhysikerInnen treiben die Verarbeitung von riesigen Datenmengen voran, sei dies aus Experimenten wie dem LHC oder aus großangelegten Computersimulationen, z.B. in der Gitter-QCD, Astronomie, theoretischer Chemi, Biophysik oder Kosmologie. Auch die Entwicklung neuer Experimente stellt erhebliche Anforderungen an den Programmierer. Ein Beispiel sind spezielle Filter, sogenante *Trigger*, welche am LHC in Nanosekunden darüber entscheiden ob ein *Event* interessant ist, oder ob es verworfen werden soll. Würde dies nicht gemacht, wäre es unmöglich die vom LHC produzierten Datenmengen zu speichern und auszuwerten. Um die Software für diese Trigger zu schreiben, wird einerseits ein genaues Verständnis der Physik vorausgesetzt, andererseits aber auch die Beherrschung verschiedenster Konzepte des Hochgeschwindigkeitsrechnens und der Informatik.
Von PhysikerInnen werden einige der ambitioniertesten Simulationen auf Großrechnern durchgeführt. In der Gitter-QCD, z.B., schreibt man Software zur Simulation der starken Wechselwirkung, welche auf tausenden bis hundertausenden von Rechnereinheiten parallel läuft. Andere Theoretiker, die sich mit der Quantenfeldtheorie befassen, erstellen komplexe Bibliotheken zur Auswertung hochkomplizierter Integrale und tragen aktiv zu Entwicklungen in der angewandten Mathematik bei.
Phänomenologen, Experimentatoren, aber auch Theoretiker benötigen zusätzlich zu den oben ganennten, spezialisierten Softwarepaketen, Programme zur Daten- und Projektverwaltung und viele weitere Softwarepakete. Obwohl es für einige dieser Aufgaben kommerzielle Angebote gibt, wird in sehr vielen Fällen spezielle, dem Fachbereich angepasste Software entwickelt, um diese Probleme zu lösen. Auch hier sind es oft Wissenschaftler, die solche Programme schreiben und dann pflegen.
Wir haben auf den Folien 4 bis 9 Beispiele für Simulationen verschiedener Art.
Der Wellenstein 7-x Stellerator ist ein Fusionsexperiment, bei dem eine komplexe Magnetgeometrie dazu genutzt wird, ein Plasma im Ring des Stellerators festzuhalten, zu komprimieren und zu erhitzen. Diese Geometrie konnte nur anhand von Simulationen der Plasmaphysik erstellt werden und PhysikerInnen am Max Planck Institut für Plasmaphysik haben mitgeholfen, diese Simulationssoftware zu entwickeln. Da diese Simulationen rechnerisch sehr teuer sind, bedarf es einer schnellsmöglichen Ausführungsgeschwindigkeit.
Um Galaxien besser verstehen zu können und insbesondere ihre Entstehung zu erklären, arbeiten weltweit WissenschalfterInnen an detaillierten Langzeitsimulationen, bei denen allgemeine Relativitätstheorie, Elektrodynamik und Plasmaphysik kombiniert werden, z.B. um realistische Modelle der Entstehung der Milchstraße zu erschaffen. Auch hierfür werden reisige Großrechner genutzt, die viele Millionen Euro in Einkauf und Betrieb kosten, weswegen höchte Performance von großer Bedeutung ist.
Am LHC kollidieren Protonen bis zu 600 Millionen mal pro Sekunde. Die daraus resultierenden Daten der Teilchenregen können nicht alle gespeicher und analysiert werden. Deshalb werden sogenannte *Trigger* gebaut, eingebettete Hardware- und Softwarelösungen zur Vorauswahl *interessanter* Ereignisse. Diese Systeme entscheiden in Bruchteilen einer Millisekunde, ob ein Datensatz gespeichert oder verworfen werden soll und werden von hochspezialisierten ExperimantatorInnen und IngenieurInnen entwickelt.
In meinem eiegenen Betätigungsfeld, der sogenannten *Gitter-Quantenchromodynamik* wird die Dynamik der starken Wechselwirkung, welche sich bei *niedrigen* Energien nicht analytisch beschreiben lässt, *ab initio* auf dem Rechner simuliert. Zum Einsatz kom-

men dabei Simulationsprogramme mit mehreren hunderttausend oder gar millionen Programmzeilen, sowie die schnellsten Großrechner der Erde. Die Entwicklung dieser Software setzt wieder voraus, dass man einerseits ein vollständiges Verständnis der Physik, anderseits aber auch hochentwickelte Fähigkeiten in der Programmierung und im Hochgeschwindigkeitsrechnen hat, damit die Programme so schonend wie möglich mit den begrenzten Rechenresourcen umgehen.
Die Kimaforschung ist ein weiterer Bereich, in dem multidisziplinäres Wissen in Simulationssoftware eingeht um auf Großrechnern Klimamodelle zu simulieren und daraus Erkenntnisse z.B. über die Erderwärmung zu gewinnen. Aufgrund der prekären Lage unseres Klimas bedarf es keiner weiteren Erklärung ob der Wichtigkeit dieser Berechnungen. Es sei nur noch gesagt, dass hier auch physikalisches und technisches Fachwissen aufeinandertreffen, um diese Simulationen überhaupt möglich zu machen.
Schlussendlich wenden wir uns der Biophysik zu. Auch hier werden mithilfe von sogenannten *Molekulardynamiksimulationen* Erkenntnisse darüber gewonnen, wie die vielen komplizierten biochemischen Prozesse ablaufen. Biologie, Chemie, Physik und Informatik kommen zusammen, um, z.B., nach Stoffen zu suchen, die gegen das SARS-CoV-2 Virus effektiv sein könnten.
All diese Beispiele haben gemein, dass die erstellten Programme eine höchstmögliche Ausführungsgeschwindigkeit benötigen und diese ist in der Regel nur mit kompilierten Programmiersprachen wie C, C++ oder Fortran zu erreichen.

1.2 Die Programmiersprache C

All diese Aufgaben haben gemein, dass man als ProgrammiererIn eine Problemstellung in eine, dem Rechner verständliche, Sprache bringen muss. Diesem Prozess unterliegt die Formulierung von sogenannten Algorithmen. Dies sind präzise Vorschriften, welche in endlich vielen Schritten für eine gewisse Eingabe, eine gewisse Ausgabe liefern sollen. Nachdem man eine Problemstellung so aufgeteilt hat, dass man sie anhand von Algorithmen darstellen kann, beginnt man damit, diese in eine Programmiersprache zu übertragen und die Konstrukte der Programmiersprache dazu zu nutzen, Daten- und Programmfluß zu steuern. Die Auswahl an verfügbaren Programmiersprachen ist groß und diese unterscheiden sich z.B. darin, welche Kontrollstrukturen und Programmkonstrukte die Sprache zur Verfügung stellt, wann geschriebene Befehle ausgeführt werden, wie Datentypen und Variablen identifiziert werden oder wie genau man die Rechnerarchitektur verstehen muss, um die Sprache zu nutzen. Im Allgemeinen nutzt man als Programmiere immer mehrere Programmiersprachen.
In dieser Vorlesung arbeiten wir ausschließlich mit C und um genauer zu sein, mit C99. Dies hat den großen Vorteil, dass es in der Wissenschaft sehr viel Software gibt, die entweder in C geschrieben ist oder auf Komponenten aufbaut, die wiederum in C geschrieben sind. Desweiteren gibt es viele Sprachen, welche C syntaktisch gleichen und ein Verständnis von C erlaubt es, diese anderen Sprachen schnell zu erlernen. Schlussendlich sind Programme die in C geschrieben sind oft, relativ gesehen, schnell. Dies bedeutet nicht, dass man nicht auch langsame Software in C schreiben kann.
Die Verwendung von C hat jedoch auch einige Nachteile.

- Es handelt sich um eine relativ alte Programmiersprache, die viele moderne Konstrukte nicht unterstützt. In anderen Programmiersprachen erlauben es diese Konstrukte, komplizierte Software mit weniger Aufwand zu entwickeln. Desweiteren werden viele Details, die man in C berücksichtigen muss, automatisch vom Compiler erledigt.

- C zwingt den Programmierer oft, genau darüber nachzudenken, was der Rechner eigentlich im Hintergrund macht. Dies passiert z.B., bei der Speicherverwaltung.

1.3 Algorithmen: Pseudocode

Bevor man damit beginnt, ein Programm zur Lösung eines bestimmten Problems zu implementieren, lohnt es sich, zunächst dieses Programm in einer verständlichen Sprache aufzuschreiben. Wir nehmen das sogenannte *Einfügensortieren* als Beispiel. Nachdem wir uns überlegt haben, was das Programm überhaupt machen soll, übertragen wir es zunächst in sogenannten *Pseudocode*. Hierbei handelt es sich um eine abstrakte Syntax, die einem Programm gleicht, aber durch die Verwendung von mathematischen Symbolen kompakter ist. Es fehlen auch Eigenheiten der einen oder anderen Programmiersprache, sodass man sich vollständig auf den Algorithmus konzentrieren kann. Nachträglich kann ein solcher Pseudocode dann recht einfach in ein fertiges Programm in einer beliebigen Programmiersprache übersetzt werden.
Gegeben sei eine unsortierte Liste U mit Elementen x_i, $i \in \{0, \ldots, n-1\}$. Ziel ist es, für diese Liste eine Permutation der Indizierung, $\sigma(i)$ zu finden, welche die Elemente der Liste nach dem Kriterium kleiner-gleich (oder größer-gleich) sortiert, mit dem Ergebnis, dass

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

Ein möglicher Ansatz:

1. Man beginne mit zwei Listen, der unsortierten, U, und einer zweiten Liste, S, welche zu Beginn leer ist
2. Man verschiebe das jeweils erste Element der Liste U in die Liste S und füge es dabei so ein, dass S immer sortiert ist.
3. **2.** wird so lange wiederholt, bis U leer ist.

In der Praxis wird dieser Algorithmus entweder destruktiv implementiert, wobei die Anfangslemente der Liste U nach und nach einfach vertauscht werden, oder so, wie oben beschrieben, aber ohne die Elemente aus der Liste U zu löschen. Bei der Übertragung eines Problems in Quelltext, auch *code* genannt, ist der Schritt über sogenannten Pseudocode, insbesondere für kompliziertere Problemstellungen, sehr hilfreich. Ein Pseudocode für diesen Algorithmus könnte so aussehen:

Algorithmus 1 Einfügensortieren

Input: Lists U, S
Output: List S

```
1:  for i = 0 to length(U)-1 do
2:      S_i ← U_i
3:      j ← i
4:      while j > 0 do
5:          if  S_j < S_{j-1} then
6:              t ← S_j
7:              S_j ← S_{j-1}
8:              S_{j-1} ← t
9:              j ← j − 1
10:         else
11:             break
12:         end if
13:     end while
14: end for
```

Die fettgedruckten Teile des Pseudocodes entsprechen Kontrollstrukturen, welche den Programmfluss steuern und nachher in die ensprechenen Kontrollstrukturen der Programmiersprache abgebildet werden. Variablen verschiedener Datentypen und daraus

zusammengesetzte Datenstrukturen dienen dann als Speicher für die verarbeiteten Eingabedaten und die generierten Ausgabedaten und die Zuweisung von Werten zu Variablen wird durch ← dargestellt. Schlussendlich gibt es natürlich noch arithmetische und logische Operationen welche anhand von Operatoren dargestellt werden. In den Übungen werden Sie selber Pseudocode verfassen und diesen dann auch in Programme übersetzen. An dieser Stelle sei noch gesagt, dass es viele verschiedene Konventionen für Pseudocode gibt.
Wir werden die Datentypen, Operatoren und Kontrollstrukturen gleich kennenlernen, zuerst aber, wollen wir ein einfaches C-Programm erstellen.

1.4 Die Struktur eines C-Programms

Wir sehen auf Folie 13 als Beispiel ein sehr einfaches Programm, welches den Text "`Hallo, Welt!`" auf der Konsole ausgibt. Dazu müssen wir eine sogenannte *Headerdatei* einbinden, welche die Funktionsdefinitionen der Eingabe- und Ausgabebibliothek enthält → `#include <stdio.h>`.
Jedes Programm hat eine sogennante *main* Funktion, welche einen ganzzahligen Rückgabewert hat und selbst Argumente erhalten kann. In diesem Fall übergeben wir keine Argumente (`void`) und haben den Rückgabewert `0`. Die Funktion `printf` dient zu formatierten Ausgabe, hier geben wir bloß den gewollten Text, gefolgt von einem Zeilenumbruch aus.

1.4.1 Quelltext und Kompilieren

C-Programme bestehen aus Quelltextdateien welche vom *Compiler* in Maschinencode übersetzt (auch *kompiliert*) werden müssen. Diese Maschinencodeschnipsel werden dann vom *Linker* (meist Teil des Compilers) zu ausführbaren Programmen *verlinkt*. Man unterscheidet hier zwischen *Header*-Dateien, die Funktionen und Datentypen beschreiben und *Quelltextdateien*, in welchen die Funktionen *implementiert* sind. Diese Trennung erlaubt es, C-Programme modular zu gestalten.

1.5 Variablen deklarieren und definieren

Um in C eine Variable zu *deklarieren* (dem Compiler bekannt zu machen), muss für die Variable ein Name und ein Datentyp gewählt werden. C stellt eine Reihe elementarer Datentypen zur Verfügung. Diese unterscheiden sich z.B. darin, dass sie als Speicher für ganz-zahlige oder reelle Zahlen dienen.
In C werden Programmblöcke mit geschweiften Klammern abgegrenzt: ein Programmblock beginnt mit `{` und endet mit `}`. Wie man im Beispiel sehen kann, können wir eine Variable gleichen Namens sowohl ausserhalb, als auch innerhalb eines Block definieren. Es handelt sich hierbei um zwei verschiedene Variablen, wobei die im inneren Block definierte, jene aus dem darüberliegenden Block überschattet. Man spricht auch von *masking*: nur die innere Variable ist sichtbar, wenn diese den gleichen Namen hat, wie eine Variable aus einem übergeordneten Block.

1.5.1 Einrückung und Kommentare

Die Lesbarkeit des Quelltextes ist ein wichtiger Aspekt der Programmierung. In C werden Programmteile durch geschweifte Klammern in Blöcke aufgeteilt. Die Hierarchie dieser Blöcke und Unterblöcke solltte
Ein weiterer wichtiger Aspekt ist die Dokumentation des Quelltextes. Besonders wenn nicht-triviale Strukturen auftreten, oder man arbiträre Entscheidungen trifft (z.B. eine

Konstante auf einen bestimmten Wert setzt), sollte man dies beschreiben oder Begründen.
Einzeilige Kommentare werden in C99 mit `//` eingeführt und mehrzeilige Kommentare kann man mit `/* [...] */` in den Quelltext schreiben.

1.5.2 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	65535
	unsigned	4	0	$+4,294,967,295$
	long	4	$-2,147,483,648$	$2,147,483,647$
	long long	8	$-9,223,372,036,854,775,807$	$9,223,372,036,854,775,807$
char	signed	1	-128	127
	unsigned	1	0	255
float		4		
double		8		
long double		8		

Der bei der Deklaration gewählte Datentyp ist für die Dauer der Existenz der Variable festgelegt und kann nicht geändert werden. In C können jedoch Variablen verschiedener Typen einander zugewiesen werden. Hierfür wird, wenn man eine Variable eines größerwertigen Typs einer kleinerwertigen Variable zuweist, erstere gekürzt, was zu Problemen führen kann. In die umgekehrte Richtung funktioniert die Konversion meist richtig, es sollte aber klar sein, dass man einem vorzeichenlosen Datentyp keinen negative Wert zuweisen kann.

1.6 Operatoren

Mathematische und logische Operationen werden in C mithilfe von Operatoren dargestellt. Man unterscheidet zwischen *unären*, *binären* und *ternären* Operatoren, welche jeweils ein, zwei oder drei Argumente haben. Desweiteren unterscheidet man zwischen *infix*, *präfix* und *postfix* Operatoren, welche respektiv zwischen, vor und nach Ausdrücken stehen.

1.6.1 Arithmetische Operatoren

Operator	Ausdruck	Auswertung
Zuweisung	`a = b`	Werte 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`	Werte von `a+b`
Zuweisung und Subtraktion	`a -= b`	Werte von `a-b`
Zuweisung und Multiplikation	`a *= b`	Werte von `a*b`
Zuweisung und Division	`a /= b`	Werte 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`

1.6.2 Vergleichsoperatoren

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`

1.6.3 Logische Operatoren

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

1.6.4 Priorität von Operatoren

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

1.7 Kontrollstrukturen

Keine Kommentare.

1.7.1 Bedingte Ausführung: `if` / `else` statement

Keine Kommentare.

1.7.2 Schleifen: `while` loops

Keine Kommentare.

1.7.3 Schleifen: `for` loops

Keine Kommentare.

1.8 Maschinenzahlen

Nur eine Teilmenge, $\mathcal{M}$, der reellen Zahlen ist auf dem Rechner darstellbar. Der IEEE-Standard definiert folgendes Format:

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

wobei $E \in \mathbb{N}, N > 1$ die Basis, $k \in \mathbb{N}$ 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:

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

wobei k die Mantissenlänge darstellt. Auf modernen Rechnern ist fast immer $a_i \in 0, 1$ (Binärsystem).
Bei der Abbildung der reellen Zahlen auf die Menge der Maschinenzahlen muss fast immer eine Rundungsoperation vorgenomme werden. Dabei geht Information verloren, eine Rückabbildung ist nicht eindeutig möglich.

1. Die Abbildung der Zahl $0,1$ im Dezimalsystem auf das Dualsystem $0,1_{10} = 0,000110011001100\ldots_2$ 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 eines Verfahrens führt.

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

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

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

$$-\delta_M \leq \delta x \leq \delta_M \,. \tag{2}$$

2 Vorlesung 2

2.1 Mini-Intro: Textausgabe

Wir haben in der ersten Vorlesung und den Beispielen schon die Funktion `printf` genutzt. Diese dient zur formatierten Textausgabe und hat in ihrer Funktionssignatur als erstes Argument den sogenannten *Formatstring* und nimmt beliebig viele weitere Argumente an. Beim Formatstring handelt es sich um einen *Zeiger* (Vorlesung 3) auf ein Array von Zeichen (`char`). Der Formatstring enthält Text und Platzhalterzeichen, wobei letztere durch die Werte der Variablen ersetzt werden, welche über die variable Argumentenliste (`...`) übergeben werden. Die Platzhalter im Formatstring sollten von rechts nach links den Variablen entsprechen, die man ausgeben möchte.
Es gibt für verschiedene Datentypen unterschiedliche Platzhalter. Hierbeit ist zu beachten, dass der Compiler (je nach Architektur) zwar Warnungen ausgibt, wenn falsche Platzhalter verwedet werden, ansonsten aber einfach annimmt, dass die angegebene Variable den richtigen Datentypen hat. Wenn man also `%d` nutzt, aber ein `char` als enstprechendes Argument übergibt, wird `printf` versuchen, aus der Speicherstelle dieser Variablen anstelle eines Bytes, *vier* Bytes auszulesen. Dabei kann es sein, dass `printf` einen unerlaubten Speicherzugriff durchführt und das Programm abstürzt.

2.2 Mini-Intro: Textausgabe **flags** und Feldbreiten

Die formatierte Ausgabe kann mithilfe von *flags* und Feldbreiten genauer gesteuert werden, um, z.B., Text links- oder rechtsbündig auszugeben, aber auch, um den Text mit einer Mindestbreite auszugeben. Auch die Anzahl auszugebender Nachkommastellen bei Fließkommazahlen wird mittels eines solchen flags bestimmt.

2.3 Mini-Intro: Texteingabe

Ich hatte ganz am Anfang schon darauf hingewiesen, dass C einem direkten Zugriff zum Speicher eines Programms gibt. Wir merden in der nächsten Vorlesung noch genauer auf *Zeiger* zu sprechen kommen, schauen uns aber zunächst den Adressoperator `&` an. Auf den Wert einer Variablen wird mit dem Variablennamen direkt zugegriffen, z.B. `dezimalzahl`. Will man jedoch wissen, an welcher Stelle im Speicher der Compiler diese Variable abgelegt hat, nutzt man den Adressoperator. Der Ausdruck `&dezimalzahl` gibt die Speicheradresse der Variablen `dezimalzahl` zurück.
Mit der Funktion `scanf` wird formatierter Text eingelesen (kann als umgekehrtes `printf` verstanden werden). `scanf` erhält als erstes Argument auch einen Formatstring und eine beliebige Zahl weiterer Argumente. Anders als bei `printf`, werden hier die Adressen von Variablen angegeben und *nicht* die Namen der Variablen. Für jeden Platzhalter, schreibt `scanf` direkt in den Speicher, dessen Adresse übergeben wurde. Die Anzahl Bytes, die `scanf` schreibt, hängt vom Platzhaltertypen ab. Das bedeutet auch, dass wenn Platzhaltertyp und Variablentyp nicht übereinstimmen, der Compiler unerlaubt in Speicher schreiben wird, in den er eigentlich nicht schreiben sollten. Bestenfalls stürzt das Programm ab, meistens wird aber einfach in den Speicher geschrieben und man merkt davon erst etwas, wenn subtile und schwer erklärbare Fehler auftreten.
Im dazugehörigen Beispiel wird noch etwas genauer auf `scanf` eingegangen, insbesondere auf den Rückgabewert. Hierbei handelt es sich um ein `int`, also eine Ganzzahl, die der Anzahl gelesener Felder entspricht. Man sollte diesen Rückgabewert immer überprüfen, um sicherzustellen, dass auch alle Felder gelesen wurden, die man lesen wollte. Das Auslesen der Eingabe schlägt dabei schon beim ersten unerwarteten Zeichen fehl. Wenn man bei `02_02_test_scanf` z.B. als erste Eingabe ein `a` übergibt, wird auch eine darauf folgende Zahl nicht gelesen werden.

2.4 Funktionen

Bei der Programmiersprache C handelt es sich um eine *prozedurale* Programmiersprache. Das beudetet, dass C Programme im Wesentlichen aus Funktionsdefinitionen und Funktionsaufrufen bestehen. Die auf Folie 31 gezeigte Baumstruktur stammt aus der Gitter-QCD Simulationssoftware `tmLQCD` und zeigt wie die `main` Funktion andere Funktionen aufruft, die wiederum weiter Funktionen aufrufen.

2.5 Funktionskopf und Signatur

Man unterscheidet, ebenso wie bei Variablen, zwischen der *Deklaration* und der *Definition* einer Funktion. Die Deklaration erklärt dem Compiler, welchen Namen und Rückgabewert, aber auch wie viele und welche Argumente eie Funktion hat. Eine Funktion kann zwar mithilfe der variablen Argumentenliste `...` beliebig viele Argumente beliebiger Datentypen annehmen, es existiert jedoch in C keine *Überladung* (Unterscheidung von Funktionen gleichen Namens aber mit unterschiedlichen Argumentenlisten). Eine Funktion muss also einen eindeutigen Namen haben.

2.6 Funktionsaufruf und Rückgaberwert

Keine Kommentare

2.7 Definition

Die Funktionsdeklaration sagt dem Compiler lediglich, wie auf die Funktion zugegriffen wird, nicht aber, was die Funktion eigentlich macht. Die Deklarationen befinden sich meist in den *Headerdateien* (`*.h`).
In der Funktionsdefinition, die in einem modularen Programm meist in eigenen Quelltextdateien (`*.c`) gehalten werden, wird die Funktion, wie der Name schon sagt, definiert. Wir werden die modulare Programmierung später noch im Detail kennenlernen, zunächst schreiben wir jedoch ein Programm, das eine Annäherung an die in Folie 34 gezeigte unendliche Summe berechnet.

2.7.1 `02_03_sum_xn_funktion`

Im ersten Beispielprogramm haben wir die Funktion `sum_xn` implementiert, in die wir die Berechnung auslagern. Wir binden drei Headerdateien ein. `stdio.h` kenne wir schon, aus `math.h` werden wir die Funktion `fabs` nutzen und aus `stdlib.h` die Funktion `exit` zum Beenden des Programms mit einem Rückgabewert. Die Definition der `sum_xn`-Funktion steht *vor* der Definition der `main`-Funktion, damit der Compiler innerhalb der `main`-Funktion weiß, was mit `sum_xn` eigentlich gemeint ist.
Die Funktion erhält einen Startwert `x` (Fielßkommazahl in doppelter Genauigkeit `double`), eine Maximalzahl an Iterationen `nmax` (Ganzzahl, `int`) und eine Toleranz. Alle Argumente sind sind mit `const` vermerkt. Es ist allgemein eine gute Konvention, Funktionsargumente als `const` zu übergeben, da der Compiler somit weiß, dass sich die Variablen während der Funktionsaufführung nicht mehr verändern.
In der Funktion überprüfen wir, dass der Absolutwert von `x` kleiner als 1.0 ist. Ist dies nicht der Fall, wird das Programm mit dem Rückgabewert 55 beendet.
In der Funktion werden Variablen initialisiert, die jeweils die Summe (`S`) und den momentanen Wert von x^n (`xn`) enthalten. In einer `for`-Schleife wird bis `n == nmax-1` iteriert oder es wird die Toleranz erreicht, wobei die Schleife mit `break` verlassen wird.

Rückgabewerte von Programmen: Was bedeutet hier der Rückgabewert des Programms? Wenn ein Programm beendet wird, kann es dem Betriebssystem einen Rückgabewert zurückliefern. Damit kann man, z.B., Fehlerbedingungen mit Zahlen assoziieren und diese dann bei Beendung des Programms auswerten. Auf Unix-systemen (Linux, MacOS X und co.) in der Konsole kann man den Rückgabewert des zuletzt ausgeführten Programms durch die Ausgabe der Umgebungsvariable `$?` anzeigen lassen.

```
# ls bei einer Datei, die nicht existiert
 $ ls test
ls: cannot access 'test': No such file or directory
 $ echo $?
2
# ls bei einer Datei, die existiert
 $ ls Makefile
Makefile
 $ echo $?
0
```

2.7.2 `02_04_sum_xn_eingabe`

Wir erweitern jetzt das Programm um das Einlesen (von der Standardeingabe) des Anfangswerts für `x`, sowie der maximalen Iterationszahl und der Toleranz. Hierbei überprüfen wir den Rückgaberwert von `scanf` und beenden das Programm, falls dieser nicht unseren Erwartungen entspricht.

2.8 Deklaration und Definition

Im Beispiel `02_05_funktion_deklaration.c` wird veranschaulicht, wie man Funktionsdeklaration und Funktionsdefinition voneinander trennen kann. In der Funktionsdeklaration sind nur die Datentypen der Argumente angegeben, das genügt dem Compiler, um zu wissen, wie er mit dem Aufruf in der `main`-Funktion umzugehen hat. Die eigentliche Funktionsdefinition befindet sich unter der `main`-Funktion, könnte aber auch in einer separaten Datei liegen (sehen wir, wenn wir über modulare Programmierung sprechen).

2.9 Funktionen: Rekursion

Der rekursive Funktionsaufruf, also eine Funktion die sich selbst aufruft, ist für viele Probleme ein verlockender Lösungsansatz und wird oft gerne genutzt. In C wird Rekursion theoretisch unterstützt, ist aber im praktischen Gebrauch mit Gefahren verbunden, da der sogenannte *call stack* ("Aufrufstapel") in C nicht unendlich tief verschachtelt werden kann. Dies kann zu schwer auffindbaren Fehlern führen.
Wir können unseren Summenalgorithmus rekursiv implementieren. Dazu ruft `sum_xn` sich selbst auf, die arithmetischen Operationen werden dabei im Aufruf selbst getätigt: es werden der Startwert `x`, der gegenwärtige Summenwert `S`, der gegenwärtige Wert von x^n und die Toleranz übergeben. Ist die Toleranz erreicht, wird der erreichte Wert von `S` den gesamten Aufrufstapel hochgereicht und schlussendlich an die `main`-Funktion zurückgegeben.
Beim Versuch, das Beispiel auszuführen, geht aber etwas schief...

2.10 `gdb` und `valgrind`

Um rauszufinden, wo es jetzt hakt, kompilieren wir das Programm nochmal mit *Debugging-Symbolen* indem wir die Argumente `-g` und `-ggdb` übergeben. Dann führen das Pro-

gramm innerhalb von `gdb` aus, indem wir zuerst gdb mit dem Programmnamen als Argument aufrufen, und dann in `gdb` das Kommando `run` übergeben.
Bei `gdb` handelt es sich um einen sogenannten *Debugger*. Durch Einbindung der Debuggingsymbole kann der Debugger den erzeugten Maschinencode mit dem Quelltext in Verbindung bringen, und uns mehr oder weniger genau sagen, wo etwas schief gelaufen ist. In diesem Fall hilft `gdb` leider nicht besonders (je nach Compiler und `gdb`-Version).
Ein weiteres nützliches Programm, um Programmfehler zu verstehen ist `valgrind`. Dabei handelt es sich zwar nicht um einen klassischen Debugger, `valgrind` meldet jedoch ganz klar, was das Problem ist:

```
  $ valgrind ./02_06_sum_xn_rekursiv
==1266== Memcheck, a memory error detector
==1266== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1266== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1266== Command: ./02_06_sum_xn_rekursiv
==1266==
==1266== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==1266==
==1266== Process terminating with default action of signal 11 (SIGSEGV)
==1266==  Access not within mapped region at address 0x1FFE801FF8
==1266== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==1266==    at 0x1086DC: sum_xn (02_06_sum_xn_rekursiv.c:4)
[...]
```

Es handelt sich um einen sogenannten Stapelüberlauf (stack overflow).
Wenn ein Compiler eine Funktion ausführt, so wird Speicher bereitgestellt für die Argumente, sowie für den Rückgabewert dieser Funktion. Dieser Speicher liegt im sogenannten "Stapelspeicher", welcher endlich groß (und recht klein) ist. Führen wir nun sehr viele Rekursionen aus, wächst der Bedarf an Stapelspeicher immer weiter bis er ausgeht und das Programm abstürzt.

2.11 Statische Arrays

Statische Arrays sind Sammlungen meherer Elemente des gleichen Datentyps. In C muss die Anzahl Elemente als numerische Konstante vorliegen, es ist also z.B. nicht möglich, die Größe eines statischen Arrays aus einer Variable herzuleiten, die nicht `const` ist. Ist eine Variable `const`, muss sie natürlich bei der Deklaration auch definiert werden, es ist also keine Variable mehr, sondern eine Konstante.
Mithilfe von geschweiften Klammern kann man bei der Deklaration direkt auch die Initialisierung vornehmen. Dabei können entweder Initialwerte für alle Elemente, oder aber nur für einige Elemente festgelegt werden. Ist ein Array als `const` deklariert, können die Werte der Elemente nur bei der Initialisierung gesetzt werden.

2.12 Preview: Zeiger auf Daten

Das wohl komplizierteste Thema in C ist der Umgang mit Zeigern, weswegen wir mehrmals auf die Nutzung von Zeigern eingehen werden. In einer Zeigervariable kann die Adresse einer Variablen gespeichert werden, dabei müssen der Zeigerdatentyp und der Variablendatentyp übereinstimmen.
Im Beispiel haben eine ganzzahlige Variable `x` mit dem Wert 4 initialisiert. Wir definieren jetzt eine weitere Variable des Datentyps `int*` ("int-Sternchen"), um die Adresse der Variable `x` abzuspeichern. Der Datentyp `int*` ist ein "Zeiger auf `int`". Wieso es für

verschiedene Datentypen unterschiedliche Zeigertypen gibt, werden wir sehen, wenn wir uns mit der sogenannten Zeigerarithmetik beschäftigen.
Die Variable `zeiger_auf_x` zeigt auf den gleichen Speicher, wie die Variable `x` selbst. Mit dem Sternchenoperator kann man `zeiger_auf_x` *dereferenzieren*, damit man auf den Wert zugreifen kann. Wie wir sehen, können wir mit `printf` über `*zeiger_auf_x` den Wert von `x` ausgeben.
Mit dem Platzhalter `%p` können wir auch die Adresse ausgeben lassen. Wie wir sehen, zeigt `zeiger_auf_x` auf die gleiche Speicherstelle wie `x`.
Über einen Zeiger können wir auch in `x` schreiben, indem wir `(*zeiger_auf_x)` auf der linken Seite des Zuweisungsoperators angeben.
Zum "Tafelbid" verweise ich auf die Grafiken im Skript zum Thema Zeiger.

3 Vorlesung 3

3.1 Vervollständigung: Logische Negation

Es seit noch erwähnt, dass die Anwendung des Negationsoperators auf Variablen verschiedener Datentypen zwar in der Praxis oft zu finden ist, ich würde aber in der Regel zum Zwecke besserer Codeverständlichkeit davon abraten. Wenn man eine boolsche Variable braucht, sollte man dafür `int` nutzen.

3.2 Der `sizeof` Operator und Dereferenzierung

3.2.1 `sizeof`

Je nach Rechnerarchitektur und Compiler unterscheiden sich die Größen elementarer Datentypen von Fall zu Fall. Wir werden auch noch sehen, dass man selbst neue Datentypen definieren kann, deren Größe natürlich auch von der Rechnerarchitektur abhängen kann. Um die Größe (in Bytes) *statischer* Objekte (statische Aarrays sowie lokale und globale Variablen) zu erfragen, nutzt man den `sizeof` Operator. Dieser hat als Rückgabewert den vorzeichenunbehafteten Datentypen `size_t`, welcher im Regelfall dem größten elementaren vorzeichenunbehafteten, ganzzahligen Datenypen entspricht. Wenn wir in den nächsten Vorlesungen mit dynamischem Speicher hantieren, wird es wichtig zu sein, herausfinden zu können, welche Größe verschiedene Objekte haben können.

3.2.2 Derefernzierung

Wir hatten in der letzten Vorlesung Zeiger kurz kennengelernt. Mit dem Dereferenzierungsoperator erhält man Zugriff auf den Speicherbereich, auf den der Zeiger zeigt. Der Zeigerdatentyp bestimmt dabei, wie der Compiler diesen Speicherbereich interpretiert. Handelt es sich um einen `int*`-Zeiger, so wird der Speicherbereich bei der Dereferenzierung als `int` interpretiert (es werden also in der Regel 4 Bytes gelesen). Wenn es sich um einen `double*`-Zeiger handelt, wird der Speicherbereich bei der Derefernzierung hingegen als `double` interpretiert und es werden 8 Bytes aus dem Speicher gelesen.
Der Adressoperator `&`, angewandt auf eine Variable des Typs `double`, liefert also einen Zeiger des Typs `double*` zurück. Im Umkehrschluss bedeutet `*(&x)` also (von innen nach aussen gelesen): erhalte einen Zeiger auf `x` und dereferenziere diesen dann wieder. `*(&x)` und `x` bedeuten also das gleiche: Zugriff auf den *Wert* der Variablen `x`.
Im Beispiel `03_01_sizeof_address_dereference.c` zeigen wir die Größen in Bytes verschiedener Daten- und Zeigerdatentypen. Für die Ausgabe mit `printf` nutzen wir den Platzhalter `%lu`, also "long unsigned integer".

3.3 Zeiger

Zeiger sind also Datentypen, die auf den Anfang eines Speicherbereichs zeigen. Auf Folie 44 sehen wir den Speicher linear dargestellt und in Bytes aufgeteilt. An der Adresse 8 liegt eine Variable `c` des Typs `char` mit einer Größe eines Bytes. Die Variable `x` des Typs `double` liegt an der Adresse 10 und hat eine Größe von 8 Bytes.
Die Adressen diese beiden Variablen erhalten wir mit den Ausdrücken `&c` und `&x`. Wir können die Adresse von `x` auch in einer Zeigervariable speichern: `double *p_x = &x;`. Konventionell wird die Notation genutzt, dass das Sternchen am Zeigervariablennamen steht. Wir hätten aber auch schreiben können `double* p_x = &x;`, womit wir verdeutlichen würden, dass es sich bei der Variable `p_x` um einen Zeiger auf `double` handelt.
Es gibt auch noch einen speziellen Zeigerwert: `NULL`, dies ist ein Zeiger, der auf die Adresse 0 im Speicher zeigt. Diese Adresse darf nicht dereferenziert werden, da sonst das

Programm mit einem sogenannten *segmentation fault* (Speichersegmentierungsfehler) abstürzt.
Im Beispiel `03_02_zeiger_demo.c` definieren wir uns zwei `double` Variablen `x` und `y`, sowie zwei Zeiger, `z_x` und `z2_x`, die beide auf die Variable `x` zeigen. Durch Derefenzierung des Zeigers `z_x` auf der *linken* Seite des Ausdrucks, also `*z_x = 7.2;`, weisen wir der Variable `x` den Wert 7.2 zu.
Daraufhin weisen wir dem Zeiger `z2_x` die Adresse der Variablen `y` zu und zeigen, dass diese jetzt auch wirklich auf `y` zeigt. Schlussendlich weisen wir dem Zeiger `z2_x` noch `NULL` zu. Der Datentyp von `NULL` ist `void*`, was soviel bedeutet wie "Ein Zeiger auf alles mögliche". Damit die Zuweisung ohne Warnung abläuft, nutzen wir eine explizite Typenumwandlung in den Typ `double*`, also Zeiger auf `double`. Diese Typenumwandlung wird durch das Voranstellen eines Datentyps in Klammern vor eine Variable bewerkstelligt, mehr Details dazu sind auch im Skript zu finden.
Im letzten `printf` im Beispiel versuchen wir `z2_x`, welcher jetzt den Wert `NULL` hat zu dereferenzieren. Dies schlägt natürlich mit einem Speichersegmentierungsfehler fehl.

3.4 Arrays, Zeiger

Wenn wir ein Array `int a[4];` deklarieren, dann setzt der Compiler einen Zeiger, `a`, der auf den Anfang des von `a[4]` besetzten Speicherbereichs zeigt, `a` (ohne eckige Klammern) entspricht also der Adresse des esten Elements des Arrays `a[4]`. Wir können auf das Array `a[4]` auch einen zweiten Zeiger `b` zeigen lassen und haben dann auf die Elemente dieses Speicherbereichs sowohl über `a`, als auch über `b` Zugriff.
Hier sehen wir auch, dass man auf hintereinanderliegende Speicherelemente, auf deren Anfang ein Zeiger `b` zeigt, mit eckigen Klammern zugreifen kann. Hierbei ist `b[0]` das erste Element, dessen Adresse `&b[0]` entspricht also auch `b`. Die Adresse des nächsten Elements, `&b[1]` ist `&b[0]`, plus eine Anzahl Bytes, die der Datengröße des Datentyps entspricht. Bei einem Integer liegt das Element `b[1]` also 4 Bytes hinter dem Element `b[0]`.
Das Tafelbild, auf das hier verwiesen wird, entspricht ungefähr der Darstellung in Abb. 1.

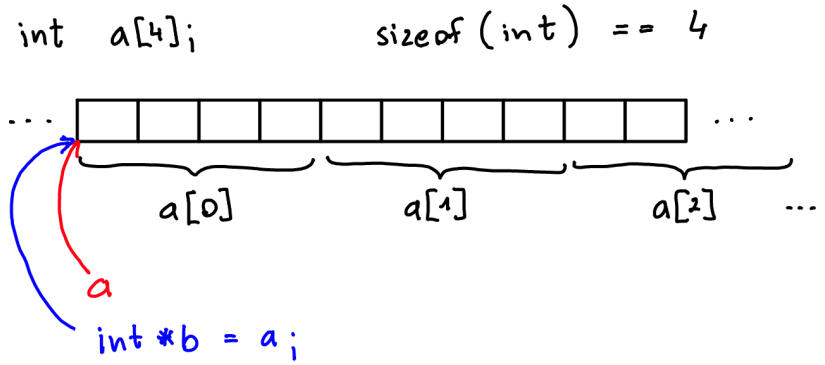

Abbildung 1: Ein Array von Integern im Speicher. Da ein Integer eine Größe von 4 Bytes hat, belegt jedes Element vier Kästchen. Die Arrayvariable `a` zeigt auf den Anfang des Speicherbereichrs, ebenso der Zeiger `b`.

Im Beispiel `03_03_zeiger_array_demo.c` wird verdeutlicht, dass Zeiger und Arrays nicht das gleiche sind. Insbesondere weiß der Compiler bei einem statischen Array, wie viele Elemente dieses enthält und kann so die Größe ausgeben. Ausserdem kann man dem "automatischen" Zeiger `a`, den der Compiler zum Array `a[4]` generiert, keine andere Adresse zuweisen.

3.5 Einfache Zeigerarithmetik

Auf den Zeigerdatentypen, mit Ausnahme von `void*`, sind auch arithmetische Operationen definiert. Wir können eine Zeigervariable, z.B., inkrementieren: `p_x++`. Für dein Compiler bedeutet dies: bei `p_x` handelt es sich um einen Zeiger auf Integer und es wird verlangt, dass im Speicher auf das nächste Element gezeigt wird, das genau um die Größe eines Integers weiter im Speicher liegt, also bei den meisten Architekturen 4 Bytes. Nach der Inkrementierung zeigt der Zeiger `p_x` also nicht mehr auf `x`, sondern auf den Speicherbereich, der 4 Bytes hinter der Adresse `&x` liegt.
Genau wie mit dem Inkrementierungsoperator, können wir einen Zeiger auch um einen ganzzahligen Wert erhöhen. Der Zeiger `p2_y` zeigt auf den Speicherberech, der genau um die Größe eines `double` (also 8 Bytes) hinter der Adresse von `y` liegt.
Auf Folie 46 sind also zwei Zeiger, `p_x` und `p2_y`, die unter Umständen auf Speicherbereiche zeigen, auf die nicht zugegriffen werden darf. Würden wir diese Zeiger dereferenzieren, würden wir entweder irgendwelche anderen Daten unseres Programms lesen, oder aber es kommt zu einem Speichersegmentierungsfehler.
Wir können mit Zeigerarithmetik auch auf die Elemente eines Arrays zugreifen. Der Zeiger `p2_z` zeigt zunächst auf das erste Element des Arrays `z`. Bei der Zuweisung in der letzten Zeile wollen wir auf die Adresse Zugreifen, die 16 Bytes hinter der Anfangsadresse des ersten Elements von `z` liegt. Diese Adresse dereferenzieren wir und weisen dem Arrayelement den Wert 4.5 zu. Wir hätten auch schreiben können: `z[2] = 4.5;`. Dies ist auch in Abb. 2 verdeutlicht.

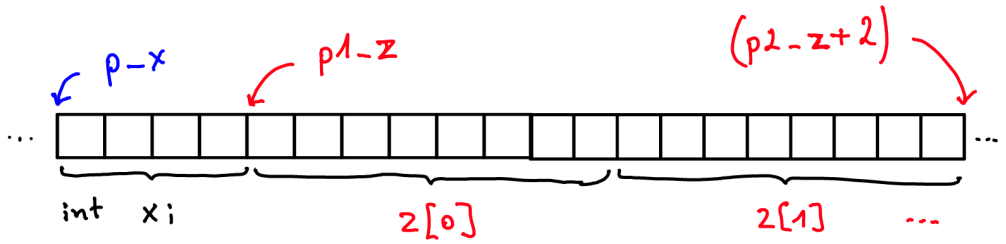

Abbildung 2: Die Variable `x` und das `double` Array `z` liegen linear hintereinander im Speicher. Der Zeiger `p_x` zeigt auf den Anfang des Speicherbereichs von `x`. Die Zeiger `p1_z` und `p2_z` Zeigen auf den Anfang des Arrays `z`. Wenn wir zu `p2_z` die Ganzzahl 2 hinzufügen, entspricht dies einem Zeiger auf das dritte Element von `z`, also `z[2]`, da wir "zwei Schritte" von jeweils 8 Bytes durch den Speicher machen.

3.6 Zeichenketten

Strings, also Zeichenketten, sind in C Arrays von `char` Elementen, wobei ein String mit dem Zeichen `\0` beendet wird, dem sogenannten *Nullterminierzeichen*. Ein String kann also maximal $n-1$ Zeichen halten.
Ein Zeiger auf eine Zeichenkettenkonstante, also etwas wie `char * str = "String"\;` sollte mit dem `const`-Modifikator *vor* dem Sternchen versehen werden. Diese Notation bedeutet: der Zeiger zeigt auf den Anfang eines *konstanten* `char`-Arrays, also `char const * str2 = "String";`. Würde man jetzt versuchen, über diesen Zeiger den Speicher zu verändern, würde der Compiler sich darüber beschweren. Hätte man den `const`-Modifikator nicht, würde das Programm abstürzen, da in den Speicher, der für Zeichenkettenkonstanten vom Compiler bereitgesteltt wird, nicht geschrieben werden darf.
Das kann im Beispiel `03_05_test_strings.c` ausprobiert werden, indem man in Zeile 16 den `const`-Modifikator entfernt. Dort wird auch gezeigt, wie unsicher `printf` ist. Die Adresse `&x` wird einfach als `char`-Array interpretiert und mit dem `%s`-Platzhalter

ausgelesen.

3.7 Strings vergleichen und bearbeiten: `strcmp` und `snprintf`

In der Headerdatei `string.h` sind viele Funktionen für den Umgang mit Strings deklariert, es sind jedoch leider sehr viele intrinsisch unsicher, in dem Sinne, dass diese Funktionen, wenn man nicht vorsichtig ist, ungehindert in irgendwelchen Speicher reinschreiben können. Die meisten Sicherheitsfehler in Web- und Desktopsoftware sind auf einen unsicheren Umgang mit Strings zurückzuführen. Diese Bugs sind oft sehr schwer auffindbar zu machen, weil es sein kann, dass die Programmlogik auf subtile Art und Weise scheitert.

Manpages: Ein kleiner Einwurf an dieser Stelle zum Thema `manpages` (oder auch "manual pages"). Auf UNIX-systemen sind große Teile der Systembibliotheken und Programme in diesen manpages dokumentiert. So auch die C-Standardbibliothek: mit dem Kommando `man 3 strcmp` können wir auf die manpage zur Funktion `strcmp` zugreifen. Hier sehen wir, in welcher Headerdatei diese Funktion deklariert ist, aber auch, wie viele
Dies wird im Beispiel `03_06_unsichere_zeichenketten.c` anhand der Funktion `strcpy` verdeutlicht. Es wird der String `str` in den *kürzeren* String `str2` kopiert, wobei das Nullterminierzeichen verlorengeht und `printf` nicht mehr weiß, wo der eine String aufhört und der andere anfängt.
Wie schon im Skript betont, möchte ich auch hier argumentieren, dass die Funktion `snprintf` eine ziemlich sichere Alternative zu unsicheren Stringfunktionen bietet, sofern man die Argumente der Funktion richtig angibt und den Rückgabewert überprüft.

- das erste Argument von `snprintf` ist die Zieladresse (ein Zeiger auf `char`), also der Zielstring
- das zweite Argument ist die maximal zu schreibende Anzahl Zeichen, *inklusive* des Nullterminierzeichens
- der Rückgabewert entspricht der Anzahl Zeichen, die versucht wurde zu schreiben
- überschreitet die Länge des transformierten Formatstrings die maximal zu schreibende Anzahl Zeichen n, so schreibt `snprintf` $n-1$ Zeichen plus ein Nullterminierzeichen und gibt dann einen Rückgabewert zurück, der größer als n ist. Man kann also abfangen, falls der zu schreibende String zu lang war.

Im Beispiel `03_07_test_snprintf.c` wird dies nochmal verdeutlicht. Es wird versucht 104 Zeichen zu schreiben, obwohl $n = 40$. `snprintf` schreibt 39 Zeichen und setzt das vierzigste Zeichen (an Index 39) auf den Wert `\0` und gibt zurück, dass versucht wurde einen String mit 104 Zeichen zu schreiben.
Noch etwas zur Notation: `const char * str` und `char const * str` bedeuten das gleiche. Anders verhält es sich mit `char * const str`, hier wird der `const`-Modifkator *hinter* das Sternchen gesetzt. Das bedeutet, dass es sich bei `str` um einen konstanten Zeiger auf einen nicht-konstanten Speicherbereich handelt. Für einen solchen Zeiger gilt also, dass man nicht ändern kann, auf welchen Speicherbereich der Zeiger zeigt, jedoch durchaus diesen Speicherbereich über diesen Zeiger verändern darf.

3.8 Pass-by-reference

Wie auf Folie 49 beschrieben kann der dort gezeigte Code nicht kompiliert werden, da die Variable `x` nur innerhalb der `main`-Funktion bekannt ist. In der Praxis kann es

jedoch sein, dass wir mit einer Funktion eine große Menge an Daten irgendwie bearbeiten möchten, um daraus, z.B., einen Mittelwert zu bestimmen. Diese jedoch *by value* an eine Funktion zu übergeben (also eine Kopier der Daten zu übergeben) ist ineffizient. Auch wenn wir innerhalb einer Funktion Daten aus einem anderen Block bearbeiten möchten, geht dies nur, wenn wir dafür Zeiger nutzen.
Man spricht dabei von *pass-by-reference*, also "übergeben als Referenz". Anstelle der Daten übergeben wir einfach einen Zeiger auf die Daten, wie auf Folie 50 gezeigt. Die Signatur der Funktion `inkrement` enthält als erstes Argument eine Zeigervariable des Typs `int*`, also einen Zeiger auf einen Integer. Beim Aufruf dieser Funktion in der `main`-Funktion übergeben wir die Adresse von `x`, sowie den Wert, um den wir `x` inkrementieren möchten.
Das Beispiel veranschaulicht das Ganze nochmal anhand unserer Summenfunktion.

3.9 Kommandozeilenargumente

Wie schon vorher im Begleitskript erwähnt, kann die `main`-funktion vom Betriebssystem Kommandozeilenargumente entgegennehmen. In ihrer Signatur wird die Zahl der Argumente über das erste Argument, `argc`, übergeben. Auf die eigentlichen Argumente wird über den Doppelzeiger `char **argv` zugegriffen. Es handelt sich hierbei um ein Array von Arrays von `char`, also um ein Array von Zeichenketten.
Das erste Argument (`argv[0]`) ist immer der Programmname und entspricht genau dem Aufruftext. Mit dem Beispielprogramm `03_09_test_argc.c` kann man ausprobieren, wie es aussieht, wenn man das Programm mit seinem lokalen Pfad `./03_09_test_argc` ausführt, oder den gesamten Pfad angibt, `$(pwd)/03_09_test_argc`[1]
In `03_10_test_read_argv.c` schauen wir uns dann noch beispielhaft an, wie man die Kommandozeilenargumente auslesen kann.
An dieser Stelle eine Empfehlung: auf UNIX-Systemen gibt es die `getopt`-Bibliothek (`man 3 getopt`) mit deren Hilfe man komplexe Kommandozeilenargumente definieren kann, wie man sie von verschiedenen UNIX-Programmen her kennt. Darunter fallen z.B. sogenannte *flags*, also so etwas wie `-x`, aber auch wertbehaftete Argumente, wie `--file="pfad/zu/einer/datei"`, die sich mit `getopt` un einem `switch`-Ausdruck relativ leicht auslesen lassen.

[1] das Programm `pwd` gibt in der Konsole den momentanen Pfad zurück und die Notation `$(kommando)` wird ersetz durch diese Rückgabe. Einfach mal ausprobieren!

4 Vorlesung 4

4.1 Rückblick auf Zeiger

Wenn ich die Vorlesung halte, lasse ich diese Folie ein paar Minuten stehen und warte auf Fragen um zu sehen, ob es hier noch Unklarheiten gibt. An dieser Stelle nur eine kurzer Kommentar zu der Zeile `z_y = arr_y`: auf der Folie steht "Zeiger `z_y` zeigt jetzt auf `arr_y`". Das ist genau gesehen nicht ganz richtig, da der Zeiger `z_y` auf das erste Element von `arr_y` zeigt. Wenn man diesen Zeiger derferenziert, `*z_y`, bedeutet dies das gleiche wie `z_y[0]`.

4.2 Vervollständigung: Zeiger auf lokale Variablen

Wenn innerhalb einer Funktion eine lokale Variable definiert wird, initialisiert der Compiler dafür Speicher, welcher beim Verlassen dieser Funktion wieder freigegeben wird. Geben wir aus einer Funktion einen Zeiger auf diesen Speicher zurück, zeigt dieser natürlich auf einen nicht mehr allokierten Bereich. Beim Versuch, diesen Zeiger zu derferenzieren, stürzt das Programm mit einem Speichersegmentierungsfehler ab.

4.3 Arrays übergeben

Im Beispiel `04_01_array_ubergabe.c` schreiben wir uns ein Programm, welches die Koordinaten von N Teilchen im dreidimensionalen Raum initialisiert und dann die Entfernung jedes Teilchens vom Ursprung des Koordinatensystems bestimmt. In der `main`-Funktion definieren wir uns zunächst drei Arrays für die x-, y- und z-Komponenten unserer Koordinaten, sowie ein Array `S`, in dem wir die Distanz zum Ursprung jedes Teilchen abspeichern wollen.
Zur Initialisierung der Koordinaten, übergeben wir diese, als auch die Teilchenzahl N an die Funktion `pos_initialisieren`. Da wir die Anfangsadressen dieser Arrays übergeben haben, kann die Funktion jetzt in diese Speicherbereiche schreiben. Zur Signatur der Funktion: die Zeiger `p_x`, `p_y` und `p_z` haben keine `const`-Modifikatoren, da wir ja in den Speicher schreiben möchten, auf den diese Zeiger zeigen. Dafür wird `N` aber als Konstante übergeben und kann innerhalb der Funktion nicht verändert werden.
Um die Betragsquadrate zu berechnen, übergeben wir wieder Zeiger auf unsere Koordinatenarrays, diesmal jedoch mit `const`-Modifikator *vor* dem Sternchen: innerhalb der Funktion kann also nicht in den Speicher geschrieben werden, auf den diese Zeiger zeigen. Wir übergeben auch einen Zeiger auf unser Array von Distanzen und schreiben für jedes Teilchen die Quadrawurzel des Betragsquadrats dort hinein. Wir müssen immer die Gesamtzahl Teilchen N übergeben, da es in C nicht möglich ist, dass die Funktionen die Größe der Speicherbereiche erfragen können, auf die mit Zeigern verwiesen wird.
Es ist guter Stil, Zeiger, die auf Speicher deuten, welcher nur gelesen wird, mit `const`-Modifikatoren zu versehen. Erstens ist so das *Interface* der Funktion klarer: schon an der Signatur erkennt man, dass die übergebenen Speicherstellen von der Funktion nicht beschrieben werden. Zweitens wird das Programm dadurch unter Umständen schneller: wenn der Compiler weiß, dass es sich um konstante Speicherbereiche handelt, müssen diese auch nur ein einziges Mal aus dem Speicher geladen werden, selbst, wenn sie mehrmals genutzt werden.
Wir können an dieser Stelle auch mal ins `Makefile` schauen, da wir die Funktionen aus der `math.h`-Headerdatei nutzen. Diese Funktionen sind nicht Teil der C-Standardbibliothek, sondern werden über das Argument `-lm` zu `gcc` in die Programme eingebunden. `-l` ist das Arguemnt für das Einbinden einer Bibliothek und `m` ist in diesem Fall der Name dieser Bibliothek.

4.4 Zeiger auf Zeiger auf Zeiger auf Zeiger ...

In der Praxis wird man oft mit verschachtelten Zeigerhierarchien konfrontiert, z.B., wenn man mit mehrdimensionalen Arrays arbeitet. Eine andere Anwendung findet sich in iterativen Algorithmen, wenn man zwei oder mehr Vektoren hat, und diese Vertauschen muss. Zum Beipsiel, könnte man einen Vektor haben, der das Ergebnis von Iteration $i-1$ enthält und einen weiteren Vektor mit der gegenwärtigen Iteration i. Am Ende der Iteration möchte man die beiden Vektoren vertauschen, sie zu kopieren wäre jedoch Verschwendung. Viel effizienter ist es, zwei Zeiger zu haben, die man jeweils vertauscht und die dann im Wechsel auf den einen oder anderen Vektor zeigen.
Im Beispiel `04_02_zeiger_vertauschen.c` probieren wir das mal aus. Wir haben ein Programm, das zwei Arrays `x` und `y` erstellt und in der Funktion `init` initialisiert. Auf diese Arrays zeigen jeweils die Zeiger `p_1` und `p_2`. Wir wollen jetzt in einer zweiten Funktion `swap_ptr` die Zeiger vertauschen. Dazu müssen wir dieser Funktion die *Adressen* der beiden Zeigervariablen übergeben, weswegen die Signatur der Funktion `swap_ptr(double **p1, double **p2)` ist, also zwei Argumente mit dem Datentyp "Zeiger auf Zeiger auf `double`" erhält. Innerhalb der Funktion speichern wir die in `p1` gespeicherte Adresse in einer temporären Zeigervariablen `tmp`. Wir müssen dafür `p1` dereferenzieren, d.h., aus `**double` wird `*double`, was `&x` aus der `main`-Funktion entspricht. Dem Zeiger `p_1` aus der `main`-Funktion wird jetzt die in `p_2` gespeicherte Addresse zugewiesen und `p_2` wiederum wird die in `tmp` gespeicherte Adresse zugewiesen. Wichtig ist es hier zu verstehen, wieso an den gezeigten Stellen derferenziert werden muss.

4.5 Modulare Programmierung

Ein "echtes" C-Programm besteht nicht nur aus einer Datei, sondern aus vielen Modulen, die schlussendlich zu einem Programm *verlinkt* werden. Beim kompilieren (`gcc -c`) wird der Quelltext zunächst in Maschinencode übersetzt, danach wird aus den einzelnen Modulen das eigentliche Programm verknüpft.

4.6 Modulare Programmierung: Quell- und Header-dateien

Wir haben natürlich schon systemeigene Headerdateien in unsere Programme eingebunden, möchten aber auch unsere eigenen Programme aus Modulen aufbauen, da diese sonst schnell unüberschaubar werden. Der weitere Vorteil eines modularen Aufbaus ist, dass man Komponenten einfach ersetzen kann.
Die Folie zeigt ein einfaches Beispiel für zwei Module eines Programms. Einerseits die Funktion `AblT`, welche die Zeitableitung einer Positionsvariable (also die Geschwindigkeit) berechnet, andererseits die Funktion `kinEnerg`, welche die kinetische Energie bestimmt. Zu jedem Modul gibt es eine Quelltextdatei mit der Funktionsdefinition und jeweils eine Headerdatei mit der Funktionsdeklaration. Da die Funktion `kinEnerg` die Zeitableitung der Position benötigt, binden wir die Headerdatei `AblT.h` ein, damit der Compiler weiß, dass es eine solche Funktion gibt und welche Signatur sie hat. Wenn das Programm später verknüpft wird und die Funktion `AblT` aufgerufen wird, sucht das Programm in allen verknüpften Modulen nach dieser Funktion und wird in `AblT.o` fündig werden.

4.7 Compiler und Linker

Die Folie zeigt uns die Schritte, um aus einer Modulsammlung ein ausführbares Programm zu erhalten. Die `main`-Funktion unseres Beispiels befindet sich in `teilchen/teilchen.c`. Es wird zunächst ein Array mit 1000 Elementen deklariert und dann initialisiert, dabei

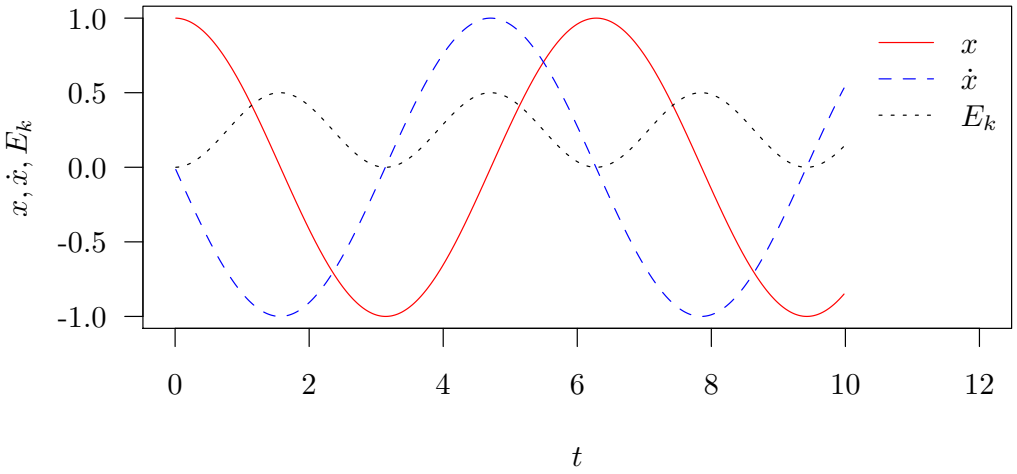

Abbildung 3: Position, Geschwindigkeit und kinetische Energie eines Teilchens aus dem Beispiel `04/teilchen/teilchen.c` (willkürliche Normierung).

handelt es sich um die Position eines Teilchens als Funktion des diskreten Zeitschritts n. Für jeden Zeitschritt werden jetzt die Zeit, die Position, die Geschwindigkeit und die kinetische Energie ausgegeben. Nachdem wir das Programm kompiliert haben, können wir diese Ausgabe in eine Datei umleiten:

```
$ gcc -c kinEnerg.c
$ gcc -c AblT.c
$ gcc -c teilchen.c
$ gcc -o teilchen teilchen.o kinEnerg.o AblT.o -lm
$ ./teilchen > x.dat
```

Wenn wir Position, Geschwindigkeit und kinetische Energie grafisch darstellen, sieht das in etwa so aus, wie in Abb. 3 gezeigt.

4.8 Modulare Programmierung: Interface und Implementierung

Der große Vorteil einer modularen Programmierweise ist die klare Trennung zwischen dem *Interface* und der *Implementierung*. Wenn wir z.B. eine der Bibliotheksfunktionen von C nutzen, ist es uns ganz egal, wie diese Funktion implementiert wurde. Was wir lediglich wissen müssen ist, welchen Input die Funktion benötigt, welchen Output sie liefert und welche Nebeneffekte sie hat. Ein Nebeneffekt könnte z.B. sein, dass ein Speicherbereich, der via Zeiger übergeben wurde von der Funktion überschrieben wird. Für unser Beispiel können wir die Unabhängigkeit von Interface und Implementierung nutzen, um die Implementierung von `AblT` einfach zu ersetzen. Wir können anstelle der Vorwärtsableitung eine symmetrische Ableitung nutzen, diese hat kleinere Näherungsfehler. Wenn wir das Programm kompilieren, können wir anstelle des Moduls `AblT.o`, einfach das Module `AblT_symmetrisch.o` einbinden.

4.9 Der C-Präprozessor

Bevor der Compiler Quelltext kompiliert, läuft der sogenannte Präprozessor. Dieser bewirkt z.B., dass beim Kompilieren die Headerdateien eingebettet werden, aber auch, dass sogenannte Makros ersetzt werden. Man spricht auch von *Präprozessorkonstanten*. Wie in Beispiel `04_03_test_makros.c` gezeigt, unterstützt der Präprozessor auch

rudimentäre Verzweigung. Wir können z.B. überprüben, ob ein Makro definiert worden ist und dann den einen oder den anderen Teil eines Quelltextes kompilieren. Hier wird in Abhängigkeit davon, ob `MY_EULER` definiert wurde, entweder der entsprechende Wert ausgegeben, oder darauf verwiesen, dass das Makro nicht bekannt ist. In der Praxis wird eine solche bedingte Kompilierung dazu genutzt, um Code z.B. für verschiedene Architekturen auszulegen, oder in einer *Debugging*-version mit mehr Ausgabe zu kompilieren.

4.10 Zusammengesetzte Datenstrukturen

Auf die Elemente einer zusammengesetzten Datenstruktur wird mit der `.`-Notation zugegriffen. Hat man jedoch einen Zeiger `z_t` auf ein `struct`, nutzt man den Pfeil `z_t->x`, oder aber, man dereferenziert den Zeiger und nutzt dann den Punkt: `(*z_t).x`. Der Datentyp einer Zusammengesetzten Datenstruktur muss immer mit `struct` markiert sein, wir werden jedoch sehen, wie man mit einem sogenannten `typedef` Kürzel für eigene Datentypen erstellen kann.

4.11 Mehrfachdefinitionen?

Haben wir jetzt eine eigene Datenstruktur erfunden, z.B. für die Position eine Teilchens im mehrdimensionalen Raum, möchten wir diese natürlich in den verschiedenen Modulen unseres Programms nutzen. Dazu wird die Definition der Datenstruktur in der Reger in eine Headerdatei gepackt und diese wird dann überall dort eingebunden, wo man die Datenstruktur nutzen möchte. Leider entsteht hierbei eine Mehrfachdefinition.
Im Beispiel `04/teilchen_pos` haben wir eine Datenstruktur `stuct pos_st` eingeführt, welche `x` und `y`-Koordinaten eines Teilchens bündelt. Kommentieren wir dort in der Datei `pos_st.h` das `# pragma once` aus:

```
//#pragma once
struct pos_st {
  double x;
  double y;
};
```

wird der Compiler sich beschweren:

```
 $ gcc -c teilchen.c
In file included from AblT.h:1:0,
                 from teilchen.c:2:
pos_st.h:3:8: error: redefinition of ‘struct pos_st’
 struct pos_st {
        ^~~~~~
In file included from kinEnerg.h:2:0,
                 from teilchen.c:1:
pos_st.h:3:8: note: originally defined here
 struct pos_st {
        ^~~~~~
In file included from teilchen.c:3:0:
pos_st.h:3:8: error: redefinition of ‘struct pos_st’
 struct pos_st {
        ^~~~~~
[...]
```

Das `# pragma once` bedeutet für den Präprozessor, dass der Inhalt dieser Datei nur ein einziges Mal in einer Übersetzungseinheit eingefügt werden soll. Leider wird

`# pragma once` nicht von allen Compilern unterstützt und funktioniert auch nicht in allen Fällen hunderprozentig. Konventionell werden daher sogenannte *header guards*, auch *include guards* genannt, genutzt.

4.12 Include / Header Guards

Die include guards bestehen aus einer Abfrage `# ifndef NAME`, in der abgefragt wird, ob die Präprozessorkonstante `NAME` *nicht* existiert. Ist dies also nicht der Fall, wird diese Konstante definiert und die darunterstehenden Deklarationen werden einkompiliert. Wird jetzt diese Headerdatei ein weiteres Mal eingebunden, ist die Präprozessorkonstante schon definiert und der Inhalt der Datei wird von `#ifndef NAME` bis zum nächsten `#endif` übersprungen. Es ist guter Stil hinter das `#endif` einen Kommentar mit dem Namen des dazugehörigen Präprozessorkonstante zu setzen. In dem auf Folie 66 gezeigten Beispiel könnte man `#endif // ifndef(ABLT_H)` schreiben, um zu markieren, dass dieses `#endif` das Ende dieser logischen Bedingung stellt.
Im Beispiel `04/teilchen_pos_guarded` nutzen wir header guards, damit `struct pos_st` nur ein einziges Mal definiert wird. Genauer hätten wir auch in `kinEnerg.h` und `AblT.h` header guards einführen sollen.

5 Vorlesung 5

5.1 Zeigerarithmetik Quiz

Auf diese Folie sollte man ein paar Minuten starren um zu versuchen, die ganzen gezeigten Ausdrücke nachzuvollziehen. Wir haben ein Integer Array `int k[10];`, einen Zeiger `z1_k` auf den Anfang dieses Arrays und einen weiteren Zeiger `z2_k` auf das dritte Element des Arrays. Mit dem Beispiel `05_00_test_zeigerarithmetik.c` kann man noch ein wenig rumspielen, um sein Verständnis zu testen.

5.2 Dynamische Speicherverwaltung

Wollen wir in C ein Array einer beliebigen Größe erstellen, welche z.B. vom Wert einer Variablen abhängt, so müssen wir den Speicher dafür dynamisch allokieren. Speicher, der nicht mehr genutzt wird, muss per Hand wieder freigegeben werden.
Die Speicherallokationsfunktion `malloc` allokiert einen *zusammenhängenden* Speicherbereich, dessen Größe wir in *Bytes* angeben müssen und gibt einen Zeiger des Typs `void*` auf den Anfang zurück. Bei der Zuweisung an einen `double*`-Zeiger, z.B., sollten wir also ein Typecast `(double*)` einfügen, damit der Compiler keine Warnungen liefert. Schlägt die Allokation aus irgendeinem Grund fehl, liefert `malloc` einen `NULL`-Zeiger zurück, der Rückgabewert von `malloc` sollte also immer überprüft werden. Über den Zeiger `x` können wir jetzt ganz normal auf die Speicherstellen zugreifen, als handele es sich um ein Array. *Wichtig:* Zugriffe vor das erste, sowie hinter das letzte Element des allokierten Speicherbereichs werden anstandslos und ohne Warnung durchgeführt. Im besten Fall stürzt das Programm mit einem Speichersegmentierungsfehler ab, wenn man versucht in einen nicht allokierten Bereich zu schreiben. Im schlimmsten Fall jedoch gehört dieser Bereich auch schon dem Programm und man überschreibt einfach irgendwelche Speicherstellen mit unvorhersagbaren Konsequenzen (falsche Ergebnisse, Logikfehler etc . . .).

5.2.1 `05_01_test_malloc.c`

Das erste Beispielprogramm nimmt zwei Kommandozeilenargumente entgegen: wie viele Elemente wir allokieren wollen (`groesse`) und welches Element des allokierten Speicherbereichs wir ausgeben möchten, `element`. Es wäre an dieser Stelle besserer Stil gewesen, die Variablen `groesse` und `element` mit `const`-Modifkatoren zu versehen, da es sich hierbei klar um Konstanten handelt.
Wir allokieren also einen Speicherbereich der Größe `sizeof(double) * groesse`, überprüfen, ob die Allokation erfolgreich war, initialisieren die Elemente in einer `for`-Schleife und geben dann das gewünschte Element aus. Schlussendlich geben wir den genutzten Speicher wieder frei.

5.2.2 `05_02_test_malloc_segfault.c`

In diesem Beispiel versuchen wir Speichersegmentierungsfehler zu verursachen, um zu zeigen, wie sehr man aufpassen muss. Einer der Gründe, wieso C und verwandte Sprachen zu effizientem Maschinencode kompiliert werden können, ist der Mangel an sogenannten *bounds checks*, also Überprüfungen, ob ein Speicherzugriff erlaubt ist oder nicht.
An dieser Stelle sollten wir uns ansehen, wie der Rechner den Speicher eigentlich nutzt. Die Details zur Speicherarchitektur sind nicht besonders wichtig, wir sollten jedoch wissen, dass die Speicheradressen, wie wir sie in den vorherigen Vorlesungen und Beispielen gesehen haben, ein logisches Konstrukt sind, mit denen zusammenhängende Speicherbereiche abgezählt werden.

Der vom Betriebssystem für ein Programm bereitgestellte Speicher ist zweigeteilt in den sogenannten *Stapelspeicher* (stack) und den sogennanten *Haufenspeicher* (heap). Lokale und globale Variablen, Funktionen und ihre Rückgabewerte sowie statische Arrays werden im Stapelspeicher allokiert. Wir hatten beim Rekursionsbeispiel schon gesehen, dass der stack nicht besonders groß ist und überlaufen kann. Dynamische Speicherallokation hingegen findet im heap statt, weshalb man mit einem `malloc` den gesamten verfügbaren Arbeitsspeicher allokieren kann.
Zugriffe auf verschiedene Teile des stacks und heaps können ohne Fehler*meldungen* durchgehen, selbst, wenn es sich um formal *illegale* Zugriffe handelt. Im Beispiel wird ein statisches `double`-Array mit 40 Elementen erzeugt und es wird prompt in die davor liegende Speicherstelle geschrieben. Zumindest auf meinem Rechner, ensteht hierdurch kein erkennbarer Fehler.
Auch über den Zeiger `x_dyn`, welcher auf einen Speicherbereich der Größe zweier `double` zeigt, kann unerlaubt auf irgendwelchen Speicher zugegriffen werden.
Wir können den Speicher wieder freigeben und eine erneute Allokation in der gleichen Zeigerraviablen speichern. Jetzt schreiben wir in das Element `-1`. Zunächst scheint alles zu funktionieren, versuchen wir jetzt jedoch den Speicher wieder freizugeben stürzt das Programm, zumindest auf meinem Rechner, ab. Der Grund ist, dass die Speicherstelle `x_dyn[-1]` (zumindest auf meinem Rechner) dazu genutzt wird, um die physikalische Speicheraddresse abzulegen, welche von `free` freigegeben werden soll, wenn wir versuchen `free(x_dyn)` auszuführen.
Führen wir das Programm mithilfe von `valgrind` aus, zeigt uns valgrind die ganzen illegalen Zugriffe an:

```
$ valgrind ./05_02_test_malloc_segfault
[....]
x[-1] = 4.200000
==21520== Invalid write of size 8
==21520==    at 0x108889: main (05_02_test_malloc_segfault.c:18)
==21520==  Address 0x522d470 is 16 bytes before a block of size 16 alloc'd
==21520==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==21520==    by 0x108824: main (05_02_test_malloc_segfault.c:9)
==21520==
#1 x_dyn[ 1] = 42.420000
#1 x_dyn[ 0] = 34.200000
==21520== Invalid read of size 8
==21520==    at 0x1088F0: main (05_02_test_malloc_segfault.c:22)
==21520==  Address 0x522d478 is 8 bytes before a block of size 16 alloc'd
==21520==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==21520==    by 0x108824: main (05_02_test_malloc_segfault.c:9)
==21520==
#1 x_dyn[-1] = 0.000000
==21520== Invalid read of size 8
==21520==    at 0x10891E: main (05_02_test_malloc_segfault.c:23)
==21520==  Address 0x522d470 is 16 bytes before a block of size 16 alloc'd
==21520==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==21520==    by 0x108824: main (05_02_test_malloc_segfault.c:9)
==21520==
#1 x_dyn[-2] = 4.200000
[...]
```

Aber selbst `valgrind` findet den letzten Fehler nicht, da `valgrind` die Speicherallokationsfunktion `malloc` während des Betriebs durch eine eigene Implementierung austauscht, die allem Anschein nach die für `free` relevante Speiheraddresse an einer anderen Stelle ablegt.
Speicherfehler sind sehr tückisch und oft unglaublich schwer aufzufinden.
Glücklicherweise haben wir aber noch ein Werkzeug im Ärmel, den sogenannten *address sanitizer*, der seit `gcc`-Version 4.8 existiert. Auch andere Compiler, wie z.B. LLVM-`clang` enthalten diese Funktionalität.
Bei `gcc` muss man lediglich das Argument `-fsanitize=address` beim Kompilieren hinzufügen, was bewirkt, dass die Speicherallokationsfunktionen durch (viel) langsamere Versionen ersetzt werden, die jeodch explizit Speicherzugriffe überprüfen.

```
 gcc -fsanitize=address -g -ggdb -Wall -Wpedantic -std=c99 \
 -o 05_02_test_malloc_segfault 05_02_test_malloc_segfault.c
 $ ./05_02_test_malloc_segfault
=================================================================
==23092==ERROR: AddressSanitizer: stack-buffer-underflow on address
0x7ffc49c896a8 at pc 0x565329da3f89 bp 0x7ffc49c89660 sp 0x7ffc49c89650
[...]
```

Es lohnt sich also, bevor man ein Programm zur Produktion von Forschungsergebnissen nutzt, dieses mit dem address santizer auszuführen. Man sollte jedoch beachten, dass Programme mit adress sanitizer viel langsamer laufen als ohne.

5.3 Speicherverwaltung: Memory Leaks

Ein weiterer Speicherfehler, der immer wieder auftaucht ist das sogennante *Speicherleck*, auch *memory leak* genannt. Wenn man einer Zeigervariable einen Speicherbereich zuweist, diesen dann nicht freigibt und dem Zeiger ein weiteres Mal einen Speicherbereich zuweist, geht der erste Speicherbereich verloren, da man die Addresse nicht mehr kennt. Es ist dann unmöglich, bis das Programm beendet wird, diesen Speicherbereich wieder freizugeben.
Wer schon bemerkt hat, dass ein Rechner nach mehreren Tagen Laufzeit immer weniger freien Arbeitsspeicher hat und langsamer zu werden scheint, der hat den praktischen Effekt von Speicherlecks schon gesehen. Diese treten in der Praxis in komplizierten Programmen leider sehr oft auf.
Im Beispiel `05_03_memory_leak.c` wird in einer Schleife immer wieder Speicher allokiert und mit dem Zeiger `x` verknüpft. Bevor das Programm beendet wird, geben wir nur die zuletzt allokierten Speicherbereich frei. Führen wir das Programm mit `valgrind` aus, meldet dieses, wie viel Speicher noch allokiert war:

```
 $ valgrind ./05_03_memory_leak
==25852== Memcheck, a memory error detector
==25852== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==25852== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==25852== Command: ./05_03_memory_leak
==25852==
==25852==
==25852== HEAP SUMMARY:
==25852==     in use at exit: 39,600 bytes in 99 blocks
==25852==   total heap usage: 100 allocs, 1 frees, 40,400 bytes allocated
==25852==
==25852== LEAK SUMMARY:
```

```
==25852==    definitely lost: 39,600 bytes in 99 blocks
==25852==    indirectly lost: 0 bytes in 0 blocks
==25852==      possibly lost: 0 bytes in 0 blocks
==25852==    still reachable: 0 bytes in 0 blocks
==25852==         suppressed: 0 bytes in 0 blocks
==25852== Rerun with --leak-check=full to see details of leaked memory
==25852==
==25852== For counts of detected and suppressed errors, rerun with: -
v
==25852== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Auch der address sanitizer meldet Speicherlecks:

```
$ gcc -fsanitize=address -g -ggdb -Wall -Wpedantic -std=c99 \
  -o 05_03_memory_leak 05_03_memory_leak.c
$ ./05_03_memory_leak

=================================================================
==26232==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 39600 byte(s) in 99 object(s) allocated from:
    #0 0x7f0ca1ffeb40 in __interceptor_malloc [...]
    #1 0x560d788f6a4e in main [...]
    #2 0x7f0ca1b50b96 in __libc_start_main [...]

SUMMARY: AddressSanitizer: 39600 byte(s) leaked in 99 allocation(s).
```

Eine Möglichkeit, Speicherlecks, die aus Mehrfachallokationen stammen, zu vermeiden, ist es die genutzte Zeigervariable als konstanten Zeiger zu deklarieren (`const` *nach* dem Sternchen).

```
double * const x = (double*)malloc( sizeof(double) * 10 );
x = (double*)malloc( sizeof(double) * 50 ); // FEHLER: x ist const!
```

Dies ist natürlich nicht immer möglich und bewahrt einen auch nicht vor Speicherlecks, die einfach nur durch fehlende Freigaben enstanden sind.

5.4 Mehr Speicherverwaltung

Für viele Algorithmen muss man Kopien von Vektoren oder Matrizzen erstellen. Würde dies elementweise gemacht, wäre es viel zu langsam, weshalb man mit `memmove` einfach `n` Bytes ab einer Quelladresse `src` zu einer Zieladresse `dest` kopieren kann. Zurückgegeben wird ein Zeiger auf `dest`. Die Speicherbereiche dürfen dabei überlappen, da zwischendurch temporärer Speicher allokiert wird.
Ist man sich sicher, dass die Speicherbreiche, auf die `dest` und `src` zeigen nicht überlappen, sollte man stattdessen `memcpy` nutzen.
Um gleich einen ganzen Speicherbereich auf einen bestimmten Wert zu setzen, wird `memset` genutzt.
Im Beispiel `05_04_test_memmove_memset.c` wird die Verwendung dieser Funktionen veranschaulicht. Hervorzuheben ist das `memset`: hier wird einfach so in irgendwelchen Speicher geschrieben, das Programm stürzt aber, zumindest auf meinem Rechner, nicht ab. Sowohl `valgrind`, als auch der address sanitizer beschweren sich aber natürlich, dass das, was dort abläuft, nicht erlaubt ist. (*heap-buffer overflow*)

5.5 Noch mehr Speicherverwaltung

Die Funktion `realloc` sollte genauer unter die Lupe genommen werden. Sie dient dazu, einen allokierten Speicherbereich zu vergrößern oder zu verkleinern, wobei man jedoch auf einige Feinheiten achten sollte.
Man übergibt einen Zeiger auf den Speicherbereich, den man vergößern oder verkleinern möchte und die neue Größe `n_new` in Bytes. Nehmen wir an, unser Speicherbereich auf den `ptr` zeigt hat eine Größe von `n_old` Bytes und wir wollen den Speicherbereich vergrößern, also `n_new > n_old`. Die `realloc`-Funktion wird versuchen am Ende unseres Speicherbereichs einen zusammenhängenden Speicherbereich der Größe `n_new - n_old` zu finden. Existiert ein solcher Speicherbereich, wird die Allokation vergrößert und der Rückgabewert von `realloc` entspricht dem ursprünglichen Zeiger. Wird ein solcher Bereich jedoch nicht gefunden, wird ein neuer Speicherbereich der Größe `n_new` allokiert, die ersten `n_old` Bytes dahin kopiert und der alte Speicherbereich freigegeben. Der Rückgabewert entsprich dabei nicht mehr dem ursprünglichen Zeiger, sondern es wird ein Zeiger auf den neuen Speicherbereich zurückgegeben. Schlägt all dies fehl, ist der Rückgabewet `NULL` und `ptr` bleibt weiterhin allokiert. Es sollte also erstens immer der Rückgabewert überprüft werden und man sollte zunächst *nicht* `ptr` überschreiben. Von einem `realloc`-Aufruf der Form: `ptr = realloc(ptr, n_new);` ist also dringend abzuraten.
Wird der Speicherbereich verkleinert, ist also `n_new < n_old`, bleiben `n_new` Bytes dieses Speicherbereichs unverändert, `ptr` bleibt gleich und es werden `n_old - n_new` Bytes freigegeben.

5.6 Mehrdimensionale Arrays

Für Programme in der Physik sind Mehrdimensionale arrays natürlich von höchster Relevanz. Statische mehrdimensionale Arrays haben eine einleuchtende Syntax, wobei die Matrixelemente *garantiert* linear hintereinander in einer festen Reihenfolge im Speicher liegen und der am weitesten rechts liegende Index am schnellsten "läuft". Wenn wir Zeiger auf einzelne Elemente definieren, zo zeigen diese wie dargestellt in den Speicher hinein. Wenn ein Rechner daten aus dem Speicher liest, wird meist mehr als nur ein einziges Element ausgelesen (selbst, wenn wir nur auf ein einziges Element zugreifen, liest der Rechner aus Effizienzgründen mehrere aus).
Mehrdimensionale Arrays linearisiert im Speicher abzulegen bringt deshalb Geschwindigkeitsvorteile, die wir auch ausnutzen möchten, wenn wir dynamische Allokationen für mehrdimensionale Arrays tätigen.

5.7 Linearisierung Mehrdimensionaler Arrays

Wenn wir Speicher für eine Matrix reservieren möchten, müssen wir zunächst Speicher für alle Matrixlemente allokieren. Um darin dann die einzelnen Elemente abzulegen, berechnen wir den linearen Index `i`, welcher sich aus dem Spaltenindex `s` und dem Zeilenindex `z` ergibt (das erste Element hat $s = 0$ und $z = 0$). Das Matrixelement A_{zs} wird also linear an der Stele $i = s + N \cdot z$ abegelegt.
Für ein höherdimensionales Objekt wird die Formel ensprechend verallgemeinert.

5.8 Dynamische Mehrdimensionale Arrays

Wollen wir jetzt auf eine dynamisch allokierte Matrix mit zwei Paar eckiger Klammern zugreifen, brauchen wir Speicher für die eigentlichen Matrixelemente, aber auch Speicher für Zeiger, die jeweils auf das erste Element jeder Zeile zeigen. Die Adresse des Speichers für die Matrix wird in einem `double`-Zeiger, `double * matrix_mem` abgelegt. Wir

haben N_0 Zeilen, also brauchen wir ebensoviele `double*`-Zeiger. Die Anfangsadresse dieses Speicherbereichs legen wir in einem Zeiger auf `double*` ab, dieser hat also den Datentyp `double**`. (Zeiger auf Zeiger auf `double`)
Wir erinnern uns, die Notationen `*(p+n)` und `p[n]` sind identisch, schreiben wir also `Matrix[z]`, ist der Datentyp `double*`. Dort können wir jetzt die Anfangsadresse der jeweiligen Zeile im linearen Speicher ablegen und danach mit `Matrix[z][s]` auf die Matrixelemente zugreifen.
In den Beispielen `05_06_lin_multidim_array` und `05_07_dyn_multidim_array` wird das noch weiter veranschaulicht und in Folie 79 für eine 4×3-Matrix visualisiert.

5.9 Eigene Typen anlegen: `typedef`

Ich hatte `typedef` schon kurz in der vorherigen Vorlesung angesprochen. Mit diesem Schlüsselwort lassen sich eigene Typen definieren, entweder Aliase für elementare Datentypen, wobei die Notation immer so ist, dass der Alias an letzter Stelle steht.
Bei einer zusammengesetzten Datenstruktur kann man mithilfe von `typedef` auf das `struct`-Kennwort verzichten, wenn es darum geht, die Datenstruktur zu nutzen. Wichtig: der Alias und der Name der Struktur können identisch sein, so wie im gezeigten Beispiel. Der Alias steht hier *nach* der abschließenden geschweiften Klammer und steht hinter dieser Definition für `struct teilchen_2d_t`.
Es ist konventionell (aber nicht universell), so definierte Datentypen mit einem `_t` zu markieren, um klarzustellen, dass es sich um eine irgendwo definierte Datenstruktur handelt.

6 Vorlesung 6

6.1 Eingabe und Ausgabe: `fopen`

Der Zugriff auf Dateien erfolgt in C über einen *file pointer* (Dateizeiger), der von der Funktion `fopen` zurückgegeben wird, wenn man damit eine Datei öffnet. Wichtig ist, auf jeden Fall den Rückgabewert auf `NULL` zu prüfen, um sicherzustellen, dass die Datei geöffnet werden konnte. Das zweite Argument von `fopen` ist der Modus, welcher darüber entscheidet, ob der zur Datei gehörige Stream nur gelesen, nur geschrieben oder gelesen und geschrieben werden kann. Dass der Dateizeiger sich *nicht* wie ein üblicher Zeiger verhält, sehen wir in den nächsten Folien.
Für die Modi `r` und `w` gilt, dass der Dateizeiger nach dem Öffnen an den Anfang der Datei gesetzt wird. Sollte die Datei nicht existieren, wird sie im Modus `w` erstellt. Der Modus `a` unterscheidet sich von `w` darin, dass der Dateizeiger ans Ende der Datei gesetzt wird, falls diese schon existiert, es wird also hinzugefügt und nicht überschrieben. Existiert die Datei nicht, wird sie auch im Modus `a` erstellt.
Die `x+`-Modi kombinieren Lesen und Schreiben, wobei man für `r+` bemerken muss, dass eine Datei existieren muss, damit diese erfolgreich geöffnet wird. Der `a+`-Modus ist speziell: der Schreibzeiger wird ans Ende der Datei gesetzt, der Lesezeiger jedoch an ihren Anfang.
Der Pfad kann entweder ein vollständiger, also absoluter, Pfad sein, oder aber relativ zum Ausführungsverzeichnis. Wichtig: die Pfade sind systemabhängig, möchte man also UNIX- und Windows-systeme gleichzeitig unterstützen, muss man mit `#ifdef` oder anderen logischen Verzweigungen mehrere Versionen der Lese- und Schreibfunktionen implementieren.
Benötigt man einen Dateizeiger nicht mehr, *muss* dieser mit `fclose` geschlossen werden. **Wichtig:** Systemweit ist die Zahl von Dateizeigern begrenzt[2], mehrere unvorsichtig geschriebenes Programme, welche hunderte oder tausende von Dateien öffnen, aber nicht mehr schließen, können einen Rechner komplett lahmlegen. Auf den meisten Linuxrechnern kann ein Programm etwas mehr als 1000 Dateien öffnen, bevor das Betriebssystem das Öfnnen weiter Dateien verweigert.
Das Beispiel `06_01_test_fopen_fclose.c` veranschaulicht die Verwendung von `fopen` und `fclose` nochmal.

6.2 Formatierte Textdateien lesen in `fscanf`

Genauso wie mit `scanf` und `sscanf` aus der Standardeingabe bzw. einem String formatierte Eingabe ausgelesen werden kann, nutzt man dazu bei Dateien `fscanf`. Das erste Argument der Funktion ist ein Dateizeiger, das zweite ein Formatstring und über die variable Argumentenliste werden Adressen von Zielvariablen übergeben, die dem Platzhalten im Formatstring entsprechen. Der Umgang mit `fscanf` ist unter Umständen schwierig, weil auch unvollständige Leseoperationen den Dateizeiger bewegen, es ist also sehr leicht in die Situation zu kommen, dass man nicht mehr weiß, wo in einer Datei man sich eigentlich befindet.
Wir illustrieren dies anhand des Beispiels `06_02_test_fscanf.c`, indem wir versuchen die Datei `format.txt` auszulesen.

```
Test 3.1415
Testb 2.71
```

[2] genauergesagt handelt es sich um eine Einschränkung der Zahl von *file descriptors*, in der Praxis können wir aber genausogut von Dateizeigern sprechen

Wir implementieren zunächst eine Funktion, die uns ausgibt, ob das Ende einer Datei erreicht wurde. Dazu wird ein Dateizeiger an die `feof`-Funktion übergeben, die 1 als Rückgabewert liefert, wenn dies der Fall sein sollte.
In der `main`-Funktion erstellen wir uns ein `char`-Array zum Einlesen von Strings und öffnen die Datei im Lesemodus. Wir können uns mithilfe der `ftell`-Funktion ausgeben lassen, an welcher Stelle der Dateicursor gerade liegt. Anfangs liegt dieser natürlich an Position 0, also dem Dateianfang.
Wir lesen nun einen String und eine `double`-Variable aus. Hier ist zu bemerken, dass, anders als bei `printf` ein Unterschied besteht, ob es sich um eine `double`- oder eine `float`-Gleitkommazahl handelt. Für `double` nutzen wir den Platzhalter `%lf` und für `float` würden wir den `%f`-Platzhalter. Da die Funktionen der `scanf`-Familie einfach so in den Speicher schreiben, ist es zwingend notwenig, dass die richtigen Platzhalter genutzt werden.
Wir sehen, dass der Dateicursor jetzt an Position 12 ist, dabei wurden die vier Zeichen von `Test` gelesen, das Leerzeichen, sowie die 6 Zeichen von `3.1415` und schlussendlich noch der Zeilenumbruch, also 12 Zeichen, sodass der Dateicursor jetzt am Anfang des 13. Zeichens in der Datei sitzt. Der Aufruf von `fscanf` war erfolgreich, da der Rückgabewert wie erwartet 2 beträgt, es wurden also 2 Felder gelesen.
Für den nächsten Versuch setzen wir zunächst `x = 4.2;` und nutzen absichtlich einen falschen Formatstring. Wie wir sehen, bleibt `x` unverändert, der Dateicursor hat sich jedoch um 4 Zeichen bewegt. Klar: die ersten vier Zeichen des Formatstrings tauchen auch tatsächlich in der Datei auf, werden also auch ausgelesen. Danach folgt jedoch das Zeichen `b` und nicht ein Leerzeichen, das weitere Lesen schlägt also fehl und `fscanf` liefer zurück, dass keine Felder gelesen wurden.
Dass dies wirklich der Wahreheit entspricht sehe wir am nächsten Versuch mit einem an die Situtation angepassten Formatstring, da wir wissen, dass der Dateicursor am `b` steht. Jetzt lesen wir auch die letzte Zahl aus der Datei aus und sehen, dass `feof` jetzt auch berichtet, dass wir das Ende der Datei erreicht haben.

6.3 Mit `fprintf` Textdateien schreiben

Der einzige Unterschied zu `printf` besteht darin, dass ein Dateizeiger als erstes Argument übergeben werden muss. Eine Datei kann durch mehrfaches Ausführen von `fprintf` immer weiter geschrieben werden. Auch hier ist es wichtig, vor Beendigung des Programms die Datei mit `fclose` zu schließen, da ansonsten unter Umständen nicht alles geschrieben wird, was man eigentlich schreiben wollte.

6.4 Mit `fwrite` / `fread` Binärdateien schreiben und lesen

Textdateien, wie wir sie bisher gelesen und geschrieben haben, sind nicht das einzige Datenformat, mit dem wir umgehen müssen. Wollen wir zum Beispiel Datensktrukturen aus unserem C-Programm direkt auf die Festplatte schreiben, machen wir dies mithilfe *binärer* Ausgabe, schreiben also direkt die Bitdarstellung eines Speicherbereichs.
In Beispiel `06_03_test_fwrite.c` beginnen wir zunächst damit, einen Speicherbereich für 40 `double` zu allokieren und füllen diese dann mit Daten. Wir erstellen eine Datei mit Dateinamen `x.dat` im Modus `wb`, dabei steht das `b` für den Binärmodus.
Jetzt schreiben wir mit `fwrite` 40 Blöcke mit einer Blockgröße von jeweils 8 Bytes (der Größe eines `double`). Wir überprüfen den Rückgabewert, welcher der Anzahl erfolgreich geschriebener Blöcke entspricht. Zuätzlich überprüfen wir, welchen Wert die Funktion `ferror` zurückgibt. Diese Funktion dient dazu, alle möglichen Fehlerzustände der Eingabe- und Ausgabefunktionen (für Dateien) aus `stdio.h` zu überprüfen. Schlussendlich lassen wir uns anzeigen, ob `feof` gesetzt ist, geben unseren Speicher

wieder frei und schließen die Datei. Da wir gerade in die Datei geschrieben haben, ist der Lesecursor natürlich nicht an ihrem Ende.
Im Beispiel `06_03_test_fread` versuchen wir die soeben geschriebenen Daten auszulesen und anzuzeigen. Zunächst benötigen wir wieder einen Speicherbereich, um die zu lesenden Daten zu fassen. Dann öffnen wir die Datei im `rb`-Modus und lesen die Daten mit `fread` ein, wobei wir wieder die Anzahl gelesener Blöcke überprüfen und `ferror` auslesen. Wie so oft in C müssen wir höllisch aufpassen, dass `fread` nicht einfach so in irgendwelchen Speicher reinschreibt. Schlussendlich geben wir die gelesenen Daten auf dem Bildschirm aus.
Noch ein Kommentar zur Geschwindigkeitsoptimierung von Schreib- und Leseroutinen: wählt man eine höhere Blockgröße, ist das Lesen und Schreiben wesentlich effizienter. Im Idealfall versucht man also die Menge zu lesender/schreibender Daten zu berechnen und diese in möglichst große Blöcke aufzuteilen, wobei man zur Optimierung `fread` / `fwrite` zwei Mal aufruft: ein Mal mit einer optimalen Blockgröße um den Großteil der Daten zu schreiben oder zu lesen und ein zweites Mal, um sich um den Rest zu kümmern.
In Beispiel `06_04/06_04_test_fwrite.c` versuchen wir Schreiben und Lesen zusammengesetzter Datenstrukturen anstelle von elementaren Datentypen. Wir bedienen uns dabei der Datenstruktur `teilchen_2d_t`, die wir in einer vorherigen Vorlesung entwickelt haben. Zunächst erstellen wir ein Array mit vier Elementen, initialisieren dieses und lassen uns das dritte Element ausgeben.
Unter Zuhilfenahme von `sizeof`-Funktion können wir die Blockgröße bestimmen und das Array von `teilchen_2d_t` in die Ausgabedatei schreiben. Wir überprüfen wieder alle möglichen Fehlerbedingungen und versuchen dann in `06_04/06_04_test_fread.c` die geschriebene Datei auszulesen.

6.5 systemabhängigkeit von `fwrite` / `fread`

An dieser Stelle sollten wir eine Komplikation der binären Ein- und Ausgabe besprechen. Einerseits müssen wir damit leben, dass Binärdateien von der *Endianess* der gentutzen Rechnerarchitektur abhängen. Wenn wir eine Zahl im Binärsystem darstellen, ist es bloß Konvention, ob wir das größte Bit an den Anfang oder ans Ende schreiben. Bei einer 8-bit Zahl könnten wir schreiben:

$$a_0 \cdot 2^7 + a_1 \cdot 2^6 + a_2 \cdot 2^5 + a_3 \cdot 2^4 + a_4 \cdot 2^3 + a_5 \cdot 2^2 + a_6 \cdot 2^1 + a_7 \cdot 2^0 \,,$$

oder aber,

$$a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + a_3 \cdot 2^3 + a_4 \cdot 2^4 + a_5 \cdot 2^5 + a_6 \cdot 2^6 + a_7 \cdot 2^7 \,.$$

Ersteres nennt man *little-endian*, weil das kleinste Bit rechts steht, letzteres nennt man *big-endian*, weil das größte Bit rechts steht. Wird eine Binärdatei, die auf dem einen System geschrieben wurde auf dem anderen gelesen, muss man danach die Bitreihenfolge umdrehen. Dafür muss natürlich bekannt sein, welche Datentypen gerade gelesen wurden.
Eine weitere Komplikation bei zusammengesetzten Datenstrukturen ist das sogenannte *Padding*. Um den Speicherzugriff zu optimieren, werden, je nach Architektur, zwischen den Elementen eines `struct` Nullstellen im Speicher eingefügt. Im Beispiel `06_04/06_04_sizeof_struct.c` wird dies veranschaulicht. Wir haben ein `struct`, welches ein `char` (1 Byte), ein `int` (4 Bytes) und ein `double` (8 Bytes) enthält. Wir würden also erwarten, dass `struct test_t` eien Größe von 13 Bytes hat. Lassen wir uns die Größe ausgeben, sehen wir jedoch, dass diese Datenstruktur in der Regel eine Größe von 16 bytes haben wird.

Wir könnten uns auch noch die Speicheradressen der einzelnen Elemente ausgeben lassen, um zu sehen, wo das Padding eingeführt wurde. Am wahrscheinlichsten ist es, dass hinter das `char` 3 Bytes eingefügt wurden.

6.6 Dateien Byte-weise auslesen / schreiben

Wir demonstrieren jetzt anhand des Beispiels `06_05_test_fgetc_fputc.c`, wie man Dateien Byte-weise lesen und schreiben kann. Dazu fertigen wir eine (leicht veränderte) Kopie der Datei `sw.txt` an.
Die Funktion `fgetc` gibt den Wert des gelesen Bytes als Integer zurück. Ist das Ende der Datei erreicht oder tritt ein Fehler auf, gibt sie stattdessen -1 zurück. Wir schreiben also eine `while`-Schleife und weisen den Rückgabewert der Variablen `in` zu und brechen aus der Schleife aus, falls der Rückgabewert -1 ist.
Innerhalb der Schleife geben wir das eingelesene Zeichen mithilfe von `putchar` direkt wieder in die Konsole aus. Sollte das gelesene Zeichen ein Zeilenumbruch sein, geben wir noch zwei weitere Zeilenumbrüche aus und warten eine Sekunde. Handelt es sich nicht um einen Zeilenumbruch, schreiben wir das Zeichen mit `fputc` in die Ausgabedatei.
Schlussendlich überprüfen wir noch, ob Fehler aufgetreten sind.

6.7 Den Dateicursor bewegen mit `fseek`

Die Funktion `fseek` nutzt man, um den Dateicursor selbständig zu bewegen. Dabei wird sich relativ zu einem Referenzpunkt `origin` um eine Anzahl `offset` Bytes bewegt. Es gibt drei spezielle Werte für `origin`, die in `stdio.h` als Präprozessorkonstanten `SEEK_SET`, `SEEK_CUR` und `SEEK_END` definiert sind. Setzt man `origin = SEEK_SET`, gilt das Offset ab Dateianfang während `origin = SEEK_END` das Offset ab Dateiende gezählt wird (`offset` kann also auch negativ sein). Schlussendlich bedeutet `origin = SEEK_CUR`, dass das Offset ab der momentanen Position des Dateicursors gilt.
In Beispiel `06_06_test_fseek.c` wird die Verwendung von `fseek` veranschaulicht.

6.8 Zeilenweise Datei auslesen: `getline`

Zum Abschluss dieser Vorlesungen möchten wir uns noch eine relativ sichere Alternative zu den ganzen Lesefunktionen anschauen. Der sogenannte POSIX-Standard legt für UNIX-Systeme Funktionalität fest, welche diese haben müssen, wenn sie die Kriterien für eine bestimmte POSIX-Version erfüllen möchten. In der 2008er Version des Standards wurde die Funktion `getline` eingeführt, die ein recht komfortables Arbeiten mit Textdateien erlaubt.
Um diese Funktion nutzen zu können, müssen wir die Präprozessorkonstante `_POSIX_C_SOURCE` auf den Wert `200809L` setzen. Wie in Beispiel `06_07_getline.c` gezeigt, brauchen wir zunächst einen Zeiger auf einen Puffer, den wir erstmal auf den Wert `NULL` setzen. Wir öffnen jetzt hintereinander mehrere Dateien unterschiedlicher Größe, lesen diese Zeilenweise aus und geben sie auf dem Bilschirm aus.
Erreicht `getline` das Ende einer Datei, gibt die Funktion -1 zurück, wir können dies also als Abbruchkriterium für unsere Schleife nutzen. Ansonsten gibt uns `getline` zurück, wie viele Zeichen gelsen wurden und schreibt ins zweite Argument die Länge des Pufferspeichers, den die Funktion zum Lesen der Zeile allokiert hat.
`getline` kümmert sich selbständig um den Pufferspeicher, wir müssen jedoch nach dem letzten Aufruf, egal, ob dieser erfolgreich war oder nicht, den Puffer selbständig mit `free` wieder freigeben.

7 Vorlesung 7

7.1 Algorithmische Komplexität und *Big-O*-Notation

In der heutigen Vorlesung wollen wir von technischen Details ein wenig Abstand nehmen und uns auf die Entwicklung von Algorithmen zurückbesinnen. Zunächst werden wir dazu die sogenannte *algorithmische Komplexität* kennenlernen.
Mit der Komplexität eines Algorithmus bezeichnen wir die Anzahl Rechenschritte und Speichermenge, die ein Algorithmus benötigt, um für eine bestimme Zahl an Elementen n, eine bestimme Aufgabe zu erfüllen. Man unterscheidet hierbei zwischen *Zeitkomplexität* und *Speicherkomplexität*.
Die Komplezität beziffern zu können erlaubt es, verschiedene Algorithmen zur Lösung des gleichen Problems miteinander vergleichen zu können, um den besten Algorithmus zu wählen. Allgemein werden diese Vergleiche im Limes von $n \to$ "sehr groß" angestellt, man spricht daher von *asymptotischer Zeitkomplexität*.
Jeder Rechenschritt in einem Algorithmus benötigt eine gewisse Zeit t_i und diese Rechenschritte werden im Regelfall in einer gewissen Häufigkeit wiederholt. Die Anzahl Wiederholungen kann dabei vom gegenwärtigen Zustand des Rechenproblems abhängen. Wenn eine Liste, zum Beispiel, schon sortiert ist, wird ein Sortieralgorithmus natürlich nur ein Mal durch die Liste iterieren und feststellen, dass diese schon sortiert ist.
Um die Zeitkomplexität eines Algorithmus zu verstehen wird zunächst oft vom *worst case* also dem schlimmsten Fall ausgegangen, es werden aber auch der *best case* und *average case* berechnet, wobei insbesondere der durchschnittliche Fall stark von den genutzten Daten abhängt (z.B.: wie "unsortiert" sind die Datem im Regelfall?).
Wir werden auf die Speicherkomplexität nicht im Detail eingehen, man sollte sich aber immer vor Augen führen, wie viel Speicher eine bestimmte Implementierung eines Algorithmus eigentlich benötigen wird und ob dies den verfügbaren Speicher des Rechners nicht übersteigt. Oft gilt in der Computerwissenschaft, dass Speicher- und Rechenbedarf vertauschbar sind: hat man, z.B., einen Algorithmus bei dem potentiell Zwischenergebnisse oft wiederverwedet werden können, dann kan man Rechenzeit sparen, indem ma man diese Zwischenergebnisse im Speicher ablegt und immer wieder darauf zugreift. Umgekehrt kann man, wenn notwendig, Speicher sparen, indem man Zwischenergebnisse immer wieder neu berechnet. Bei modernen Rechnerarchitekturen ist es oft so, dass mehr Rechenleistung als Speicherbandbreite verfügbar ist. Dies führt dazu, dass in einigen Fällen ein Algorithmus effizienter implementiert werden kann, wenn Zwischenergebnisse immer wieder neu berechnet werden anstatt die eingeschränkte Speicherbandbreite weiter zu beanspruchen.

7.2 Zeitkomplexität von Einfügensortieren

Um die Kosten von Einfügensortieren zu verstehen, weisen wir jeder Zeile eine Kostenfunktion zu. Wir beziffern dabei zunächst wie oft diese Zeile ausgefürt wird und multiplizieren dies mit einer unbekannten Konstante c_i. In der ersten Zeile unseres Algorithmus weisen wir der Schleife die Kosten $c_1 \cdot n$ zu, da bei jeder Iteration die Schleifenbedingung überprüft und eine Zählervariable inkrementiert werden muss. Die Zuweisungen in den nächsten beiden Zeilen schlagen jeweils mit Kosten $c_2 \cdot n$ und $c_3 \cdot n$ zu Buche.
In den Zeilen 4 bis 6 hängen die Kosten von der Datenlage ab und die innere Schleife in der vierten Zeile hängt von i ab. Wir beziffern diese Kosten also ganz allgemein mit einem Wert t_i für jede Iteration i, da die Bedingung in der fünften Zeile entweder wahr oder falsch sein kann. Jetzt können wir die Kosten aufsummieren und erhalten einen linear von n abhängigen Beitrag, sowie zwei Summen über i. Im schlimmsten Fall ist die Bedingung in Zeile 5 des Algorithmus immer erfüllt und wir haben $t_i = (i + 1)$, woraus sich eine Abnhängigkeit von n^2 ableiten lässt.

Im Limes $n \lim \infty$ dominieren diese beiden Terme ganz klar und wir sprechen von einem $\mathcal{O}(n^2)$-Algorithmus, einem "Ordnung n-Quadrat Algorithmus". Die Zeitkomplexität $T(n)$ ist von oben beschränkt durch eine Funktion $f(n^2)$ und es gibt ein k, das die Zeitkomplexität nach oben beschränkt.
Auch eine untere Schranke kann man finden: ist die Liste schon sortiert, wird aus der inneren Schleife sofort nach einer Iteration ausgebrochen. Es existiert also eine zweite Funktion $g(n)$ und eine Konstante ℓ, die $T(n)$ nach unten beschränken.

7.3 Zeitkomplexität von Mergesort

Im sechsten Übungszettel haben wir den Mergesort-Algorithmus implmentiert. Wir werden jetzt sehen, dass die Zeitkomplexität von Mergesort im Vergleich zum Einfügensortieren viel geringer ist.
Zunächst müssen wir wissen, wie viele Ebenen wir brauchen und behandeln zunächst mal nur den geraden Fall, im Beispiel in der Folie mit $n = 8$ Elementen: auf jeder Ebene wird die Elementenliste halbiert, wir haben also $\log_2 n$ Aufeilungen und insgesamt $(\log_2 n)+1$ Ebenen.
Jetzt lesen wir die Zeitkomplexität von *unten nach oben* ab, es lohnt sich an dieser Stelle auf die eigene Implementierung des Algorithmus zu blicken. Auf der rechten Seite der Grafik ist die Zahl der Gruppen auf jeder Ebene angegeben (die Notation ist etwas eigenartig, das gebe ich zu...).
Für 8 Elemente, müssen wir im ersten Schritt $8/2 = 4$ Vergleiche durchführen und, unter Umständen, $8/2 = 4$ mal vertauschen, also insgesamt 8 Rechenschritte. Auf der nächsten Ebene wissen wir, dass die Paare in sich schon sortiert sind, wir müssen die Elemente innerhalb der Paare also nicht mehr miteinander vergleichen, haben jetzt also pro Merge bis zu 4 Rechenschritte und führen 2 Merges durch. Im nächsten Schritt haben nur noch zwei Gruppen, die in sich sortiert sind. Die notwendigen Vergleiche und Vertauschungen belaufen sich auf maximal $2 \cdot 4$ Rechenschritte und es wird ein einziger Merge durchgeführt. Auf der letzten Ebene haben wir fast eine sortierte Liste und müssen die Elemente nochmal paarweise miteinander vergleichen und unter Umständen vertauschen.
Für den allgemeinen Fall haben wir also auf jeder Ebene *konstante* Kosten cn, wobei c eine beliebige Konstante ist und wir haben $(\log_2 n) + 1$ Ebenen, also Kosten $T(n) = cn \cdot ([\log_2 n] + 1)$. Im Limes $n \lim \infty$ dominiert der $n \log_2 n$ Teil und es handelt sich um einen $\mathcal{O}(n \log_2 n)$-Algorithmus.
Der Trick bei dieser Kostenreduktion im Vergleich zu Einfügensortieren liegt darin, dass wir das Problem in viele kleine Unterprobleme aufgeteilt haben und zunächst diese Lösen (das paarweise Sortieren) und dann für jeden nächsthöheren Schritt eine bessere Ausgangssituation haben. Anstatt, dass wir n Mal n Rechenschritte machen müssen, reicht es, wenn wir $(\log_2 n) + 1$ Mal, n Rechenschritte machen. Man spricht auch von *divide and conquer*, also "Teilen und Erobern".

7.4 Zeitkomplexität von Mergesort ganz konkret

Bei unseren Sortieralgorithmen können wir jetzt von Kosten $T_1(n) = c_1 n^2$ für Einfügensortieren und $T_2(n) = c_2 n \log_2 n$ für Mergesort ausgehen und mal annehmen, dass die Rechenschritte und Vertauschungen bei Mergesort viel teurer sind als beim Einfügensortieren, wir nehmen also mal an $c_2 = 10 \cdot c_1$.
Selbst bei diesem gigantischen Unterschied un der Skalierungskonstante haben wir bei nur 10000 Elementen schon einen Faktor 10^2 Unterschied: T_2 ist 100 mal kleiner als T_1. Für noch größeres n wird dieser Unterschied natürlich immer riesiger.

Die Schlussfolgerung lautet, dass man immer einen besseren Algorithmus bevorzugen sollte, anstatt zu versuchen, einen schlechten Algorithmus durch Optimierungen zu verbessern. In der Regel wird der bessere Algorithmus auf lange Sicht immer gewinnen.

7.5 Kategorisierung von Algorithmen

Es gibt verschiedene Konventionen, anhand derer man die Zeitkomplexität von Algorithmen kategorisieren kann. Wir haben für Einfügensortieren und Mergesort die sogenannte *Big-O*-Notation genutzt. Hierbei wird die worst-case Laufzeit bestimmt und man sagt, dass der Algorithmus nie schlechter skalieren wird als diese obere Schranke.
In der Θ-Notation wird versucht sowohl obere als auch untere asymptotische Schranken zu bestimmen, was natürlich nicht immer möglich ist.
Schlussendlich wird in der Ω-Notation versucht eine untere Schranke zu finden, damit wenigstens verstanden werden kann, wie schlecht der Algorithmus mindestens skaliert.
Welche kategorisierung man verwendet hängt maßgeblich davon ab, welche der Schranken sich überhaupt berechnen lassen.

7.6 Komplexität von Operationen auf Datenstrukturen

Das Skalierverhalten eines Algorithmus hängt nicht nur vom Algorithmus selbst ab, sondern auch von den genutzten Datenstrukturen. Stellen wir uns vor, wir haben einen Algorithmus, bei dem regelmäßig Elemente in ein Array eingefügt oder aus einem Array entfernt werden müssen.
Wenn wir in ein bestehendes Array ein Element enfügen möchten, müssen wir zunächst den Speicher vergrößern. Sollte es keinen zusammenhängenden Speicherbereich geben, wird `realloc` eine Kopie des Arrays erstellen müssen, dies skaliert linear mit der Größe des Arrays, also $\mathcal{O}(n)$. Ab Index i, schlimmstenfalls also n Mal, müssen jetzt die Elemente verschoben werden, also $\mathcal{O}(n)$. Schlussendlich wird das Element eingefügt, dabei handelt es sich um eine Operation mit konstanten Kosten, also $\mathcal{O}(1)$. Insgesamt skaliert die Operation also mit $\mathcal{O}(n)$. Beim Entfernen eines Elements verhält es sich ähnlich.

7.7 Anwendung: Doppelt verkettete Liste

Die doppelt verkettete Liste ist eine Standarddatenstruktur der Computerwissenschaft, bei der Einfügen und Entfernen $\mathcal{O}(1)$-Operationen sind. Man spricht auch von *konstanter Zeitkomplexität*, also Kosten, die unabhängig von der Zahl Elemente sind.
Wir haben zunächst ein Wurzelelement, welches Zeiger auf das erste und letzte Element hat. Hat die Liste keine Elemente, sind die beiden Zeiger `first` und `last` natürlich `NULL`. Abzweigend von diesem Wurzelelement haben wir die eigentlichen Datenelemente, die sogenannten `Nodes` oder `Knoten`. Diese haben zunächst Speicherplatz für die eigentlichen Daten (welcher Art auch immer), sowie jeweils einen Zeiger auf das vorherige und das nächste Element. Der `prev`-Zeiger des Anfangsknotens wird auf `NULL` gesetzt, ebenso wie der `next`-Zeiger des Endknotens.

7.8 Doppelt verkettete Liste: Listenelemente

Wir zeigen hier eine Headerdatei für eine doppelt verkettete Liste (*doubly-linked list, [DLL]*) zum Speichern von `double`-Variablen. Zunächst haben wir ein `struct`, das zwei Zeiger auf den Datentyp `struct dll_node_t` enthälgt, sowie Speicher für eine Gleitkommazahl, `double data`. In der Deklaration der beiden Zeiger müssen wir an dieser Stelle das `struct` noch explizit angeben, da das `typedef` an dieser Stelle noch nicht gegriffen hat.

Die nächste Datenstruktur, die wir definieren, ist das Wurzelelement, welches jeweils einen Zeiger auf das erste und das letzte Element hält. Da wir eine solche Liste dynamisch allokieren und deallokieren möchten, brauchen wir eine Funktion, die uns eine Liste erstellt, sowie eine Funktion, welche die Liste wieder freigibt. Schlussendlich brauchen wir natürlich noch Funktionen zum Einfügen und Entfernen von Elementen.

7.9 Doppelt verkettete Listen durchlaufen

In der 7. Übung wird eine solche DLL implementiert werden, an dieser Stelle wollenwir nur beispielhaft zeigen, wie man durch eine solche Liste iterieren würde. Man setzt einen Zeiger zunächst auf das erste (Vorwärtsdurchlauf) oder letzte Element und iteriert so lange, bis der Zeiger `NULL` ist. Inkrementiert wird durch `ptr = ptr->next` bzw. `ptr = ptr->prev`.
Es gibt natürlich noch etliche andere Datenstrukturen, die mithilfe von Zeigern durchlaufen werden, bzw. hierarchische Verhältnisse zwischen Datenelementen organisiseren.

7.10 Doppelt verkettete Liste: Nachteile

Der große Nachteil verketteter Listen ist, dass wir nicht einfach mal so auf das n-te Element zugreifen können (ausser, wir befinden uns gerade bei Element $n-1$). Um auf ein beliebiges Element n zuzugreifen, müssen wir bei `first` anfangen und die Liste bis n durchlaufen. Wenn wir sowieso alle Elemente lesen müssen, ist dies effizient (*sequentieller Zugriff*), wenn wir aber hin- und herspringen, hat der Zugriff Kosten $\mathcal{O}(n)$. In diesem Fall wäre ein Array die bessere Wahl, denn das hat konstante Kosten für springenden oder zufälligen Zugriff. Die verwendete Datenstruktur muss also immer auf den Algorithmus abgestimmt sein.

7.11 Funktionenpointer

Beim Programmieren versucht man, sofern möglich, generisch zu programmieren, damit man Code und Module möglichst oft wiederverwenden kann. Nehmen wir mal an, wir möchten eine Funktion implementieren, welche die numerische Ableitung einer anderen Funktion berechnen soll. Wir können natürlich nicht für jede beliebige Funktion eine solche Ableitungsfunktion implementieren, weshalb wir zu einem Funktionenzeiger, einem *function pointer* greifen werden.
Ein Funktionenzeiger ist ein "Datentyp", der für die komplette Signatur einer Funktion einstehen kann und dem die Adresse einer Funktion als Wert übergeben werden kann. Funktionen sind nichts weiteres als Zeiger auf einen ausführbaren Speicherbereich mit reserviertem Speicher für Argumente und einen Rückgabewert. Wenn eine Funktion aufgerufen wird, setzt der Compiler die Argumente entsprechend und der *Ausführungskontext* wechselt in den Scope der Funktion. Ist die Funktion beendet, wird der Rückgabewert gesetzt, der Ausführungskontext wechselt zurück zu dem Ort, an dem die Funktion aufgerufen wurde, und der Rückgabewert wird unter Umständen ausgelesen.

7.12 Funktionenpointer: Deklaration

Die Notation für einen Funktionenzeiger besteht aus einem Namen in Klammern, mit einem Sternchen versehen, also `(*fp)`. Der zweite Satz Klammern für die Datentypen von Argumenten markiert diesen Ausdruck als Funktionenzeiger und der Datentyp des Rückgabewerts ist der gleiche, wie jener der Funktion, auf die der Zeiger schlussendlich zeigen soll.

Haben wir jetzt eine Funktion `func`, die wir über den Funktionenzeiger aufrufen möchten, so schreiben wir einfach `fp = func;` oder `fb = &func;`. Diese beiden Schreibweisen sind identisch.
Will man die Funktion `func` jetzt über `fp` aufrufen, schreibt man bei zwei Argumenten z.B. `(*fp)(1, 2)`.

7.13 Funktionenpointer: Anwendung

Über einen Funktionenpointer können wir jetzt z.B. unsere Ableitungsfunktion verallgemeinern, indem wir als letztes Argument einen Zeiger auf eine Funktion übergeben, dessen Ableitung wir berechnen möchten.
Funktionenzeiger können auch auf Gleichheit überprüft werden und wir können, z.B., beim Tangens das Verhalten der Ableitung anpassen, um die problematischen Werte von x vorsichtig zu behandeln.

7.14 Funktionenpointer

Hier sehen wir noch weitere Beispiele für mögliche Funktionssignaturen und Funktionenzeiger auf diese Funktionen. Wir haben zwei Funktionen mit identischer Signatur, welche eine zusammengesetzte Datenstruktur zurückgeben, sowie einen passenden Funktionenzeiger dafür.
Wir haben darüberhinaus eine Funktion, die eine `double`-Variable zurückgibt, aber keine Argumente erhält, sowie einen dafür passenden Funktionenzeiger.
Leider sind Funktionenzeiger, wie so vieles in C, unsicher. Wir können dem Funktionenzeiger `fp`, welcher eigentlich für eine Funktion mit fünf Argumenten gedacht ist, auch die Adresse der `void_funktion` zuweisen. Der Compiler warnt hier zwar, es entsteht jedoch kein Kompilierfehler.
Das verrückte Verhalten, das daraus resultieren kann, wird in Beispiel `07_01_void_fpointer.c` dokumentiert.

7.15 Komplexe Zahlen

Keine Kommentare.

8 Vorlesung 8

8.1 Quicksort aus der C-Standardbibliothek

Wir werden heute unser Wissen zu Funktionenzeigern nutzen, um den sogennanten *Quicksort*-Algorithmus aus der C-Standardbibliothek zu verwenden. Diese Funktion ist eine gute Schablone dafür, wie in C generische (also vom Datentyp unabhängige) Funktionalität implementiert wird, außerdem ist Quicksort ein sehr guter Sortieralgorithmus, der insbesondere im Regelfall sehr schnell ist.
Die `qsort`-Funktion aus der `stdlib.h` erhält als erstes Argument einen `void`-Zeiger auf die zu sortierenden Daten. Das zweite Argument gibt an, wie viele Datenelemente denn im Speicherbereich liegen, auf den der Zeiger im ersten Argument zeigt. Das dritte Argument ist die Größe in Bytes der zu sortierenden Datenelemente, hier wird in der Regel `sizeof(datentyp)` übergeben.
Das letzte Argument ist ein Funktionenzeiger auf eine Funktion mit `int`-Rückgabewert und zwei Argumenten, bei denen es sich um `void`-Zeiger auf konstante Daten handelt (`const` *vor* dem Sternchen). Diese Funktion gilt es jetzt für jeden Datentyp, den wir sortieren möchten, zu implementieren. Es ist diese Funktion, die es `qsort` dann erlaubt, eine Sortierfolge für unsere Daten zu finden. Sie werden in der achten Übung selbst auch noch `qsort` ausprobieren, wir wollen an dieser Stelle aber mal ein komplexes Beispiel im Detail nachvollziehen.
Wir haben in der vierten Vorlesung einen zusammengesetzten Datentypen für die Position eines Teilchens, `pos_st`, entwickelt und dann auf den dreidimensionalen Fall ausgeweitet. Wir stellen uns jetzt mal vor, dass wir eine Menge solcher Teilchen haben und jetzt, z.B., nach der Distanz zum Ursprung oder der Distanz zur XY-Ebene, sortieren möchten.
Schauen wir zunächst mal in `08_01/pos_st.h`, dort sehen wir die beiden Funktionen `pos_compar_urspr_dist` und `pos_compar_xy_ebene_dist`, welche einen `int`-Rückgaberwert haben und zwei `void`-Zeiger als Argumente erhalten. In `08_01/pos_st.c` findet sich die Implementierung dieser Funktionen und wir sehen, dass wir die `void`-Zeiger auf Zeiger des Typs `pos_t const*` typecasten, bevor wir diese verwenden können.
In `08_01/08_01_quicksort.c` nutzen wir `qsort`, um die Teilchen erstmal nach ihrer Distanz zum Ursprung zu sortieren und dann nach ihrer Distanz zur XY-Ebene. Dabei übergeben wir einen Zeiger auf unser Array von Positionen, mit einem expliziten Typecast auf `void*`, wie viele Teilchen wir eigentlich haben, die Größe unseres Datentypen und die Vergleichsfunktion.

8.2 Externe Bibliotheken

Wie schon in einer der vorherigen Vorlesungen erwähnt, sind die Mathe-funktionen aus `math.h` in einer "externen" Bibliothek enthalten, in der Regel findet sich diese unter: `/usr/lib/x86_64-linux-gnu/libm.a`. Wenn wir beim Verknüpfen eines Programms `-lm` übergeben, sucht der Compiler in den Standardbibliothekspfaden, bzw. in den Systembibliothekspfaden nach der Bibliothek `libm.a`. Die Endung ist dabei architekturspezifisch und es gibt auch sogenannte dynamische Bibliotheken, auch *dynamically loaded library* (DLL) gennant. Auf UNIX-systemen werden solche Bibliotheken mit der Dateiendung `.so` versehen, Windowsnutzende werden aber sicherlich die Endung `.dll` wiedererkennen.
Der Compiler macht aus `-lm` also `libm.a` oder `libm.so` und sucht dann in den Systempfaden nach diese Bibliothek. Im allgemeinen Fall kann es aber sein, dass Bibliotheken eben nicht in den Systempfaden liegen. Hierzu wird dann beim Verknüpfen (Linken) des

Programms `-Lpfad` übergeben, wobei `pfad` z.B. `/usr/lib/x86_64-linux-gnu/` sein könnte.

8.3 Zufallszahlen mit GSL

Wir hatten in der fünften Übung einen eigenen Pseudozufallszahlengenerator (PRNG)[3] entwickelt und in einigen Beispielen auch die C-interne Funktion `rand()` genutzt. Die Qualität dieser Zufallszahlen lässt leider zu Wünschen übrig, da die Zahlen wesentlich weniger zufällig sind, als notwendig. Je nach Dimensionalität des Raums, für den man die Zufallszahlen nutzt, kann man sogar klare Strukturen in diesen "Zufallszahlen" erkennen. Anhand des hervorragenden Zufallszahlengenerators aus der GNU Scientific Library (GSL) werden wir zeigen, wie man externe Bibliotheken nutzt. Die GSL bietet ein ganzes Sammelsurium mächtiger Werkzeuge für wissentschaftliche Tätigkeiten, jedoch sind die Interfaces teilweise etwas komplex. Der Zufallszahlengenerator jedoch, hat ein sehr einfaches Interface.
Wir müssen zunächst die Funktionsdeklarationen aus der `gsl/gsl_rng.h` einbinden, diese liegt auf meinem System unter `/usr/include/gsl/gsl_rng.h`, kann aber auch ganz woanders sein. Um zu verstehen, wie wir die Bibliotheksfunktionen nutzen könnten, lohnt sich ein Blick in die recht gute Dokumentation: `https://www.gnu.org/software/gsl/doc/html/rng.html`, aber auch ein Blick in die Headerdatei `gsl_rng.h` ist hilfreich.
Um den Zufallszahlengenerator zu nutzen, benötigen wir einen Zeiger auf den Datentyp `const gsl_rng_type`, welcher den Typ des Generators bestimmen wird, deswegen schreiben wir `const` *vor* Sternchen, `gsl_rng_type const * generator_type` wäre identisch. Desweiteren benötigen wir einen Zeiger für den Zufallszahlengenerator an sich, dieser hat den Typ `gsl_rng*`. Bevor wir den GSL Zufallszahlengenerator nutzen können, müssen wir dessen Umgebung mit der Funktion `gsl_rng_env_setup()` initialisieren.
Jetzt setzen wir den Typ des Zufallszahlengenerators auf `gsl_rng_ranlxd2`, dabei handelt es sich um einen sehr hochwertigen PRNG[4]. Wenn wir in die Headerdatei `gsl_rng.h` blicken, sehen wir, dass die GSL sehr viele verschiedene Typen von Zufallszahlengeneratoren mitliefert. Jetzt allokieren wir mit diesem Typen unseren eigentlichen Zufallszahlengenerator und weisen den zurückgegebenen Zeiger `r` zu und initialisieren diesen mit dem *Seed* `12345`.[5]
Ab jetzt können wir die Funktion `gsl_rng_uniform` mit Argument `r` nutzen, um gleichverteilte Zufallszahlen im Intervall $[0, 1]$ zu generieren.

8.4 Kompilieren mit externen Bibliotheken

Da Headerdateien externer Bibliotheken nicht immer in den Standardpfaden zu finden sind, müssen wir im /Kompilierschritt| alle Pfade angeben, in denen der Compiler nach Headerdateien suchen sollte. Dies geschieht mit `-Ipfad`. Ebenso müssen wir beim Verknüpfen alle möglichen Bibliothekspfade angeben, in denen der Compiler die Bibliotheken finden kann, die wir verwenden mpchetn.
GSL liefert (standardmäßig) auch Funktionen für lineare Algebra mit, diese liegen in der Bibliothek `libgslcblas` und müssen auch verlinkt werden, wenn man mit GSL kompiliert.

[3] pseudo-random number generator
[4] `https://arxiv.org/abs/hep-lat/9309020`
[5] Wenn ein Pseudozufallszahlengenerator mit einem bestimmten Seed initialisiert wird, generiert er *immer* die gleiche Zahlensequenz. Dies garantiert, dass ein Code mit Zufallszahlen reproduzierbar ist, bedeutet aber auch, dass man unterschiedliche Seeds wählen muss, wenn man aus verschiedenen Runs eben nicht die gleichen Ergebnisse haben möchte.

8.5 Einfache Makefiles

Wir haben in den Beispielen schon Makefiles genutzt, ohne wirklich zu beschreiben, was da eigentlich vor sich geht. Ein Makefile ist eine Datei, die vom Programm `make` gelesen und ausgeführt wird. Sie beschreibt Abhängigkeiten zwischen beliebigen Dateien und Regeln, wie diese Abhängigkeiten zu erfüllen sind. Man kann Makfiles dazu benutzen, Programme zu kompilieren, man könnte aber ebenfalls die Erstellung einer PDF aus LaTeX-Dateien mit Makefiles aufsetzen[6].
Wichtig: In der zweiten Zeile des auf Folie 118 gezeigten Beispiels stehen keine Leerzeichen am Anfang der Zeile, sondern echte *Tabulatoren*! Nur so kann `make` die Datei richtig ausführen. In diesem Beispiel wird ganz generisch ein *target* definiert (hier `regel` genannt), das von den Dateien `abhaengigkeit1`, `abhaengigkeit2` und `abhaengigkeit3` abhängt. Um, nachdem diese Abhängigkeiten erfüllt sind, das target `regel` zu erstellen, wird `make` die Kommandos `kommando1` und `kommando2` nacheinander ausführen.
`make` überprüft dabei die Erstellungs- bzw. Veränderungszeiten der besagten Dateien: wenn eine Abhängigkeit neuer ist, als das Ziel, dann wird das Ziel neu erzeugt.

8.6 Makfiles: Beispiel 0

In unserem ersten Beispiel haben wir ein Programm mit zwei Modulen und einer Datei mit einer `main`-Funktion. Wir schreiben also zwei Regeln, um die beiden Module zu kompilieren, sowie eine Regel um das `main`-Module zu kompilieren. Letzteres hängt auch von den Headerdateien ab, da ja die Funktionsdeklarationen der Funktionen im Hauptmodul genutzt werden und wenn diese sich ändern, muss auch das Hauptmodule neu kompiliert werden. Schlussendlich wenden wir uns der ersten Regel zu, hier wird das Programm wird mit den Modulen verknüpft. Der Grund, wieso der letzte Schritt an erster Stelle steht, ist, dass wenn wir `make` aufrufen, dieses im gegenwärtigen Verzeichnis nach der `Makefile` sucht und dann die standardmäßig die *erste* Regel ausführt.
Wir könnten aber auch schreiben:

```
$ make modul1.o
$ make modul2.o
$ make programm.o
$ make programm
```

wenn wir die einzelnen Regeln per Hand ausführen möchten.
Jetzt ist eine solche `Makefile` natürlich nicht viel besser, als ein einfaches Shellskript, jedoch mit dem Vorteil, dass die Abhängigkeiten aufgefasst werden und nur jene Dateien neu kompiliert werden, die neu kompiliert werden müssen. Bei großen Projekten mit vielen hundert tausenden oder gar millionen Zeilen und entsprechend vielen Modulen, kann dies sehr viel Zeit sparen.
In diesen großen Projekten werden aber natürlich auch Makefiles geschrieben, die viele Verallgemeinerungen nutzen, um so ein Makefile generisch zu machen.

8.7 Makefiles: Beispiel 1

Im zweiten Beispiel nutzen wir die beiden Platzhalter `$@` und `$<`, die uns Zugriff auf den Namen des Targets und die erste Abhängigkeit geben, ohne, dass wir diese erneut tippen müssen.

[6] siehe auch die Dokumentation zu `latexmk` (`https://mg.readthedocs.io/latexmk.html`), ein Programm, das zusätzlich in einer Makefile genutzt werden kann, um große LaTeX-Projekte wie Abschlussarbeiten zu kompilieren

8.8 Makefiles: Beispiel 2

In diesem Beispiel gehen wir weiter: wir haben zwei Programme, die insgesamt von drei Modulen abhängen, jedes Programm hängt aber nur jeweils von zwei Modulen ab und die beiden Programme benötigen beide `modul2.o` Für die beiden Objektdateien, welche die `main`-Funktionen beinhalten, schreiben wir explizite Regeln, nutzen aber für die Modulobjekt eine generische Regel über den Prozentoperator. Hier bedeutet dies: nimm, was auch immer vor `.o` steht und setze es auf der rechten Seite jeweils dort wo auch ein Prozentzeichen steht, ein.
Wenn wir also jetzt die Regel für `programm1` ausführen, haben wir Abhängigkeiten von `programm1.o`, `modul1.o` und `modul2.o`. Für das erste haben wir eine explizite Regel, für die beiden Module konstruiert `make` selbständig die Regeln:

```
modul1.o: modul1.c modul1.h
      gcc -std=c99 -Wall -Wpedantic -c modul1.c

module2.o: modul2.c modul2.h
      gcc -std=c99 -Wall -Wpedantic -c modul2.c
```

genauso bei `programm2` und `modul3.o`.
Schlussendlich nutzen wir in den Regeln für die eigentlichen Programme den Platzhalter `$^`, um beim Verknüpfen gleich alle Abhängigkeiten aufzulisten.

8.9 Makefiles: Beispiel 3

Jetzt schauen wir uns ein relativ generisches Makefile an, welches ich für die Beispielprogramme nutze, sofern diese aus nur einer Datei bestehen. Zunächst definiere ich mir eine Variable, `SOURCES`, dieser werden alle `.c`-Dateien im gegenwärtigen Verzeichnis zugewiesen, was über denn `$(wildcard *.c)`-Befehl passiert.
In der nächsten Zeile wird eine Regel mit dem Namen `all` definiert, die von allen Programmnamen abhängt. Diese werden über den Befehl `basename` erstellt, welcher, z.B., `programm1.c` zu `programm1` macht.
Jetzt generieren wir uns Target für alle Programme durch ein geschachteltes Ziel mit *zwei* Doppelpunkten. zunächst werden die Basenames aus `$(SOURCES)` generiert und für jeden Eintrag, wird eine neue Regel erstellt, die schlussendlich in etwas wie

```
programm1: programm1.c Makefile
      gcc -g -ggdb -Wall -Wpedantic -std=c99 -o programm1 programm1.c
```

resultiert.
Zu guter Letzt wollen wir noch eine Regel zum aufräumen haben. Hierzu definieren wir uns ein weiteres Target, `clean`, welches keine Abhängigkeiten hat und einfach `rm` ausführt. Für dieses Ziel nutzen wir einen speziellen Modifkator: `.PHONY`. Targets, die in der `.PHONY` Liste sind, werden immer ausgeführt.

8.10 Makfiles: Beispiel 4

In den beiden Beispielen `makefiles/beispiel4` und `makefiles/beispiel5` finden sich Makefiles, die zwischen Compiler und Linker unterscheiden (dies sind auf manchen Architekturen verschiedene Programme) und Variablen für verschiedene Dateigruppen erstellen.
So wird zum Beispiel eine Variable mit allen Headerdateien generiert, sowie eine Variable, die alle Module enthält. Dafür wird der `patsubst`-Befehl genutzt, der aus einem String,

hier in der Variable `$(SOURCES)` gespeicher, alle `.c`-Dateien ausliest und durch `.o`-Dateien mit gleichen Basenames ersetzt.
Es kommen ferner die Befehle `filter-out` und `addsuffix` zum Einsatz und ein recht komplexes Konstrukt, bei dem Patterns doppelt expandiert werden (`.SECONDEXPANSION`).

8.11 Makefiles und Build-systeme

Große Softwarepakete bestehen meist aus tausenden Quelltextdateien. Für die einzelnen Module müsste man oft hunderte Makefiles schreiben und man muss zusätzlich darauf achten, dass die Software sich auf mehreren Architekturen übersetzen lässt. Der Aufwand, das Build-system (also die Makefilesammlung) zu pflegen wird dabei schnell so groß, dass es automatisierte Tools gibt, die einem die Erstellung und Pfelge eines Build-systems erleichtern.
Der *de facto* Industriestandard ist heutzutage `CMake`, ein Programm, das in einer eigenen Skriptsprache die Erstellung von *Meta-Makefiles* erlaubt, aus denen wiederrum architekturspezifische Makefiles erstellt werden.
Ein weiteres bekanntes Toolset zur automatisierten Erstellung von Makefiles sind die `Autotools`, deren `configure`, `make`-Zyklus denen bekannt sein wird, die schon mal UNIX-Software aus den Quellen kompiliert haben. Der erste Schritt dient dabei dazu, das Build-system und den Quelltext auf die gegenwärtige Architektur anzupassen.

8.12 Abbruchbedingungen mit *assertions*

Während der Entwicklung eines Programms ist es oft hilfreich viele Bedingungen abzufragen, um sicherzustellen, dass die Programmlogik richtig funktioniert. Idealerweise möchte man, dass bei einer Nichterfüllung einer dieser Bedingungen die genaue Stelle im Programm ausgegeben wird.
Ist das Programm jedoch getestet und wird für die Produktion von Forschungsdaten genutzt, kann es sein, dass die ganzen Tests das Programm dermaßen verlangsamen, dass man diese ganzen Tests wieder entfernen muss. Für Bedingungen dieser Art kann es praktisch sein, das `assert`-Makro zu nutzen, da dieses mit dem Kommandozeilenargument `-DNDEBUG` beim Kompilieren ausgeschaltet werden kann.
`assert` beendet die Ausführung des Programms, falls die Bedingung unwahr (also gleich 0) ist.

8.13 Zuverlässige Zeitmessung mit `gettimeofday`

Zum Abschluss dieser Vorlesung wollen wir uns noch ein überaus nützliches Konstrukt zur Zeitmessung ansehen. Es gibt zwar in der C-Standardbibliothek die Funktion `clock()`, diese ist aber für zuverlässige Zeitmessung ungeeignet[7], insbesondere in Programmen, die mehrere Rechenkerne gleichzeitig nutzen (nächste Vorlesung).
Mit der Funktion `gettimeofday` können wir mit einer Messgenauigkeit von Mikrosekunden Zeitintervalle bestimmen. In der nächsten Vorlesung werden wir uns ansehen, wie wir Programme durch Optimierung und Parallelisierung beschleunigen können, dabei werden wir öfter von `getttimeofday` Gebrauch machen.

[7] Es werden die "Ticks" der CPU gemessen, also die Zahl der Vibrationen des Piezo-elektrischen Taktgebers der CPU. Bei modernen CPUs verändert sich sogar die Takfrequenz ständig, `clock()` ist also noch ungeeigneter!

9 Vorlesung 9

Ich habe in den vorherigen Vorlesungen immer wieder betont, dass wir in der Physik möglichst hohe Ausführungsgeschwindigkeiten benötigen und wir deshalb die signifikaten Komplikationen von C und verwandten Sprachen in Kauf nehmen. Wir wissen schon, dass C-Compiler relativ schnellen Maschinencode produzieren, dies genügt aber in den meisten Fällen nicht. Zusätzlich müssen wir verstehen, welche Teile unserer Programme wir optimieren müssen, sogenannte *hotspots*, aber auch die Parallelisierung spielt eine immer wichtegere Rolle, da seit ca. 2005 Mehrkernrechner die Norm sind. Auf den meisten Supercomputern haben die einzelnen Rechenknoten einige Dutzend oder sogar hunderte Rechenkerne, die es auszulasten gilt.
Wir werden uns daher in dieser Vorlesung Optimierung und Parallelisierung anschauen, es wird jedoch nicht mögilch sein diese Themen auch nur ansatzweise in vollem Umfang zu behandeln. An einigen einfachen Beispielen werden wir die Philosophie veranschaulichen und verweisen dann auf weiterführende Literatur, sowie auf den *High performance computing*-Kurs (physics7505).

9.1 Der Aufbau einer CPU

Der Hauptprozessor (CPU) in einem modernen Rechner besteht meist aus mehreren Kernen (in der Abbildung sind schematisch vier gezeigt), sowie mehreren Speicherbereichen unterschiedlicher Größe und Geschwindigkeit. Die CPU ist an den Arbeitsspeicher angebunden, in dem die Speicherallokationen stattfinden. Wenn wir jetzt innerhalb unserer Programme auf Daten zugreifen, wird zunächst überprüft, ob diese Daten im Level 1 Cache (L1-Cache), einem sehr kleinen aber sehr schnellen Zwischenspeicher liegen. Ist dies nicht der Fall, wird der L2-Cache probiert, welcher größer, aber auch langsamer ist. Einige CPUs verfügen sogar über L3 und L4-Caches, die dann jeweils größer und langsamer sind als der L2 Cache. Wird die CPU in keinem der Caches fündig, werden die Daten direkt aus dem Arbeitsspeicher in die Register geladen. Bei den Registern handelt es sich um sehr schnelle Speicherstellen, auf die im Maschinencode direkt zugegriffen wird, im Beispiel auf Folie 14 sind die Register durch `%r` markiert.
Die Caches dienen dazu, die im Vergleich zur Rechengeschwidigket sehr niedrige Speicherbandbreite des Hauptspeichers zu verstecken: wenn auf Daten im Hauptspeicher zugegriffen wird, werden auch Daten um diesen Speicherbereich herum in die Caches geladen (während die Cores irgendwelche Berechnungen anstellen), in der Hoffnung, dass Zugriffe auf diese benachbartem Daten folgen werden. Die CPU versucht also vorherzusagen, welche Rechenschritte ein Programm tätigen wird. Ein Problem der Caches ist, dass sie immer synchron gehalten werden müssen, das heißt, die Daten in den Caches müssen immer mit den Daten im Hauptspeicher übereinstimmen, sonst kommt es zu Inkonsistenzen.
Um Speicherzugriffe zu optimieren, werden nicht einzelne Datenelemente aus dem Speicher geladen, sondern sogenannte *cache lines*, dabei handelt es sich um Speicherbereiche der Größenordnung von 64 Bytes (kleinere und größere cache lines sind jedoch je nach Architektur möglich). Dieser Umstand kann bei Programmen, die über mehrere Kerne verteilte parallele *Threads* nutzen (dazu später mehr) zum Problem werden: wird eine cache line ausgelesen, die Daten aus dieser Cache line aber über mehrere Kerne verteilt, muss die cache line als Ganzes synchron gehalten werden. Schreibt jetzt ein Kern einen neuen Wert in eines der Elemente, wird die gesamte cache line in allen Caches überschrieben werden und alle Kerne müssen die Daten erneut einlesen. Diese sogenannten *cache line conflicts* können Programme erheblich verlangsamen, bei der Parallelisierung muss daher darauf geachtet werden, dass die einzelnen Kerne möglichst auf weit auseinanderliegende Speicherbereihe zugreifen.

Die Ausführungsgeschwindigkeit eines Algorithmus hängt im Wesentlichen von drei Metriken ab und es hängt im Detail vom Algorithmus ab, welche dieser drei Metriken am wichtigsten ist.

- Die Anzahl Rechenschritte, die eine CPU pro Zeiteinheit ausführen kann, also die Taktrate der CPU (x GHz)
- Die verfügbare Bandbreite der verschiedenen Speicherherarchien, also des Hauptspeichers und der Caches, aber auch die Größe des Problems (wenn das gesamte Problem in den L2 Cache passt, ist die Ausführungsgeschwindigkeit natürlich höher, als wenn dauernd Daten aus dem Hauptspeicher nachgeladen werden müssen).
- Die Parallelisierbarkeit des Problems.

Wenn wir also Programme schreiben, die auf Ausführungsgeschwindigkeit ausgelegt sind,

9.2 Ein einfaches numerisches Problem

Wir werden die Optimierungen anhand der beiden grundlegenden Linearalgebraoperationen, der Matrix-vektor Multiplikation und der Vektornorm veranschaulichen. Zunächst müssen wir verstehen, wie viele Rechenschritte eigentlich (theoretisch) nötig sind, um diese Operationen durchzuführen.
Für jede Zeile der Matrix der Größe $Z \cdot S$, die mit dem Vektor multipliziert wird, müssen S Multiplikationen und (theoretisch) $S-1$ Additionen durchgeführt werden, was insgesamt $2S-1$ arithmetischen Operationen entspricht. Die gesamte Matrix-vektor Multiplikation benötigt also $Z \cdot (2S-1)$ Rechenschritte.
Die Normberechnung hingegen benötigt hingegen lediglich Z Multiplikationen und $Z-1$ Additionen.
Wir gehen von $Z-1$ Additionen aus, da im Idealfall zur Addition von x Zahlen, $x-1$ Additionszeichen notwendig sind, in der Praxis wird aber eine Addition mehr gemacht, welche wir jedoch nicht zur "nützlichen" Arbeit hinzuzählen.

9.3 Matrix-Vektor Multiplikation

Um auch den Bedarf an Speicherbandbreite zu verstehen, sehen wir uns die Operation weiter im Detail anhand einer Beispielimplementierung an. Wir haben pro Rechenschritt zwei *floating point operations* (FLOPs), und zwar eine Multiplikation und eine Addition (bzw. Akkumulation in diesem Fall). Pro Rechenschritt haben wir zwei LOAD-Operationen von jeweils 8 Bytes und eine STORE-Operation von 8 Bytes.
Die Datenelemnte von `x` werden mehrmals verwendet werden und wir können davon ausgehen, dass dieser Vektor, soweit es geht, in den Caches gehalten werden wird, was zu einer Effizienzsteigerung führt. Ebenso wird das Zielobjekt, also `y[i_z]` mit hoher Wahrscheinlichkeit über die gesamte Dauer der inneren Schleife in einem Register verbleiben.
Insgesamt haben wir einen Speicherbandbreitenbedarf von 32 Bytes pro Rechenschritt und einen Rechenbedarf von 2 FLOPs pro Rechenschritt, dies entspricht einer theoretischen Balance von 16 Bytes Bandbreitenbedarf pro 1 FLOP. Diese Balance nennt man auch die *arithmetische Intensität* einer Rechenoperation[8]. Je nach Rechnerarchitektur genügt die Speicherbandbreite auf einer gegenwärtigen CPU gerade mal um einen Datentransfer von 1 Byte pro getätigtem FLOP zu gewährleisten, die Matrix-Vektor Multiplikation wird also durch unsere Speicherbanbreite in ihrer Ausführungsgeschwindigkeit beschränkt sein, aber wie stark genau?

[8] Genauergesagt ist arithmetische Intensität definiert durch I = FLOPS/BYTES, wir drehen dies aber der Einfachheit halber um.

Wir wissen an dieser Stelle, dass `x[i_s]` und `y[i_z]` effizient aus Caches und Registern gelesen werden können und können daher raten, dass lediglich die 8 Bytes pro FLOP für die Matrixelemente unsere Ausführungsgeschwindigkeit maßegeblich beschränken werden. Daran können wir abschätzen, dass die Matrix-vektor Multiplikation etwa 8 Mal langsamger laufen wird, als auf Basis der Taktrate unserer CPU eigentlich zu erwarten ist (wenn alle Kerne ausgelastet werden, bei nur einem aktiven Kern steht diesem natürlich mehr Speicherbandbreite zur Verfügung).

9.4 Vektorrnorm

Für die Vektornorm können wir die gleiche Rechnung anstellen, hier wird aber lediglich `summe` in einem Register verbleiben, `x[i_s]` muss also wirklich aus dem Hauptspeicher gelesen werden und wir haben einen ähnlichen Banbreitenbedarf.
In Beispiel `09_01_plain` implementieren wir uns zunächst mal eine Stoppuhr, um mit `gettimeofday` Zeitintervalle vernünftig messen zu können, siehe `stopwatch.h` und `stopwatch.c`. Dazu benötigen wir einen Datentyp, der im von `gettimeofday` genutzten Format Anfangs- und Endzeit eines Intervalls speicher kann, sowie eine Funktion zum Starten und eine weitere Funktion zum Stoppen der Uhr. Schlussendlich noch eine Funktion, die uns das Zeitintervall in Sekunden zurückgibt.
Wir bauen uns auch eine Funktion `fatal_error`, die wir nutzen werden um bei Fehlerbedingungen das Programm beenden und eine Fehlermeldung ausgeben zu können. In diese Funktion ist das erste Argument eine Testbedingung, wenn diese also als 1 (wahr) übergeben wird, wird das Prgoramm beendet.
Für unsere Linearalgebra erstellen wir uns einen Datentyp für einen Vektor und einen Datentyp für eine Matrix, sowie Initialisierungs- und Freigabefunktionen für diese beiden Datentypen. Wir stellen desweiteren Funktionen bereit, die sowohl eine Matrix als auch einen Vektor mit Zufallszahlen besetzen können. Schlussendlich haben wir noch die beiden Funktionen `sq_norm` zur Berechnung der Vektornorm, sowie `mult_Mv` zur Multiplikation eines Vektors mit einer Matrix (von links).
In all unseren Funktionen nutzen wir das erste Argument von `fatal_error`, um verschiedene Bedingungen abzufragen, unter denen das Programm nicht korrekt funktionieren würde. In der Funktion `matrix_rand`, z.B., wird getestet, dass die Matrix richtig initialisert worden ist:

```
void matrix_rand( matrix_t * const m, gsl_rng const * rng ){
  fatal_error( (void*)(m->M) == NULL ||
               (void*)(m->mem) == NULL,
               "matrix_rand",
               "m contains NULL pointers..." );
  [...]
}
```

und bei der Matrix-Vektor Multiplikation stellen wir sicher, dass die beiden Vektoren, sowie die Matrix kompatibel miteinander sind.
Unsere Funktionen testen wir nun im Programm `matrix_benchmark`, welches vier Kommandozeilenargumente entgegennimmt.

```
 $ ./matrix_benchmark
usage:
matrix_benchmark <nrow> <ncol> <iterations> <random seed>

 $ ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 2.413417e+00
```

```
Gflop/s 4.344773e-01
square norm 3.194113e+07
```

Die Matrix-Vektor Multiplkation erreicht auf meinem Rechner bei einer Matrixgröße von $512 \cdot 512$ und 1000 Iterationen eine Performance von etwa 0.43 Milliarden FLOPs pro Sekunde (Gflop/s).
Wir können auch rumprobieren, wie die Ausführungsgeschwindigkeit variiert, wenn wir die Dimensionen der Matrix anpassen.

9.5 Nadelöhre verstehen mit `gprof`

Um zu verstehen, wo in unserem Programm die meiste Zeit verbracht wird, werden wir das Programm jetzt in einem sogenannten *Profiler* ausführen. Wenn wir dort Nadelöhre erkennen, wissen wir, wo wir die größten Anstrengungen aufbringen müssen, um unseren Code zu optimieren.
Dazu müssen zunächst die Kompilier- und Linkbefehle verändert werden: bei beiden muss das Argument `-pg` eingefügt werden. In der `Makefile` im Beispiel ist dies schon geschehen, es sollte jedoch erwähnt werden, dass das Profiling das Programm eventuell leicht verlangsamen kann. Wenn man zur Produktion übergeht sollte das `-pg` Argument wieder entfernt werden.
Nach erneutem Kompilieren können wir das Programm wieder ausführen, jetzt wird dieses eine Datei `gmon.out` ins gegenwärtige Verzeichnis schreiben, welche mit `gprof` wie in der Vorlesung gezeigt analysiert werden kann.

9.6 `gprof` Ausgabe

In der Ausgabe von `gprof` wird tabellarisch aufgelistet, wie viel Zeit relativ zueinander in den verschiedenen Unterfunktionen des Programms verbracht wurde. Wir sehen ganz klar, dass de Funktion `mult_Mv` die Laufzeit fast zu 100% dominiert.

9.7 Erste Optimierung: Compilerflaggen

Compiler können versuchen, verschiedene Optimierungen bei der Übersetzung von C zu Maschinencode vorzunehmen. Wir könnten zunächst das allgemeine Optimierungslevel auf 3 hochschrauben, indem wir das Kommandozeilenargument `-O3` an gcc übergeben. Auf meinem Rechner wird die Ausführungsgeschwindigkeit dadurch etwas mehr als verdoppelt:

```
 $ ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 7.311770e-01
Gflop/s 1.434092e+00
square norm 3.194113e+07
```

Caveat: Es könnte sein, dass diese Optimierungen in der Virtualbox-Umgebung gar nicht oder nur unzureichend funktionieren. Wenn man wirklich hohe Ausführungsgeschwindigkeit möchte, sollte man natürlich keine virtuelle Maschine nutzen, sondern das Betriebssystem nativ auf seinem Rechner ausführen. Bei modernen CPUs werden die Befehle aus einer virtuellen Maschine jedoch direkt an die CPU weitergeleitet, was bedeutet, dass in vielen Fällen die Performance identisch oder sehr ähnlich sein wird. Man muss jedoch beachten, dass von Virtualbox bloß ein virtueller Rechenkern bereitgestellt wird.
Zusätzlich können wir noch CPU-spezifische Optimierungen hinzuschalten, hier die richtigen Flaggen zu finden ist jedoch ein wenig mühselig. Zunächst müssen wir rausfinden,

welche CPU denn eigentlich verbaut ist in unserem Rechner. Dies tun wir, zumindest unter Linux, über den Befehle

```
$ cat /proc/cpuinfo
```

und erhalten dann für jeden CPU-Kern (bzw. Hyperthread) ein Sammelsurium an Informationen, interessant ist für uns jedoch erstmal nur `model name`. In meinem Rechner steckt z.B. ein `Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz`. Mit dieser Information kann ich mich nun auf `https://ark.intel.com` begeben und kann im dortigen Suchfeld `i7-6600U` eingeben. Dort interessiert mich vor allem der Codename für diesen CPU-Typ, es handelt such um eine CPU der "Skylake"-Generation.
Wenn wir jetzt in der Dokumentaion von `gcc` nachsehen (`man gcc`), sehen wir unter der Rubrik `x86 Options`, viele verschiedene Einstellungen für das Kommandozeilenargument `-march`, darunter, je nach Compilerversion, auch `skylake`.
Ich füge jetzt zu den Compileroptionen die Argumente `-march=skylake` und `-mtune=skylake` hinzu und sehe, dass die Performance sich noch ein wenig weiter verbessert:

```
 $ ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 6.455540e-01
Gflop/s 1.624303e+00
square norm 3.194113e+07
```

Im Vergleich zur Ausgangssituation haben wir uns fast um einen Faktor 4 verbessert!

9.8 Zweite Optimierung: Multi-threading mit OpenMP

Ich hatte zu Anfang der Vorlesung schon erwähnt, dass moderne CPUs viele Rechenkerne haben. Performancekritische Software muss also diese Rechenkerne auch wirklich ausnutzen, leider ist dies oft nicht besonders einfach, da Programmiersprachen immer noch nicht wirklich darauf ausgelegt sind.
Mit OpenMP werden sogenannte *Threads* genutzt, um Rechenaufgaben auf mehrere Kerne zu verteilen, in der Regel passiert dies über das sogenannte *fork-join*-Modell. Hier werden in speziell als parallelisierbar markierten Regionen rechenintesive Teile eines Programms auf mehrere Threads aufgeteilt. Ist die Arbeit getan, werden die Threads wieder zusammengeführt und es wird unter Umständen noch synchronisiert.
Beim Multi-threading gibt es zwei zentrale Caveats:

- Threads können sich gegenseitig in die Queere kommen: im Regelfall dürfen zwei verschiedene Threads nicht in die gleiche Speicherstelle schreiben, sonst kommt es zu sigenannten *race conditions*, also "Rennen" darum, welcher geschriebene Wert denn jetzt der aktuelle ist. Dies führt zu falschen Rechenergebnissen.
- Die Kosten für die Parallelisierung selbst sind nicht unerheblich und es muss oft abgewogen werden, ob ein Algorithmus überhaupt davon profitiert parallelisiert zu werden. Insbesondere bei Algorithmen, die viel Synchronisation zwischen Threads verlangen, werden die Vorteile schnell zunichte gemacht.

Wir werden auf die Feinheiten von OpenMP nicht im Detail eingehen können, wollen aber trotzdem anhand unserer Beispiele das Konzept zumindest vorstellen. Auf Folie 146 ist eine Liste weiterführender Links mit recht brauchbaren Tutorials.

9.9 OpenMP Blitz-Intro 1/2

Um unsere Matrix-vektor Multiplikation mit OpenMP zu parallelisieren teilen wir die Schleife über die Zeilen der Matrix auf mehrere Threads auf. Dies geschieht mit der Syntax `#pragma omp parallel for`, die sich auf die nachfolgende `for`-Schleife bezieht.
Wichtig: die einzelnen Iterationen der Schleife müssen hierbei voneinander unabhängig sein, da die Threads unabhängig voneinander arbeiten und nur auf den jeweils eigenen Schleifenteil zugreifen können. Im Gegenbeispiel auf Folie 139 werden, um das Ergebnis `x[i_z]` zu erzeugen, das Ergebnisse aus der Iteration `i_z-1` benötigt. Wenn die Schleife jedoch auf mehrere Threads aufgeteilt wurde, gibt es keine Garantie, dass eine beliebige Iteration `i_z-1` eines anderen Threads schon durchgelaufen ist, wenn ein Thread diese benötigt.

9.10 OpenMP Blitz-Intro 2/2

Ein weiteres Problem ergibt sich bei der Vektorrnorm. Wir hatten erwähnt, dass nur jweils ein Thread in einen Speicherbereich schreiben darf. Wenn wir jedoch die Vektorrnorm aufsummieren (akkumulieren), müssen wir in jeder Iteration den Wert von `summe` auslesen und den jeweiligen Beitrag der gegenwärtigen Iteration aufaddieren.
Für solche Fälle gibt es in OpenMP das Schlüsselwort `reduction(Operation,Variable)`, wobei wir `Operation` mit `+` ersetzen und `Variable` mit `summe`. Durch dieses Schlüsslwort wird für jeden Thread eine eigene Kopie der Variablen `summe` erzeugt, jeder Thread akkumuliert seinen Teil der `for`-Schleife in diese *thread-lokale* Variable und am Ende der Schleife werden diese einzelnen Zwischenergebnisse dann automatisch in `summe` aufaddiert.
Beim Kompilieren müssen wir beachten, dass wir je nach Compiler, das Kommandozeilenargument `-fopenmp` anhängen.
Testen wir also mal mit `09_03_plain_openmp`, was das alleinige Parallelisieren bringt. In meinem Rechner befinden sich zwei Rechenkerne, ich sollte also mit zwei Threads versuchen, dazu setze ich die Umgebungsvariable `OMP_NUM_THREADS=2`.

```
 $ OMP_NUM_THREADS=2 ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 1.175741e+00
Gflop/s 8.918418e-01
square norm 3.194113e+07
```

Wir sehen in der Tat eie Verbesserung um etwa einen Faktor 2. Die ausgegebene Quadratnorm zeigt uns auch, dass wir keinen Fehler eingeführt haben, da das Ergebnis identisch ist mit dem nicht-parallelisierten Fall.
Jetzt unterstützt die CPU in meinem Rechner jedoch eigenlithch 4 Hyperthreads, obwohl die CPU nur zwei Kerne hat (Threads zwischen denen relativ schnell hin- und hergeschaltet werden kann, wenn einer davon warten muss). Helfen diese zusätzlichen Threads?

```
$ OMP_NUM_THREADS=4 ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 1.425407e+00
Gflop/s 7.356320e-01
square norm 3.194113e+07
```

Nein! Das Programm wird sogar etwas langsamer, weil die Threads sich um die physikalischen Resourcen auf der CPU streiten und sich so gegenseitig ausbremsen. In manchen Fällen nützt es, wenn mehr Threads als physikalisch vorhandene Rechenkerne genutzt werden, in anderen schadet es eher.
Jetzt probieren wir noch die Kombination von Threads und Compilerflaggen mit:

```
gcc -g -std=c99 -Wall -Wpedantic -O3 -mtune=skylake -march=skylake -
mfma -fopenmp
```

und erhalten:

```
 $ OMP_NUM_THREADS=2 ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 3.393840e-01
Gflop/s 3.089642e+00
square norm 3.194113e+07
```

Wir haben also durch Compilerflaggen und OpenMP-Parallelisierung unser Programm insgesamt um etwa einen Faktor 7 beschleunigt und die Parallelisierung selbst hat uns etwa einen Faktor 1.9 gebracht.
Jetzt versuchen wir nochmal bei der optimierten Version des Programms, mehr Threads zu nutzen. Da die einzelnen Rechenschritte jetzt viel schneller sind, kann es sein, dass öfter auf den Speicher gewartet wird und dass 4 Threads die parallel auf den Speicher zugreifen eventuell mehr Speicherbandbreite erreichen würden. Wir sehen, dass dies in der Tat der Fall ist und stellen fest, dass das Programm jetzt ungefähr einen Faktor 12 schneller läuft, als in unserer Ausgangssitutation!

```
gcc -g -std=c99 -Wall -Wpedantic -O3 -mtune=skylake -march=skylake -
mfma -fopenmp
 $ OMP_NUM_THREADS=4 ./matrix_benchmark 512 512 1000 9890812
nrow= 512 ncol= 512 iters=1000 time= 1.934570e-01
Gflop/s 5.420197e+00
square norm 3.194113e+07
```

Wir können die Optimierungsmaßnahmen noch grafisch als Funktion der Matrizzenseitenlänge für quadtratische Matrizzen vergleichen. In Abb. 4 sehen wir, dass für die nicht-optimierten Fälle sowohl mit als auch ohne OpenMP die Performance mit der Matrizzengröße ansteigt. Für jene Fälle mit optimierten Compilerflaggen jedoch, fällt die Performance mit steigender Größe leicht ab. Für die schnellste Version, OpenMP mit Compilerflaggen, steigt die Performance mit zunehmender Matrizzengröße stark an und fällt dann wieder leicht ab.

9.11 Dritte Optimierung: Verteilung über mehrere Rechner mit MPI

Zur Verteilung einer Rechnung über die Kerne einer einzelnen CPU sind Threads eine gute Möglichkeit. Haben wir jedoch mehrere Rechner, auf die wir eine sehr anspruchsvolle Rechnung verteilen möchten, können wir keine Threads nutzen, da die Arbeitsspeicher der einzelnen Rechenknoten nicht miteinander verbunden sind. Wir müssen also über ein Netzwerk die einzelnen Knoten miteinander verbinden und explizit Daten austauschen. Dazu wird in den meisten wissentschaftlichen Programmen *MPI*, das *Message Passing Interface* genutzt. Wir werden in den folgenden Folien zwar nicht mehrere Rechner miteinander vernetzen, können aber die Kommunikation zwischen verschiedenen *ranks* bzw. Programmen auch auf einem Rechner ausprobieren.

9.12 MPI Blitz-Intro 1/3

Um MPI zu nutzen, müssen wir zunächst eine MPI-Version installieren. Auf Ubuntu- oder Debian-systemen können wir dafür die Pakete `openmpi-bin`, `openmpi-common`, `libopenmpi2` und `libopenmpi-dev` installieren.

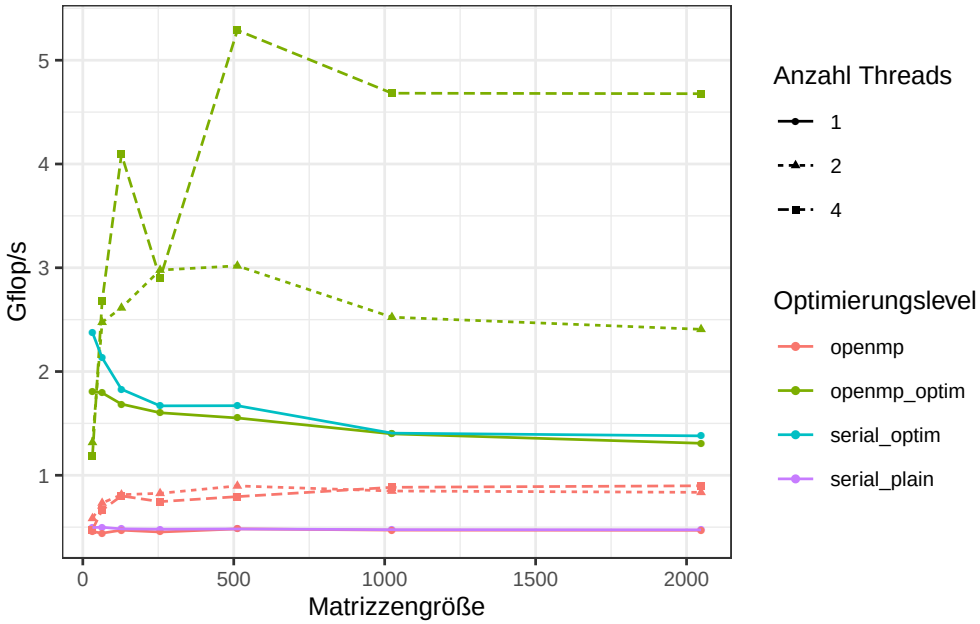

Abbildung 4: Das Skalierverhalten der Matrix-Vektor Multiplikation und Quadratnorm als Funktion der Seitenlänge für quadtratische Matrizzen ($N \in [32, 64, 128, 256, 512, 1024, 2048]$). Die Symbol- und Linientypen zeigen die Anzahl genutzter Threads (nur > 1 für den Fall, dass OpenMP genutzt wurde) und die Farben zeigen die Art der Optimierung. `serial` entspricht `09_01_plain`, `serial_optim` entspricht `09_02_compiler_flags`, `openmp` entspricht `09_03_plain_openmp` und `openmp_optim` entspricht `09_04_compiler_flags_openmp`.

Gezwungenermaßen werden wir sehr viele Details überspringen und zeigen nur das Notwendigste. In unserer `main`-Funktion müssen wir MPI zunächst initialisieren, dies geschieht über `MPI_Init`, einer Funktion, der wir die Kommandozeilenargumente auch übergeben müssen. (die Adressen, sowohl von `argc`, als auch von `argv`) Der sogenannte *Kommunikator* `MPI_COMM_WORLD` steht für die Menge *aller* Ranks[9] und wir erfragen zunächst, wie viele Ranks eigentlich gestartet wurden (`MPI_Comm_size`), dies lassen wir in die Variable `nranks` schreiben. In einem MPI-Programm werden alle Ranks gleichzeitig ausgeführt und laufen Schritt für Schritt die gleichen Programmzeilen ab. Um zu wissen, um welchen Rank es sich gerade handelt, schreiben wir die Identifikationsnummer des ranks in die Variable `myrank`. Jetzt geben wir diese Information am Bildschirm aus und beenden zunächst MPI mit `MPI_Finalize` und dann das Programm wie gewohnt, indem wir aus der `main`-Funktion einen Rückgabewert an das Betriebssystem zurückgeben.
Um das Programm zu kompilieren müssen mir `gcc` mit `mpicc` ersetzen. Der MPI-Compiler kümmert sich darum, dass alle für MPI notwendigen Bibliotheken eingebunden werden, während im Hintergrund `gcc` das eigentliche Kompilieren übernimmt.
Das Programm wird jetzt nicht einfach ausgeführt, sondern durch einen *Launcher* gestartet, in unserem Fall ist dies `mpirun`. Dort geben wir jetzt auch an, wie viele Ranks wir eigentlich starten möchten.

9.13 MPI Blitz-Intro 2/3

Jetzt haben wir MPI zwar gestartet, aber uns noch nicht wirklich überlegt, wie denn die Parallelisierung an sich vonstatten gehen soll. Zunächst müssen wir uns überlegen wie wir unser Problem vernünftig auf die verschiedenen Ranks verteilen, damit jedes Rank ein kleineres Unterproblem lösen kann. Innerhalb des Algorithmus können wir dann mit einer Vielzahl von Funktionen Daten zwischen Ranks austauschen, hier erwähnen wir ersmal nur `MPI_Send` und `MPI_Recv`.
Stellen wir uns vor, wir haben ein Problem auf einem diskreten, eindimensionalen Raum (also Punkten entlang einer Linie). Diese Punkte können wir also in N Intervalle teilen und diese dann verschiedenen Ranks zuweisen. Offensichtlich können wir jetzt von Nachbarranks sprechen, die sich links oder rechts von einem bestimmten Rank befinden. Bei der Initialisierung des MPI-Programms wird man in der Regel eine Liste der Nachbard erstellen und diese Rank IDs in irgendwelchen Variablen speichern, z.B. `nachbard_rechts` und `nachbar_links`.
Wollen wir jetzt Daten austauschen, geben wir bei `MPI_Send` eine Adresse in einem Vektor an, wo die Daten anfangen, die wir versenden möchten. Das zweite Argument ist die Anzahl Elemente und das dritte der Datentyp der Elemente, die wir versenden. MPI unterstützt nativ nur einige elementare Datentypen für die es Kennwörter wie `MPI_DOUBLE` für `double`-Werte definiert. Das vierte Argument von `MPI_Send` ist die Rank ID des Ranks, an den wir Daten verschicken möchten und das letzte Argument betrifft die Gruppe von Ranks, zu der diese Rank ID gehört (oft ist dies `MPI_COMM_WORLD` für alle Ranks). Beim fünften Argument handelt es sich um einen ganzzahligen Identifikator für den gegenwärtigen Kommunikationsprozess, mit dessen Hilfe man `MPI_Send` auf einem Rank mit einem `MPI_Recv` auf einem anderen Rank verknüpft.
Jedes `MPI_Send` benötigt als Pendant ein `MPI_Recv`, in dem die Daten eines Nachbarns entgegengenommen werden. Die Syntax unterscheidet sich nur dadurch, dass man natürlich hier einer Speicherstelle für den Datenempfang bereitstellen muss[10]. Wenn an

[9] Ganz genau gesehen sollte man `MPI_COMM_WORLD` eigentlich nicht benutzen sondern davon eine eigene Kopie erstellen, wir überspringen diesen Schritt der Einfachheit halber jedoch.
[10] Hier ist es natürlich wichtig, dass `MPI_Send` und `MPI_Recv` sich nicht in die Quere kommen: die Speicherbereiche müssen unabhängig sein.

den Nachbarn rechts von uns gesendet wird, muss natürlich vom Nachbarn links von uns empfangen werden. Das letzte Argument von `MPI_Recv` ist ein Zeiger auf eine Variable des Typs `MPI_Status`, einer Datenstruktur, die es uns erlaubt den Status abzufragen. Hier werden wir den Status nicht verwenden und übergeben einen Standardzeiger `MPI_STATUS_IGNORE`.
An dieser Stelle sei noch erwähnt, dass die meisten MPI-Funktionen einen `int`-Rückgaberwert haben, den man testen sollte, um zu erfahren, ob ein Fehler aufgetreten ist oder nicht, z.B. so:

```
int rval = MPI_Send(...);
if( rval != MPI_SUCCESS ){
  printf("Fehler in MPI_Send!\n");
  exit(rval);
}
```

Aus der MPI-Dokumentation für die möglichen Rückgabewerte:

- `MPI_SUCCESS`: No error; MPI routine completed successfully.
- `MPI_ERR_COMM`: Invalid communicator. A common error is to use a null communicator in a call (not even allowed in `MPI_Comm_rank`).
- `MPI_ERR_COMM`: Invalid communicator. A common error is to use a null communicator in a call (not even allowed in `MPI_Comm_rank`).
- `MPI_ERR_COUNT`: Invalid count argument. Count arguments must be non-negative; a count of zero is often valid.
- `MPI_ERR_TYPE`: Invalid datatype argument. Additionally, this error can occur if an uncommitted `MPI_Datatype` (see `MPI_Type_commit`) is used in a communication call.
- `MPI_ERR_TAG`: Invalid tag argument. Tags must be non-negative; tags in a receive (`MPI_Recv`, `MPI_Irecv`, `MPI_Sendrecv`, etc.) may also be `MPI_ANY_TAG`. The largest tag value is available through the the attribute `MPI_TAG_UB`.
- `MPI_ERR_RANK`: Invalid source or destination rank. Ranks must be between zero and the size of the communicator minus one; ranks in a receive (`MPI_Recv`, `MPI_Irecv`, `MPI_Sendrecv`, etc.) may also be `MPI_ANY_SOURCE`.

9.14 MPI Blitz-Intro 3/3

Im fünften Übungszettel haben wir ein Game of Life programmiert, wenn man dieses einfache Beispiel jetzt mit MPI parallelisieren möchte, würden wir zunächst das Spielfeld entsprechend in einer oder sogar zwei Dimensionen auf die verschiedenen Ranks veteilen. Da im Game of Life die Werte der nächsten Nachbarn darüber entscheiden, wer lebt oder stirbt, müssen wir diese Werte natürlich über die Ranks hinaus verfügbar machen. In der Regel wird dazu ein Speicherbereich bereitgestellt, ein sogenanntes *Halo*, in den dann nach jeder Iteration die Werte der Nachbarpunkte per Netzwerk kopiert werden.
Da wir im Game of Life periodische Randbedingungen nutzen, geht auch die Kommunikation in alle Richtungen, man spricht hier auch von einem zweidimensionalen Torus. Wie in der Abbildung zu sehen ist der Speicherbedarf für dan Halospeicher in diesem Fall größer, als der Bedarf für die jeweiligen lokalen Spielfelder, umso größer jedoch das lokale Spielfeld, umso kleiner wird der Halospeicherbedarf relativ dazu. Auch die Kommunikationskosten können so hoch sein, dass sich die Parallelisierung, je nach Algorithmus, gar nicht auszahlt.

Wir werden ein Beispiel der Komplexität eines parallelen Game of Life nicht in der Vorlesung behandeln, ich hoffe aber, dass dies einem zumindest eine Idee gibt, wie parallele Algorithmen in der Praxis implementiert werden und welche Kompromisse zu treffen sind, um dies effizient zu tun.

9.15 MPI Matrix-Vektor Produkt & Vektornorm

Bei unserem Matrix-Vektor Produkt werden wir einen recht einfachen Weg wählen und ausschließlich in der Zeilendimension der Matrix parallelisieren. Wir werden an einer Stelle fuschen: eigentlich müssten wir sicherstellen, dass die Matrixelemente unserer Matrix mit denen des seriellen (nicht-parallelisierten) Programms übereinstimmen, dazu müssten wir den Zustand des Zufallszahlengenerators von einem Rank zum nächsten geben und die Matrixelemente so seriell initialisieren. (oder einen parallelisierbaren Zufallszahlengenerator nutzen) Das werden wir nicht tun, stattdessen wird Rank 0 alle Zufallszahlen generieren und wir kommunizieren dann die Vektoren mithilfe der `MPI_Bcast`-Funktion und die entsprechenden Teile der Matrix mit der `MPI_Scatter`-Funktion.
Schauen wir uns also das Beispiel `09_05_plain_mpi/matrix_benchmark.c` an. Zunächst werden in der `main`-Funktion die Rank IDs ausgegeben und zwar geordnet nach Rank ID. Dies gelingt durch `MPI_Barrier` in jeder Iteration. Ein *Barrier* ist ein Synchronisierungspunkt an dem alle Ranks, die sich in der übergebenen Gruppe befinden, warten, bis alle Ranks diesen Befehl ausgeführt haben. Wir wollen, dass alle Ranks dies tun, also übergeben wir `MPI_COMM_WORLD`.
Jetzt teilen wir die `nrow`-Dimension durch die `nranks`, um die lokale Matrizzengröße zu bestimmen. Wir initialisieren den Zufallszahlengenerator, werden ihn aber nur auf Rank 0 benutzen, trotzdem initialisieren wir ihn so, dass alle Ranks unabhängige Zufallszahlengeneratoren haben. In dieser Initialisierung steckt ein Fehler: der Seed sollte positiv bleiben, so wie wir das hier aber gelöst haben, kann es sein, dass der Seed für einige Ranks den Wertebereich von `long int` überläuft und ins Negative rutscht.
Wir reservieren Speicher für die ganzen lokalen Matrizzen, auf Rank 0 jedoch auch Speicher für eine große globale Matrix (die alle Matrixelemente initialisieren wird), desweiteren generiert Rank 0 an dieser Stelle auch alle Zufallszahlen, was bei großen Matrizzen natürlich lange dauert.
Auf allen Ranks ausser 0 wird `m_global` der Einfachheit halber mit einer $1 \cdot 1$ Matrix initialisiert. Auf die Initialisierungen folgt wieder ein `MPI_Barrier`, um sicherzustellen, dass alle Ranks ihren Speicher initilisiert haben.
Jetzt werden mit `MPI_Bcast` die Vektoren von Rank 0 auf alle anderen Ranks kopiert und mit `MPI_Scatter` wird die globale Matrix auf alle Ranks in der Zeilendimension verteilt. Die `MPI_Scatter`-Funktion nimmt dabei an, dass die Daten in einem zusammenhängenden Speicherbereich liegen.
Für kleine Matrizzen geben wir die Matrixelemente noch auf dem Bildschirm aus.
Auf allen Ranks starten wir Stoppuhren und iterieren über Matrix-Vektor Multiplikationen und berechnen die Quadratnorm des `y`-Vektors. Da es sich beim der Norm um eine globale Größe handelt (die von Daten auf allen Ranks abhängt), haben wir diese Funktion natürlich in `matrix.c` entsprechend angepasst. Wir haben ein Argument hinzugefügt, für den Fall, dass es sich um eine parallele Ausführung handelt und warten dann zunächst mit einem `MPI_Barrier`, um sicherzustellen, dass alle Ranks einen konsistenten Zustand erreicht haben.
Danach werden die lokalen Vektorelemente quadratisch aufsummiert und im parallelen Fall wird dann auch die Summe über alle Ranks gebildet, dies passiert mit der Funktion `MPI_Allreduce`, nach deren Ausführung die Variable `ret_reduce` auf allen Ranks den gleihen Wert enthält.

Bei einer globalen Summe handelt es sich um eine sogenannte *Reduktion* des Typs `MPI_SUM`. Das dritte Argument von `MPI_Allreduce` beziffert die Anzahl von Datenelementen, die (jeweils) aufsummiert werden sollen.
Führen wir dieses Programm mit 2 Ranks aus, ist die Performance etwas höher, als mit OpenMP.

```
 $ mpirun -np 2 ./matrix_benchmark 512 512 1000 9890812
This is MPI rank with id=0 out of 2 ranks
This is MPI rank with id=1 out of 2 ranks
nrow= 512 ncol= 512 iters= 1000 time= 1.140300e+00
Gflop/s 9.195602e-01
square norm 3.194113e+07
```

Schlussendlich können wir noch die optimierte Version aus Beispiel `09_06_compiler_flags_mpi` ausprobieren und sehen ungefähr die gleiche Performance, wie bei der OpenMP-Version.

```
 $ mpirun -np 2 ./matrix_benchmark 512 512 1000 9890812
This is MPI rank with id=0 out of 2 ranks
This is MPI rank with id=1 out of 2 ranks
nrow= 512 ncol= 512 iters= 1000 time= 3.470305e-01
Gflop/s 3.021564e+00
square norm 3.194113e+07
```

Auch bei dieser Version scheint es so zu sein, dass wir von Hyperthreads profitieren:

```
 $ mpirun -np 4 ./matrix_benchmark 512 512 1000 9890812
This is MPI rank with id=0 out of 4 ranks
This is MPI rank with id=1 out of 4 ranks
This is MPI rank with id=2 out of 4 ranks
This is MPI rank with id=3 out of 4 ranks
nrow= 512 ncol= 512 iters= 1000 time= 1.888870e-01
Gflop/s 5.551335e+00
square norm 3.194113e+07
```

Zum Abschluss dieser Vorlesung möchte ich noch erwähnen, dass in modernen parallen Softwarepakaten Parallelisierung mit OpenMP und MPI gemischt wird. Es wird zum Beispiel mit OpenMP über die einzelnen Kerne einer CPU parallelisiert, während für Parallelisierung zwischen CPUs und mehreren Rechenknoten MPI genutzt wird. Nutzen die Parallelrechner zusätzlich Grafikkarten (GPUs) zur Erhöhung der Ausführungsgeschwindigkeit, werden noch weitere Technologien wie CUDA oder OpenCL genutzt, um die Grafikkarte anzusprechen. Diese hochparallelen Programme werden dadurch natürlich nicht einfacher zu handhaben...

10 That's all folks!

Damit sind wir am Ende der Vorlesung und des Kurses angelangt. Ich hoffe, dass die Erklärungen in dieser komplizierten Zeit mehr oder weniger verständlich waren und freue mich über Verbesserungsvorschläge jeglicher Art.
Die eCampus-Seite wird über die Dauer der Computerphysik bestehen bleiben und die Tutoren werden versuchen, auf C-spezifische Fragen im Diskussionsforum zu antworten, soweit dies ihnen zeitlich möglich ist.
Zur weieteren Lektüre habe ich ein paar Vorschläge gemacht, sowohl zu OpenMP und MPI, als auch zu weiteren Programmiersprachen, die in der Physik weitläufig genutzt werden.

Eine Sprache habe ich unerwähnt gelassen: FORTRAN (FORmula TRANslator) wird weiterhin von vielen Physikprogrammen, insbesondere in der Meteorologie und Plasmaphysik auch für große Softwareprojekte eingesetzt. FORTRAN-Programme sind relativ leicht verständlich, was die algorithmischen Schritte angeht und unterstützen natürlich sowohl OpenMP als auch MPI. FORTRAN hat jedoch die Nachteile, dass die Sprache recht antiquirt wirkt und, dass die Softwareentwicklung an sich (also die Entwicklung sauber getrennter modularer Programme) durch die Sprache erschwert wird.