

C-Kurs Physik, 2020

Bartosz Kostrzewa, Carsten Urbach

HISKP, Rheinische Friedrich-Wilhelms-Universität Bonn

März 2020

Administrativa

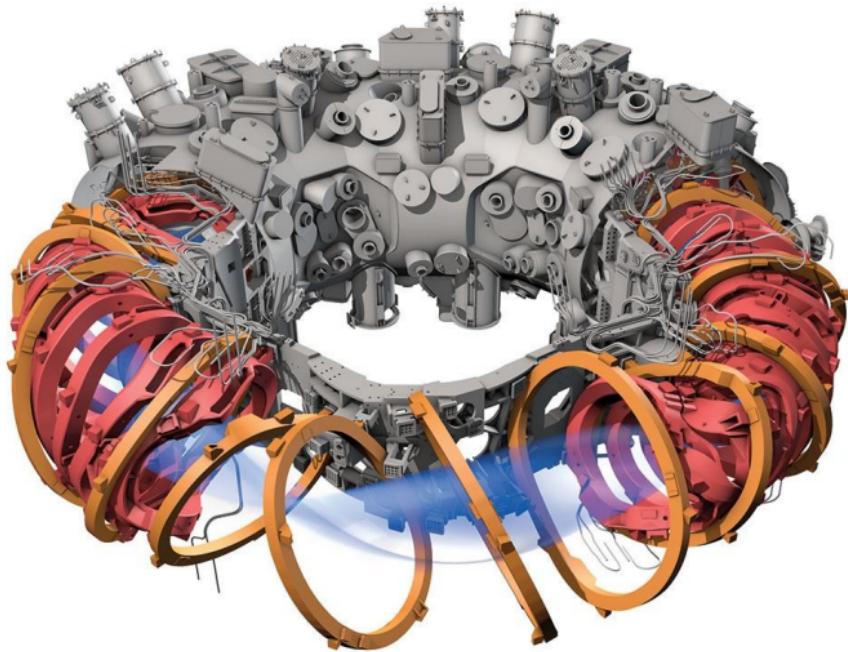
- Vorlesung 23.03-xx.04, Materialien unter
https://ecampus.uni-bonn.de/goto.php?target=crs_1661898
- Dozent: Bartosz Kostrzewa, bartosz_kostrzewa@fastmail.com, Raum 3.009 HISKP (bis auf unbestimmte Zeit im Homeoffice)
- Tutoren
 - Marcel Hohn, Marcel Nitsch, Simon Schlepphorst, Florian Taubert
- Zusätzlich zum Skript wird es ein Begleitskript geben, welches hoffentlich dabei hilft, den Folien folgen zu können.
- Fundamentale Ungleichung der Programmierung (FUP!):
Praxis » **Vorlesungen** → machen Sie die Übungen, sonst bringt die Vorlesung nichts!
- Folien, Skript, Übungszettel, Beispielprogramme Forum, sowie Fragen und Antworten auf eCampus

Lernziele

- Erstes Kennenlernen der sogennanten *imperativen, strukturierten, prozeduralen* Programmierung in einer kompilierten Programmiersprache
- Algorithmische Problemlösung
- Daten- und Kontrollstrukturen von C99
- Erstellen eigener C-Programme in den Übungen
 - einfache Beispielprogramme → kompliziertere Programme aus mehreren Quelltextdateien
 - Verwendung externer Bibliotheken
- Vorbereitung auf *physik441: Computerphysik* (SoSe 2020), *physics760: Computational Physics* (WiSe 2020/2021), etwaige Bachelor- und Masterarbeiten, weiterführende wissenschaftliche Arbeit

Anwendung: Plasmaphysik / Fusion

Wellenstein 7-x Stellerator

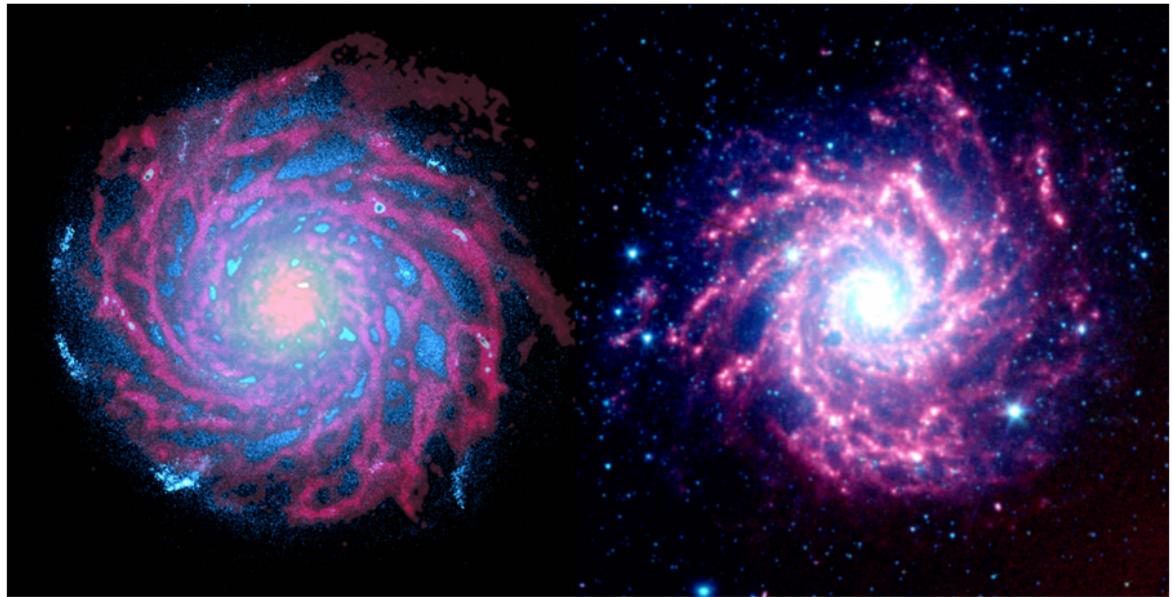


Quelle:

<http://www.sciencemag.org/news/2015/10/bizarre-reactor-might-save-nuclear-fusion>

Anwendung: Kosmologie / Astrophysik

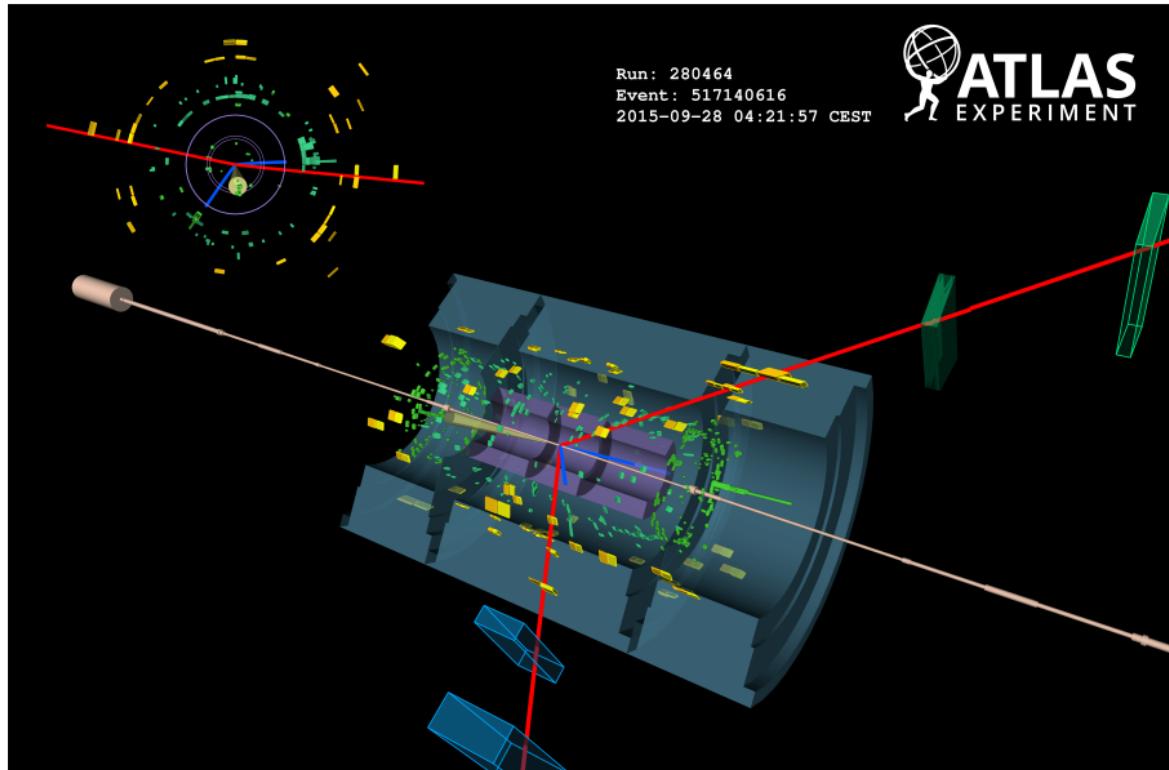
Simulation der Galaxie M74 über 13 Milliarden Jahre



Quelle: [http://www.hpc-ch.org/
first-realistic-simulation-of-the-formation-of-the-milky-way-computed-at-cscs/](http://www.hpc-ch.org/first-realistic-simulation-of-the-formation-of-the-milky-way-computed-at-cscs/)

Anwendung: Hochenergiephysik - LHC

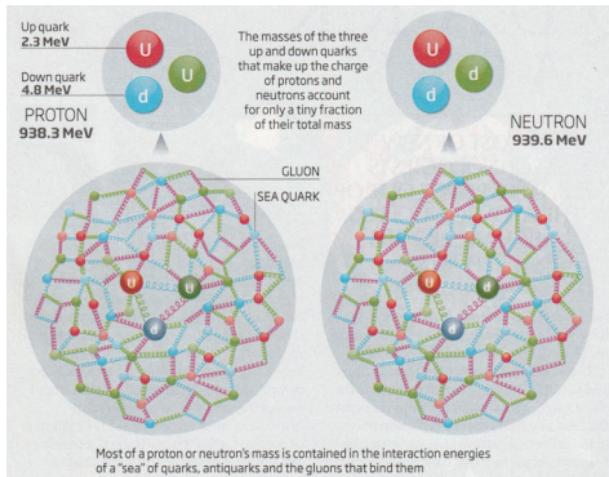
Kollision mit Higgs $\rightarrow ZZ^* \rightarrow 2\mu 2e$ Zerfall im ATLAS-Experiment



Quelle: <https://cds.cern.ch/record/2062050>

Anwendung: Hochenergiephysik / Gitter-QCD

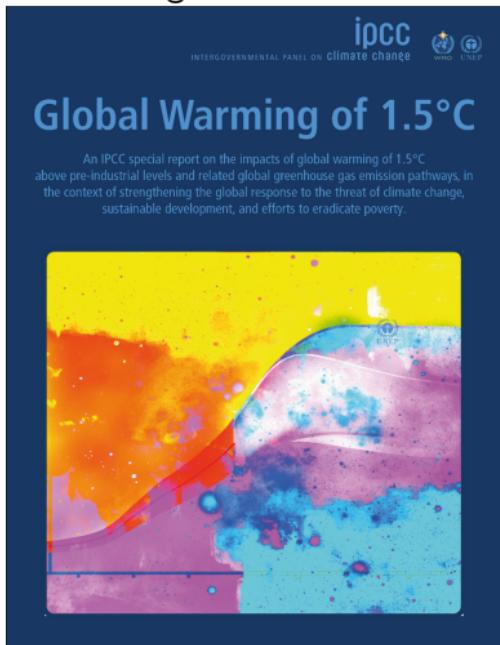
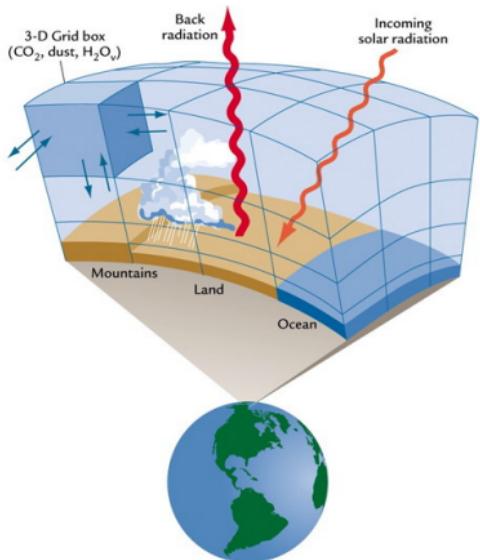
Simulation der Quantenchromodynamik auf Hochgeschwindigkeitsrechnern



Quellen: <http://universe-review.ca/R15-12-QFT18.htm>
<https://goo.gl/u6udnZ>

Anwendung: Klimaforschung

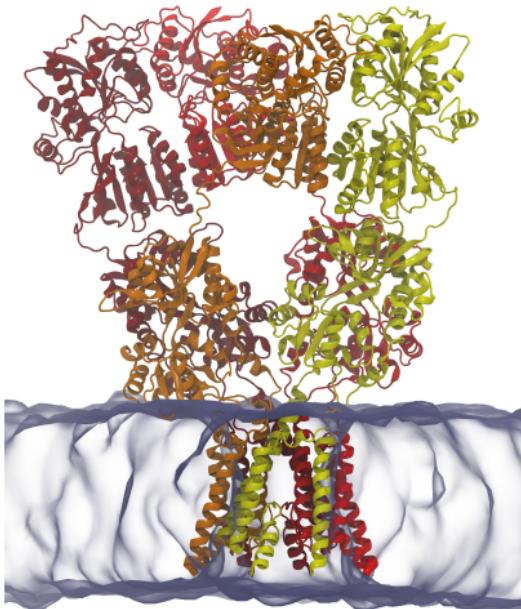
Gekoppelte dreidimensionale Strömungsmodelle der Atmosphäre sind ein zentrales Element in der Klimaforschung.



Quellen: <http://www.iac.ethz.ch/group/climate-physics/research.html>
<https://www.ipcc.ch/sr15/>

Anwendung: Biophysik / Biochemie - Molekulardynamik

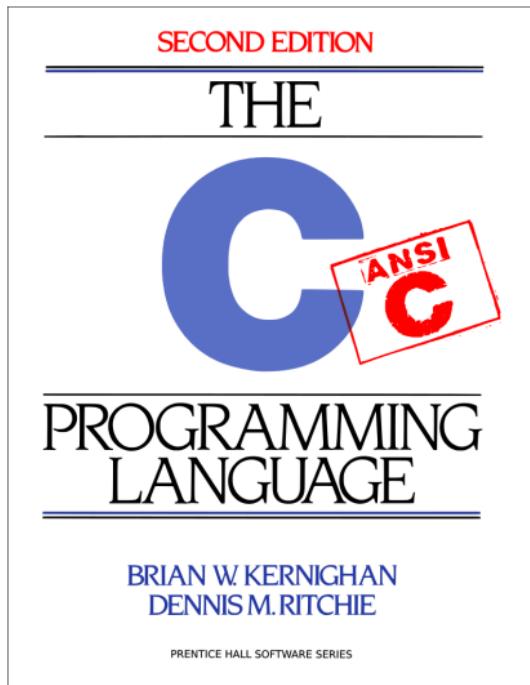
Simulation des NMDA Proteins und Rezeptors im menschlichen Gehirn



Quelle: <http://computation.llnl.gov/glutamate-receptor-molecular-dynamics-simulation>

Die Programmiersprache C

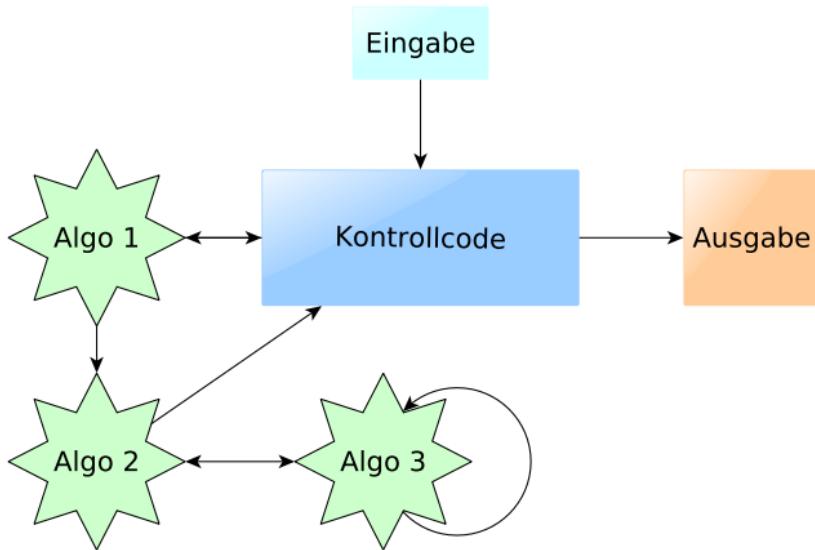
Vorlesung 1



- Besonders in der Wissenschaft sehr weit verbreitet!
- Imperativ: Präzise, schrittweise Befehle zur Lösung eines Problems
- Prozedural: Aufbauend auf Unterprogrammen (Funktionen) mit Argumentenübergabe und Rückgabewerten (nicht objektorientiert)
- Strukturiert: Aus Blöcken aufgebaut
- Effizient: übersetzt meist in relativ schnellen *Maschinencode*
- Maschinennah: erlaubt direkten Zugriff auf Speicher, Prozessor, Hardwarekomponenten

Programmstruktur und Algorithmen

Vorlesung 1



Algorithmus

Eine präzise Vorschrift, um aus vorgegebenen Eingaben in endlich vielen Schritten eine bestimmte Ausgabe zu ermitteln.

Algorithmen: Pseudocode

Vorlesung 1

Pseudocode: Schritt zwischen sprachlicher Beschreibung eines Verfahrens und des Quelltextes eines Programms.

Algorithmus 1 Einfügensortieren

Input: Lists U, S

Output: List S

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

- Pseudocode lässt sich nachträglich leicht in Quelltext übertragen.
- Wir wollen eine unsortierte Liste von Zahlen sortieren.
- Eingabe und Ausgabe definieren.
- Kontroll-, Logik- und Rechenanweisungen
 - Zuweisungen
 - Schleifen
 - Vergleiche
 - arithmetische Operationen
 - Abbruchbedingungen

Die Struktur eines C-Programms

Vorlesung 1

Das einfache Programm

```
#include <stdio.h>
int main(void){
    printf("Hallo, Welt!\n");
    return 0;
}
```

können wir in einer Textdatei abspeichern und mit dem C-Compiler in ein ausführbares Programm übersetzen

```
$ gcc -Wall -Wpedantic -std=c99 -o hallo_welt hallo_welt.c
```

und es dann ausführen

```
$ ./hallo_welt
Hallo, Welt!
$ _
```

⇒ Beispiel: 01_01_hello_world.c

Maschinencode und Übersetzung

Vorlesung 1

Der Compiler übersetzt den Quelltext

```
#include <stdio.h>
int main(void){
    printf("Hallo, Welt!\n");
    return 0;
}
```

in Maschinencode, welcher dann vom Prozessor ausgeführt werden kann.

```
0000000000400526 <main>:
```

400526: 55	push	%rbp
400527: 48 89 e5	mov	%rsp,%rbp
40052a: bf c4 05 40 00	mov	\$0x4005c4,%edi
40052f: e8 cc fe ff ff	callq	400400 <puts@plt>
400534: b8 00 00 00 00	mov	\$0x0,%eax
400539: 5d	pop	%rbp
40053a: c3	retq	
40053b: 0f 1f 44 00 00	nopl	0x0(%rax,%rax,1)

Variablen deklarieren und definieren

Vorlesung 1

Eine Variable wird in C mit:

```
DATENTYP NAME;           // deklariert
NAME = WERT;              // definiert
DATENYP NAME = WERT;     // deklariert und definiert
```

Blöcke und Sichtbarkeitsbereich (scope)

In C werden Programmblöcke mit { und } umklammert. Ein Block fasst mehrere Ausdrücke (*statements*) zu einem Ausdruck zusammen. Eine Variablen-deklaration gilt innerhalb eines Blocks und seiner Unterblöcke.

```
DATENTYP1 VAR1 = WERT1;          // Aeußerster Block
{
    DATENTYP2 VAR2 = WERT2;
    // hier gelten sowohl VAR1 als auch VAR2
    {
        DATENTYP3 VAR3 = WERT3;
        // hier gelten alle drei Variablen
    } // ab hier gilt VAR3 nicht mehr
} // jetzt gilt auch VAR2 nicht mehr
```

⇒ Beispiel: 01_02_scope.c

Einrückung und Kommentare

Vorlesung 1

```
DATENTYP1 VAR1 = WERT1;           // Aeusserster Block
{
    DATENTYP2 VAR2 = WERT2;
    // hier gelten sowohl VAR1 als auch VAR2
    {
        DATENTYP3 VAR3 = WERT3;
        // hier gelten alle drei Variablen
    } // ab hier gilt VAR3 nicht mehr
} // jetzt gilt auch VAR2 nicht mehr
```

Ein wichtiger Aspekt, der zur Lesbarkeit eines Quelltextes beiträgt, ist eine konsistente Einrückung der Programmblöcke. Man hätte auch folgendes schreiben können.

```
DATENTYP1 VAR1=WERT1;{DATENTYP2 VAR2=WERT2;{DATENTYP3 VAR3=WERT3;}}
```

An nicht-trivialen Stellen des Programmcodes ist es zudem wichtig, dass man Kommentare einfügt, um sich und anderen zu erklären, was da geschieht.

Datentypen

Vorlesung 1

C ist eine sogenannte statisch schwach-typisierte Programmiersprache.

- typisiert: Daten (also Variablen) werden zusammen mit ihrem Typ deklariert.
- Operationen sind auf Datentypen definiert (z.B. die arithmetischen Operatoren auf Zahlentypen)
- Umwandlungsregeln bestimmen, ob Typen ineinander umgewandelt werden können und wie dies zu geschehen hat.

Elementare Datentypen

Datentyp	Typische Größe	Anwendung
char	1 Byte	Zeichen (a-z, A-Z, 0-9 ...)
int	4 Bytes	ganze Zahlen
float	4 Bytes	kleine Fließkommazahl
double	8 Bytes	große Fließkommazahl

Datentypen - Modifikatoren

Vorlesung 1

- Modifikatoren erlauben es, den Datentyp genauer einzuschränken.

Modifikator	Bedeutung	Anwendung auf Datentyp
const	konstanter Wert	alle
short	kleinerer Datentyp	int
long	größerer Datentyp	int, manchmal double
signed	vorzeichenbehaftet	char, int
unsigned	positiv!	char, int

⇒ Beispiel: 01_03_modifikatoren.c

Operatoren

Vorlesung 1

Allgemeine Arithmetik- und Zuweisungsoperatoren

Operator	Datentypen
<code>+, -, *, /</code>	<code>float, double, int, char</code>
<code>+=, -=, *=, /=</code>	<code>float, double, int, char</code>
<code>++x, x++, --x, x--</code>	<code>int, char</code>

Ganzzahldivision und Modulo

`7 / 3 == 2`

`7 % 3 == 1`

Zuweisung

```
int x = 3;  
int y = x;
```

Logische Operatoren

Operator	Bedeutung
<code>==, !=, <=, >=, <, ></code>	Vergleiche
<code>&&</code>	Logisches UND
<code> </code>	Logisches ODER
<code>!0 == 1</code>	Logische Inversion
<code>!1 == 0</code>	

Weitere Operatoren: bitwise, `sizeof`, ternär (`? :`), Adressoperator `&`, Dereferenzierungsoperator `*`.

Bedingte Ausführung: if / else Statement

Vorlesung 1

Das if / else statement erlaubt es, den Programmfluss abhängig vom momentanen Zustand zu steuern.

Input: x, y, z

```
if( x > 0 ){
    // x groesser 0
} else if ( x == 0 && y <= 1 ) {
    // x gleich 0 UND y kleiner-gleich 1
    if( (z + 3) == y ){
        // z+3 gleich y
    }
} else if ( x == 0 && y > 1 ) {
    // x gleich 0 UND y groesser 1
} else {
    // In allen anderen Faellen
}
if( z > 0 || (y % 2 == 0) ){
    // z groesser 0 ODER y gerade
}
```

Schleifen: while & do-while (1/2)

Vorlesung 1

Iterative Verfahren sind ein zentraler Bestandteil vieler Algorithmen. Schleifen wiederholen Anweisungen, bis eine gewählte Bedingung erreicht ist.

Die while- und do-while-Schleifen sind die einfachsten Schleifentypen in C und haben folgende Struktur:

```
while( LOGISCHER AUSDRUCK ){  
    BEFEHLE  
}
```

```
do {  
    BEFEHLE  
} while( LOGISCHER AUSDRUCK );
```

- (1) AUSDRUCK auf Wahrheit prüfen
 wahr BEFEHLE ausführen
 ↳ Zurück zu (1)
 unwahr Schleife beenden

- (0) BEFEHLE ausführen
 (1) AUSDRUCK auf Wahrheit prüfen
 wahr zurück zu (0)
 unwahr Schleife beenden

Schleifen: while do-while (2/2)

Vorlesung 1

Wir wollen folgende unendliche Summe berechnen:

$$S = \sum_{n=0}^{\infty} x^n, \quad |x| < 1$$

```
const double x = 0.8;           // x ist eine Konstante
double S = 0.0;                 // S_0 = 0
double xn = 1.0;                // x^0 = 1.0
while( xn > 1.0e-12 ){
    S = S + xn;
    xn = xn * x;              // x^n
}
```

```
do {
    S += xn;
    xn *= x;
} while( xn > 1.0e-12 );
```

Schleifen: for (1/2)

Vorlesung 1

In C hat eine for-Schleife Kontrollelemente im Schleifenkopf, ideal um mit einer Zählvariable zu arbeiten (aber auch andere Konstrukte sind möglich).

```
for( INITIALISIERUNG; LOGISCHER AUSDRUCK; SCHRITT ){  
    BEFEHLE  
}
```

- (0) INITIALISIERUNG wird ausgeführt
 - (1) LOGISCHER AUSDRUCK wird auf Wahrheit geprüft
 - wahr Weiter zu (2)
 - unwahr Schleife beenden
 - (2) BEFEHLE werden ausgeführt
 - (3) SCHRITT wird ausgeführt
 - ⇒ Zurück zu (1)

Schleifen: for (2/2)

Vorlesung 1

Die Berechnung der Summe $\sum_{n=0}^{\infty} x^n$, könnte man, z.B. so implementieren:

```
const double x = 0.8;
double S = 0.0;
double xn = 1.0;
for( unsigned int n = 0; n < 1000; ++n ){ // max. 1000 Iterationen
    S = S + xn;
    if( xn < 1.0e-12 ){
        break; // aus der Schleife ausbrechen
    }
    xn = xn * x;
}
```

Oder aber auch so:

```
const double x = 0.8;
double S = 0.0;
for( double xn = 1.0; xn > 1.0e-12; xn *= x){
    S += xn;
}
```

Maschinenzahlen

Vorlesung 1

In unseren numerischen Algorithmen arbeiten wir mit Maschinenzahlen. Auf dem Rechner ist lediglich eine Teilmenge \mathcal{M} der reellen Zahlen darstellbar:

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

- $E \in \mathbb{N}$, $E > 1$ ist die **Basis** (meist $E = 2$)
- $k \in \mathbb{N}$ ist die **Genauigkeit**
- $e_{\min} < e < e_{\max}$, $e \in \mathbb{Z}$
- $a \in \mathbb{N}_0$ ist die **Mantisse**
 - $a = a_1 E^{k-1} + a_2 E^{k-2} + \dots + a_k E^0$
 - $a_i \in \{0, 1\}$

Beispiel:

$$(0.1)_{10} \rightarrow (0.0001\ 1001\ 1001\ 1001\dots)_2$$

Es muss also fast immer gerundet werden.

Maschinengenauigkeit: größte reelle Zahl δ_M für die der Rechner $1 + \delta M = 1$ auswertet. → Beispiel: 01_04_genauigkeit.c

Heute

Vorlesung 2

- Fragen zur Vorlesung / Übung 1?
- Text Ein- und Ausgabe
- Funktionen
- Statische Arrays

Wieso eigentlich C?

Vorlesung 1 Addendum

In den nächsten beiden Tagen wird Ihnen vielleicht klar werden, dass einige Aspekte von C doch komplizierter sind, als man glaubt. Wieso also C nutzen und nicht Python?

- C und C++ schaffen einen Spagat zwischen Nähe an der Hardware und einer relativ einfachen Sprachsyntax.
 - Wissenschaftliche Software muss oft so schnell wie möglich sein.
 - Effiziente Programmierung und gute Compiler machen dies in C und C++ möglich.

⇒ Performancevergleich C / Python3: <http://tiny.cc/lbsolz>
- In der Wissenschaft gibt es unglaublich viel Software die ursprünglich in C geschrieben wurde oder immer noch geschrieben wird. In Bachelor- Master- oder Doktorarbeiten, werden Sie damit in Kontakt kommen und müssen die "hässlichen" Grundlagen ein Mal gesehen haben.
- Andere Sprachen machen "unter der Motorhaube" nichts anderes als C, aber durch viele Abstraktionsebenen. Zu verstehen, was genau der Rechner da tun muss, hilft zu verstehen, wieso einige Dinge ineffizient sein könnten.

Mini-Intro: Textausgabe

Vorlesung 2

`printf` ist eine Funktion zur *formatierten* Textausgabe in der Konsole auf Basis eines *Formatstrings*. Um verschiedene Datentypen auszugeben, muss man Platzhalter nutzen. Sie ist in `stdio.h` deklariert:

```
int printf( const char *format, ...);
```

`char *format` ist die Notation für einen Zeiger auf `char` (nächste Vorlesung) und `...` ist eine variable Argumentenliste.

Platzhalter	Bedeutung	Anmerkung
<code>%c</code>	<code>char</code> (einzelnes Zeichen)	Dezimalwertausgabe mit <code>%d</code>
<code>%s</code>	String	Details: nächste Vorlesung
<code>%d, %i</code>	<code>int</code>	bei <code>printf</code> beide Dezimal
<code>%ld</code>	<code>long int</code>	
<code>%lld</code>	<code>long long int</code>	
<code>%u</code>	<code>unsigned int</code>	
<code>%llu</code>	<code>long long unsigned int</code>	
<code>%f, %e</code>	<code>float</code>	einfach oder in wiss. Notation

Mini-Intro: Textausgabe flags und Feldbreiten

Vorlesung 2

Die Platzhalter (auch *Umwandlungszeichen* genannt) können noch weiter spezifiziert werden mit genau **einem Flag**, der Feldbreite, sowie, bei Flieskommazahlen, der Anzahl an gewünschten Nachkommastellen.

```
printf("% FLAG FELDBREITE [.NACHKOMMASTELLEN] PLATZHALTER", ...);  
(""%10.5f\n", 3.1415); // 10 Zeichen breit, fuenf Nachkommastellen  
(""%010d\n", 42); // 10 Zeichen breit, aufgefuellt mit Nullen
```

Flag	Bedeutung
0	Auffüllung mit Nullen anstelle von Leerzeichen (bei angegebener Feldbreite)
-	Linksbündige Ausgabe
+	Vorzeichenbehaftete Ausgabe

- Feldbreite nach Flag
- Nachkommastellen nach Dezimalpunkt vor Platzhalter

⇒ Mal ausprobieren! (02_01_test_printf.c)

Mini-Intro: Texteingabe

Vorlesung 2

Der Adressoperator &: Für jede Variable, jedes Datentyps, gibt & die Speicheradresse der Variable zurück. (Um genauer zu sein, ist der Rückgabewert ein Zeiger auf ein Objekt des entsprechenden Datentyps)

Eingabe mit scanf: Die Funktion scanf kann in etwa als umgekehrtes printf verstanden werden, nutzt jedoch leicht andere Platzhalter und ignoriert Leerzeichen.

```
int ganzzahl; double dezimalzahl;  
scanf("%d %lf", &ganzzahl, &dezimalzahl);
```

Platzhalter	Bedeutung	Anmerkung
%f, %e, %g	float	
%lf, %le, %g	double	
%o	int	oktal
%x	int	hexadezimal

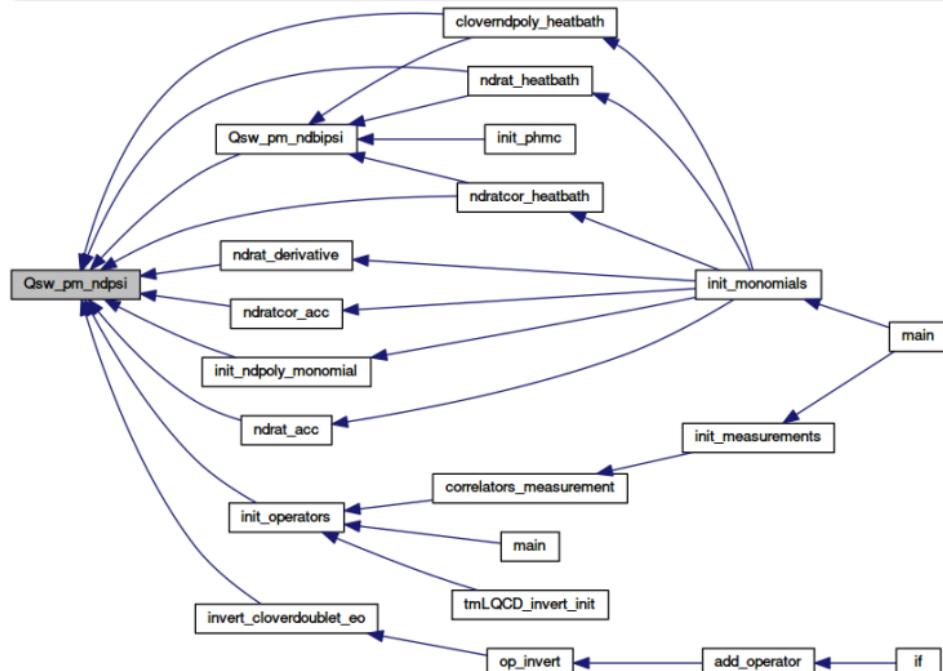
⇒ Beispiel: 02_02_test_scanf.c

Mehr Details: <https://goo.gl/mcYayS>

Funktionen

Vorlesung 2

C-Programme bestehen im Wesentlichen aus Funktionsdefinitionen und Funktionsaufrufen, sowie eingebauten und eigenen Datentypen. Ein komplexes C-Programm ruft viele Funktionen auf, welche wiederum Funktionen aufrufen.



Funktionen: Funktionskopf und Signatur

Vorlesung 2

Eine Funktion wird dem Compiler bekannt gemacht, indem ihr Name und die Datentypen ihrer Argumente festgelegt werden. ⇒ *Funktionsdeklaration*

```
RUECKGABETYP FUNKTIONSDNAME( ARGUMENT1_TYP ARG1, ARGUMENT2_TYP ARG2,  
                                ARGUMENT3_TYP ARG3 );
```

Signatur

Dies nennt man auch die *Signatur* der Funktion.

```
(FUNKTIONSDNAME) ( ARGUMENT1_TYP, ARGUMENT2_TYP, ARGUMENT3_TYP )
```

- Es gibt in C keine Überladung! Funktionen müssen eindeutige Namen haben!

Funktionen: Funktionsaufruf und Rückgabewert

Vorlesung 2

Hat eine Funktion einen Rückgabewert, so kann man diesen beim Aufruf einer Variablen zuweisen:

```
double result = teilen( 8.0, 3.2 );
```

Man kann den Rückgabewert aber auch einer anderen Funktion als Argument übergeben:

```
addieren( teilen(8.0, 3.2), 4.5 );
```

Oder aber man ignoriert den Rückgabewert, wie wir es bei `printf([...])` getan haben (Gesamtzahl der geschriebenen Zeichen).

Funktionen: Definition

Vorlesung 2

In der letzten Vorlesung hatten wir ein Programm entwickelt um eine Annäherung an die unendliche Summe

$$S = \sum_{n=0}^{\infty} x^n, \quad |x| < 1$$

zu bestimmen.

```
const double x = 0.8;
double S = 0.0;
double xn = 1.0;
for( unsigned int n = 0; n < 1000; ++n ){ // max. 1000 Iterationen
    S = S + xn;
    if( xn < 1.0e-12 ){
        break; // aus der Schleife ausbrechen
    } // selbst wenn n < 1000
    xn = xn * x;
}
```

Wir wollen nun mit einer Funktion die Berechnung auslagern und verallgemeinern: 02_03_sum_xn_funktion.c
Dann mit Eingabe: 02_04_sum_xn_eingabe.c

Funktionen: Deklaration und Definition

Vorlesung 2

Die *Funktionsdeklaration* informiert den Compiler über die Existenz und Signatur einer Funktion

```
double power( const double, const int );
```

Die *Funktionsdefinition* beschreibt erst, wie diese Funktion ihre Aufgabe erfüllt.

Funktionsdeklarationen führen einerseits zu lesbarerem Code, finden aber andererseits Verwendung in der *modularen Programmierung*, die wir in den nächsten Tagen kennenlernen werden.

Erstmal ein Beispiel. (02_05_funktion_deklaration.c)

Funktionen: Rekursion

Vorlesung 2

Rekursion

Eine Funktion, die sich selbst aufruft, nennt man *rekursiv*. Rekursion kann man oft nutzen, um Aufgaben auf fast magische Art und Weise zu lösen. Leider birgt Rekursion auch viele Gefahren.

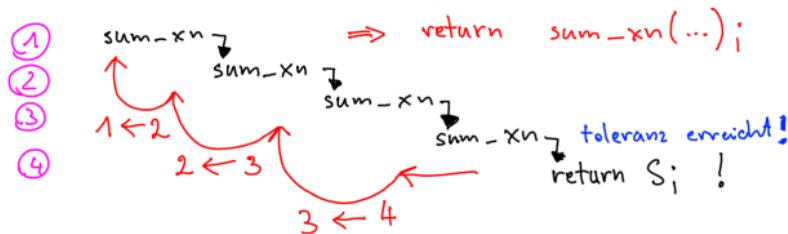
```
TYP rekursion( TYP arg ){
    if( [...] ){ // Ziel erreicht!
        return arg;
    } else {
        return rekursion( arg ); // noch einmal tiefer verschachteln!
    }
}
```

Bei einer rekursiven Funktion wird der nicht-rekursive (letzte) Rückgabewert im Aufrufstapel hochgereicht, bis der erste Aufruf das Ergebnis dann zurückgibt.

Rekursion

Vorlesung 2

```
sum_xn( x, s, xn, tol ){  
    if( xn < tol){  
        return s;  
    } else {  
        return sum_xn( x, s+xn, xn*x, tol );  
    }  
}
```



Versuchen wir mal, unsere Summe rekursiv auszurechnen!
(02_06_sum_xn_rekursiv.c)

gdb und valgrind

Nanu, was ist denn da schiefgegangen? Ein Debugger, ist ein Programm, mit dem man Fehler in anderen Programmen versuchen kann nachzuvollziehen.

Debugging-Symbole mit einbinden und Programm in gdb ausführen

```
$ gcc -Wall -Wpedantic -g -ggdb -std=c99 \
-o sum_xn_rekursiv_DEBUG sum_xn_rekursiv.c
$ gdb ./sum_xn_rekursiv_DEBUG
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
(gdb) run
```

valgrind

Ein weiteres nützliches Programm ist valgrind. Strikt gesehen kein Debugger im eigentlichen Sinne, aber spezialisiert darauf, Speicherfehler zu erkennen.

```
$ valgrind ./sum_xn_rekursiv_DEBUG
```

Statische Arrays

Vorlesung 2

Oft brauchen wir viele Daten des gleichen Datentyps. Diese kann man praktisch in **Arrays** abspeichern.

```
DATENTYP NAME[ANZAHL];
```

Hierbei muss, selbst in C99, ANZAHL fast immer eine numerische Konstante sein!

```
double a[3]; // Array mit 3 double deklarieren

double b[3] = {1.0, 2.3, 9.9}; /* Array mit 3 double deklarieren
                                 und initialisieren */

double c[4] = {4.5, 7.3}; /* Die ersten beiden Elemente mit
                           4.5 und 7.3 initialisieren,
                           alle anderen werden mit 0.0
                           initialisiert */

a[1] = 0.0001; // 0.0001 im zweiten Element von v abspeichern
```

⇒ Wir möchten nun für unsere Annäherung an die unendliche Summe alle Terme abspeichern und ausgeben! 02_07_sum_xn_alleterme.c

Preview: Zeiger auf Daten

Vorlesung 2

Zeigervariablen

Eine Zeigervariable speichert die Adresse einer Variablen für einen bestimmten Datenyp.

Dereferenzierungsoperator *

Dieser Operator gibt Zugriff zu den Daten, auf die der Zeiger zeigt.

⇒ Tafelbild + 02_08_zeiger_preview.c

Heute

Vorlesung 3

- Fragen zu Vorlesung 2?
- Zeiger
- Einfache Zeigerarithmetik
- Statische Arrays und Zeiger
- Zeichenketten (Strings)
- Zeiger an Funktionen übergeben

Vervollständigung: Logische Negation

Vorlesung 3

Wie schon erwähnt, gibt es in C keinen boolschen Datentyp. Jeder Datentyp kann als logischer Ausdruck interpretiert werden und der **!** Operator kann darauf angewandt werden, um den logischen Ausdruck zu negieren. Das wird in der Praxis auch oft genutzt...

```
double x = 0.0;
double y = !x;
double w = !(x); // gcc warnt, dass &x immer verschieden von 0 ist
int z1 = y;
int z2 = !y;
printf(" y = %f\n", y);
printf(" w = %f\n", w);
printf(" z1 = %d\n z2 = %d\n", z1, z2);
```

Ausgabe:

```
y = 1.000000
w = 0.000000
z1 = 1
z2 = 0
```

Der `sizeof` Operator und die Dereferenzierung

Vorlesung 3

`sizeof`

Der `sizeof` Operator gibt die Größe eines *statischen* Objektes in Bytes zurück.

```
int a[10];
double x = 4.2;
size_t size_of_a = sizeof(a);
size_t size_of_x = sizeof(x);
size_t size_of_double = sizeof(double);
size_t size_of_int = sizeof(int);
```

Dereferenzierung

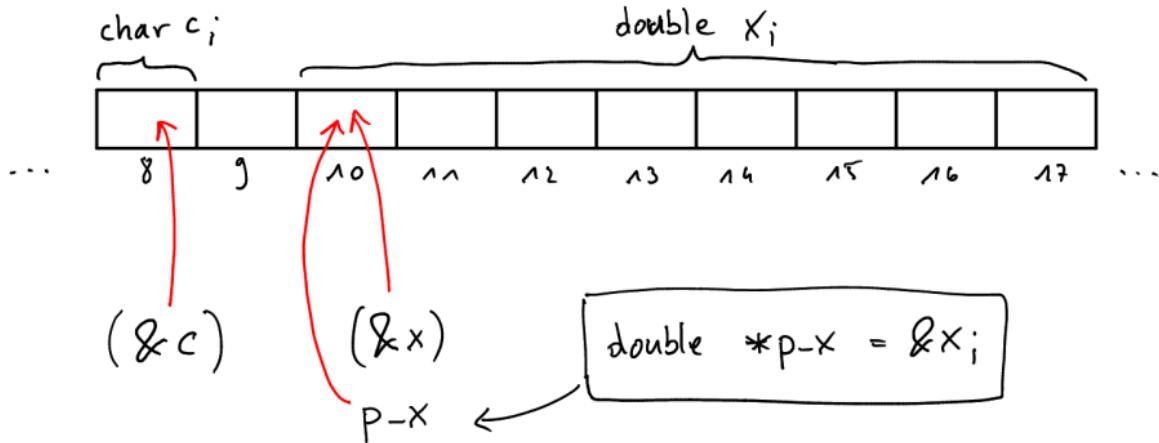
Hatten gesehen, dass der Adressoperator die Speicheradresse einer Variablen liefert. Diese kann man wieder dereferenzieren.

```
double x = 4.2;
printf("x = %f\n", x);
printf("x = %f\n", *(&x));
```

⇒ 03_01_sizeof_address_derefrence.c

Zeiger

Vorlesung 3



Zeiger sind Datentypen, welche die Adresse einer Variablen oder, allgemein, den Anfang eines Speicherbereichs darstellen.

⇒ 03_02_zeiger_demo.c und Tafelbild

Ein NULL-pointer zeigt auf gar nichts. (Strikt gesehen falsch: zeigt auf Adresse '0', diese Adresse darf nicht dereferenziert werden! [segmentation fault])

Arrays, Zeiger

Vorlesung 3

Statische Arrays sind spezielle Speicherbereiche, für die der Compiler einen Standardzeiger setzt. Dieser Zeiger ist der Name des Arrays.

```
int a[4]; /* 'a' hat formal den Datentyp (int*) und zeigt  
           auf ein Array des Typs (int) */  
int *b = a; /* 'b' hat auch den Datentyp (int*) und zeigt jetzt  
              auf den gleichen Speicherbereich wie 'a' */  
  
a[3] = 4; // direkter Zugriff auf Elemente von a  
  
b[3] = 4; // Zugriff über Zeiger
```

Strikt gesehen sind arrays und Zeiger (*pointer*) auf Daten des gleichen Datentyps aber nicht identisch!

⇒ 03_03_zeiger_array_demo.c und Tafelbild

Einfache Zeigerarithmetik

Vorlesung 3

Wir haben gesehen, dass `int*` oder `double*` spezielle Datentypen sind. Auf diesen sind auch arithmetische Operationen definiert.

```
int x;
int *p_x = &x;
p_x++; // Zeigerarithmetik

double y;
double *p1_y = &y;
double *p2_y = p1_y+1; // Zeigerarithmetik

double z[4] = {3.3, 3.4, 3.5, 3.6};
double *p1_z = z;
double *p2_z = &(z[0]);
z[1] = 4.2; // zweites Element von z
*(p1_z) = 4.3; // erstes Element von z
*(p2_z+2) = 4.5; // drittes Element von z
```

⇒ 03_04_zeiger_arithmetik.c und Tafelbild

Zeichenketten

Vorlesung 3

- Die Manipulation von Zeichenketten (auch *strings*) ist in vielen Programmen von zentraler Bedeutung, z.B. um Dateinamen für Ausgabedateien zu setzen.
- In C werden einfache strings in char-Arrays gespeichert. Da char nur 1B groß ist, stehen einem bloß 255 verschiedene Zeichen zur Verfügung.
- Strings werden in C *null-terminiert*: Am Ende eines Strings steht ein spezielles Zeichen, welches man auch selbst mit '\0' einfügen kann.
- Ein C String ist also ein char-Array der Länge n für maximal $n - 1$ Zeichen.

```
// dies ist ein string
char begruessung[20] = "Hallo, Welt!\n";
printf("%s", begruessung);
```

Vorsicht bei Zeigern auf Zeichenkettenkonstanten:

```
char * str = "String"; // kompiliert ohne Probleme
char const * str2 = "String"; // besser!
```

⇒ 03_05_test_strings.c

Strings vergleichen und bearbeiten: `strcmp` und `sprintf`

Vorlesung 3

- In `string.h` sind unheimlich viele Funktionen für Strings deklariert.
- `strcmp` zum Vergleichen zweier Strings.

```
char const * str1 = "Test";
char const * str2 = "Test";
int equal = strcmp(str1, str2); // gibt '0' zurück, wenn gleich
```

man pages durchsuchen: `man -k strcmp`

⇒ `man 3 strcmp`

- C enthält viele Funktionen zur Manipulation von Zeichenketten. Fast alle sind unsicher.
- Es gibt eine Ausnahme: `sprintf`

`03_06_unsichere_zeichenketten.c` und

`03_07_test_snprintf.c`

Pass-by-reference: Zeiger an Funktionen übergeben

Vorlesung 3

Sofern es sich nicht um globale Variablen handelt, existieren Variablen nur innerhalb eines Blocks. Aus Effizienzgründen ist es oft aber unratsam, Kopien großer Datenstrukturen hin- und herzuschieben.

Wir können aus der Funktion `inkrement`, nicht einfach so auf `x` aus der Funktion `main` zugreifen.

```
// inkrement hat keinen Rueckgabewert -> void
void inkrement(int const n){
    x += n; // Fehler: 'inkrement' kennt 'x' nicht!
}
int main(void){
    int x = 22;
    inkrement(4); // Fehler: 'inkrement' kennt 'x' nicht!
    return 0;
}
```

Mithilfe von Zeigern, können wir Funktionen direkt auf Daten anwenden, welche in einem anderen Block deklariert wurden. (wie bei `scanf` gemacht)

⇒ nächste Folie

Pass-by-reference: Zeiger an Funktionen übergeben

Vorlesung 3

Übergeben wir die Adresse von x an die Funktion, hat diese direkten Zugriff auf die Variable x aus der main-Funktion.

```
void inkrement(int *z_x, int const n){  
    *z_x += n;  
}  
int main(void){  
    int x = 22;  
    inkrement(&x, 4); // x wird von 'inkrement' um 4 inkrementiert  
}
```

Das können wir auch anhand unserer Summenfunktion zeigen:

⇒ 03_08_sum_xn_by_reference.c

Kommandozeilenargumente

Vorlesung 3

In unseren Programmen haben wir bisher

```
int main(void){ [...] }
```

geschrieben. `void` bedeutet leer oder auch ungültig. Wir haben damit darauf hingedeutet, dass unsere `main`-Funktion keine Argumente entgegennimmt.

- Der `main`-Funktion werden aber vom Betriebssystem Argumente übergeben:
die Kommandozeilenargumente

```
$ ./programm arg0 arg1 arg2
```

```
int main(int argc, char **argv){  
    [...]  
}
```

⇒ 03_09_test_argc.c

Kommandozeilenargumente auslesen

Vorlesung 3

- Bei `**argv` (oder auch `*argv[]`) handelt es sich um ein Array von Strings (ein Array von char-Arrays).
- C bietet einige Funktionen, die es erlauben aus Strings andere Datentypen auszulesen.
- `sscanf` ist eine mit `scanf` verwandte Funktion zum Auslesen einzelner Elemente aus formatierten Texten in Variablen
- Für Kommandozeilenargumente ist es oft praktischer einfachere Funktionen zu nutzen.
 - `atoi`, `atof`, `strtod`, `strtol`, `strtoul`, `strtoull`, ...
- Aber Vorsicht: `atoi` und `atof` prüfen nicht auf Fehler!

⇒ 03_10_test_read_argv.c

Heute

Vorlesung 4

- Fragen zu Vorlesung 3?
- Arrays an Funktionen übergeben
- Modulare Programmierung
- Der C-Präprozessor
- Zusammengesetzte Datenstrukturen und Typendefinition

Heute gibt es ein physikalisches Beispiel! (yaaay).

Rückblick: Zeiger

Vorlesung 4

Nochmal ein Rückblick auf Zeigernotation

```
double x; double y; // zwei double Variablen deklarieren
double arr_y[10]; // und ein double Array mit 10 Elementen

double *z_x, *z_y, w; /* zwei Zeiger auf double ('z_x' und 'z_y')
                       * und ein double ('w') */

z_x = &x; // 'z_x' zeigt jetzt auf 'x'
z_y = &y; // 'z_y' zeigt jetzt auf 'y'

*z_y = 4.2; // 'z_y' dereferenzieren und 'y' auf 4.2 setzen

z_y = arr_y; // Zeiger 'z_y' zeigt jetzt auf 'arr_y'

z_y[0] = 6.2; // erstes Element von 'arr_y' auf 6.2 setzen

z_x = &w; // 'z_x' zeigt jetzt auf 'w'

*z_x = 5.4; // 'w' wird auf 5.4 gesetzt
```

Vervollständigung: Zeiger auf lokale Variablen

Vorlesung 4

Die Rückgabe von Zeigern auf lokale Variablen ist nicht zulässig.

```
double* test_ptr(void){  
    double v = 4.2; // v existiert nur innerhalb des Funktionsblocks!  
    return &v; // Compiler warnt, aber kein Fehler!  
}  
int main(void){  
    double *z_v = test_ptr(); // legale Zuweisung, kein Fehler!  
    printf("%f\n", *z_v ); // Segmentation Fault bei Dereferenzierung!  
    return 0;  
}
```

Dies gilt natürlich nicht für Zeiger auf dynamisch allokierten Speicher (Vorlesung 5)

Arrays übergeben

Vorlesung 4

Hatte erwähnt, dass Zeiger z.B. zur Übergabe von Arrays an Funktionen genutzt werden.

```
void init( double *p_x, double *p_y, double *p_z, int const N );
double x[100];
double y[100];
double z[100];
init( x, y, z, 100 );
// oder: init( &x[0], &y[0], &z[0], 100 );
// Jetzt können x, y und z hier genutzt werden
```

⇒ 04_01_array_uebergabe.c

Zeiger auf Zeiger auf Zeiger auf Zeiger ...

Vorlesung 4

- Wir hatten schon eine Anwendung von Zeigern auf Zeiger gesehen: die Kommandozeilenargumente als Array von char-Arrays.
- Weitere Anwendungen sind mehrdimensionale Arrays (morgen), oder Funktionen, die einen Zeiger umdefinieren sollen.

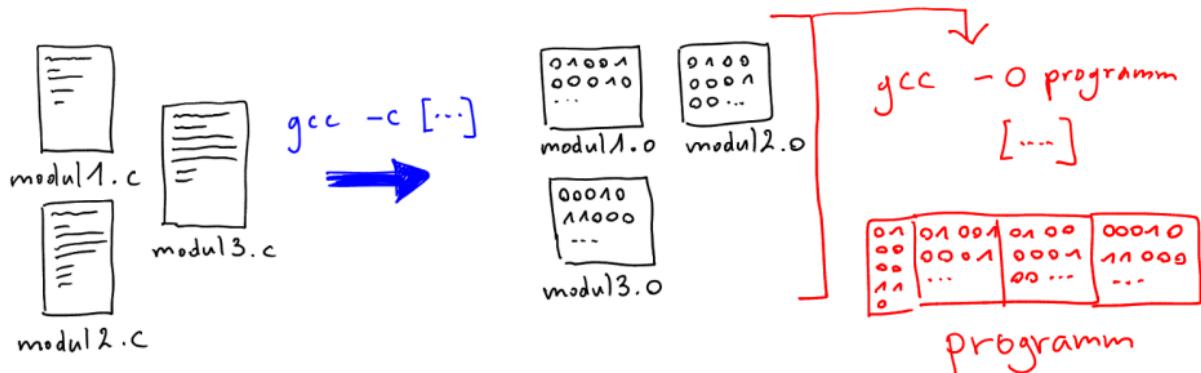
```
double x = 4.2;
double *p_x = &x; // Zeiger auf double
double **p_p_x = &p_x; // Zeiger auf Zeiger auf double
double ***p_p_p_x = &p_p_x; // Zeiger auf Zeiger auf Zeiger auf
                           double
```



- Und noch ein sinnvolles Beispiel:
⇒ 04_02_zeiger_vertauschen.c

Modulare Programmierung

Vorlesung 4



- Themenspezifische Funktionssammlungen können in C in separaten Quelltextdateien stehen und unabhängig voneinander kompiliert werden.
 - 1 Funktionen in separate Dateien auslagern
 - 2 Einzelne Module kompilieren
\$ `gcc -c modulX.c → modulX.o`
 - 3 Module zu Programm verlinken
\$ `gcc -o programm modul1.o modul2.o`

Modulare Programmierung: Quell- und Header-dateien

Vorlesung 4

x sei ein Array, welches die 1D Position eines Teilchens zur Zeit t enthält. Wir wollen die kinetische Energie berechnen. Die einzelnen $x[i]$ liegen im Abstand von $dt = 0.01$.

```
#include "AbiT.h"
double
kinEnerg( double *x,
           double m,
           unsigned int t ) {
    double v = AbiT( x, t );
    return 0.5*m*v*v;
}
```

kinEnerg.c

```
double
Abit( double *x,
       unsigned int t){
    double dt = 0.01;
    [...] // hier Checks
    return (x[t+1]-x[t])/dt;
}
```

```
double kinEnerg( double*,
                  double,
                  unsigned int );
```

kinEnerg.h

In den Header-dateien sind die Funktionsdeklarationen.

```
double AbiT( double*,
              unsigned int );
```

Abit.h

Vollständiger Code gleich im Beispiel.

Abit.c

Compiler und Linker

Vorlesung 4

Die Kompilierung und das *Linken* sind unabhängige Schritte bei der Erstellung eines ausführbaren Programms.

Kompilierung

```
# erstellt kinEnerg.o Objektdatei  
$ gcc -c kinEnerg.c
```

```
# erstellt AbLT.o Objektdatei  
$ gcc -c AbLT.c
```

```
# erstellt teilchen.o Objektdatei  
$ gcc -c teilchen.c
```

```
# erstellt ausfuehrbares Programm  
$ gcc -o teilchen teilchen.o AbLT.o kinEnerg.o -lm
```

⇒ (teilchen/kinErg.c teilchen/AbLT.c teilchen/teilchen.c)

Modulare Programmierung: Interface und Implementierung

Vorlesung 4

Trennung Interface/Implementierung → Implementierung austauschbar!

```
#include "AbiT.h"
double
kinEnerg( double *x,
           double m,
           unsigned int t ) {
    double v = AbiT( x, t );
    return 0.5*m*v*v;
}
```

kinEnerg.c

```
double kinEnerg( double*,
                  double,
                  unsigned int );
```

kinEnerg.h

Es gilt immer noch die gleiche Header-datei mit der Deklaration von AbiT(...).

```
double AbiT( double *x,
              unsigned int t ){
    double dt = 0.01;
    return (x[t+1]-x[t-1])/(2*dt);
}
```

AbiT_symmetrisch.c

```
double AbiT( double*,
              unsigned int );
```

AbiT.h

Modulare Programmierung: Implementierung austauschen

Vorlesung 4

- Symmetrische Ableitung, $\frac{x(t+1)-x(t-1)}{2\delta t}$ hat einen kleineren Näherungsfehler als die Vorwärtsableitung aus AbLT.c.
- Unsere Implementierung dieser Ableitung in AbLT_symmetrisch.c hat die gleiche Signatur (also auch den gleichen Namen) wie AbLT(...).
- Wir wollen also die symmetrische Ableitung nutzen!

```
$ gcc -c AbLT_symmetrisch.c
$ gcc -o teilchen_AbLT_symmetrisch teilchen.o AbLT_symmetrisch.o
kinEnerg.o -lm
```

- Wir haben den Namen des Programms geändert, damit wir wissen, dass hier eine andere Ableitung genutzt wird.
⇒ (teilchen/kinErg.c, teilchen/AbLT_symmetrisch.c, teilchen/teilchen.c)

Der C-Präprozessor

Vorlesung 4

- Wir haben bisher einen Teil des Kompilierprozesses ausgelassen: den Präprozessor.
- Bevor der Compiler seinen Dienst verrichtet, wird der Quelltext vom Präprozessor bearbeitet.

Präprozessorkonstanten und Makros

Ein Makro ist ein Stück Code mit einem Namen, welches vom Präprozessor beim Durchlauf in den Quelltext eingefügt wird.

```
// MY_PI wird der Wert des Textes bis zum Ende der Zeile zugewiesen  
#define MY_PI 3.1415
```

- Schreiben wir jetzt in unserem Programm MY_PI, ersetzt der Präprozessor, bevor die Datei kompiliert wird, MY_PI durch 3.1415
- Mit den Präprozessorkommandos #ifdef und #ifndef kann man im Präprozessor die Existenz eines Makros überprüfen

⇒ 04_03_test_makros.c

Zusammengesetzte Datenstrukturen

Vorlesung 4

Konzeptionell ist es oft ratsam, mehrere Variablen zu einem *struct* zusammenzufassen. Die Position unseres Teilchens könnte in zwei Dimensionen z.B. mit folgender Datenstruktur dargestellt werden:

```
struct pos_st {  
    double x;  
    double y;  
}; // Achtung: nach struct Definition -> Semikolon  
  
// Variable "teilchen1" vom Typ "struct pos_st"  
struct pos_st teilchen1;  
  
teilchen1.x = 1.2; // Zugriff auf Elemente ueber ','  
teilchen1.y = 1.3;  
  
struct pos_st teilchen2[1000]; // Array von struct  
teilchen2[0].x = 8.3; // Zugriff auf struct-Elemente  
teilchen2[0].y = 4.2; // im Array teilchen2  
  
struct pos_st *z_t; // Zeiger auf "struct pos_st"  
z_t = &teilchen1;  
z_t->x = 3.3; // Zugriff auf Elemente ueber Zeiger 'z_t' und '>'
```

⇒ 04_04_test_struct.c

Mehrfachdefinitionen?

Vorlesung 4

Die Definition einer zusammengesetzten Datenstruktur wird in der Regel in eine Headerdatei gelegt, damit man sie aus mehreren Modulen nutzen kann.

```
#include "pos_st.h"
#include "AbLT.h"
double
kinEnerg( struct pos_st *pos,
          double m,
          unsigned int t ) {
    [...]
}
```

```
#include "pos_st.h"
double kinEnerg( struct pos_st *,
                  double,
                  unsigned int );
```

kinEnerg.h
⇒ 04/teilchen_pos

kinEnerg.c

```
#include "pos_st.h"
double AbLT(struct pos_st *pos,
            unsigned int t){
    [...]
}
```

```
#include "pos_st.h"
double AbLT( struct pos_st *,
              unsigned int );
```

AbLT.h

AbLT.c

Include / Header Guards

Vorlesung 4

```
#include "xyz.h" // "xyz.h" wird eingebunden
```

Wenn eine Header-datei in einem Program mehrfals eingebunden wird, muss sichergestellt sein, dass die darin enthaltenen Definitionen und Deklaration nur einmal durchgeführt werden! → *include guards*

```
#ifndef ABLT_H
#define ABLT_H
double AbLT( double*,
              unsigned int );
#endif // um welches #if[n]def handelt es sich hier? ABLT_H!
```

- 1 AbLT.h wird das erste Mal eingebunden: ABLT_H ist nicht definiert.
 - ▶ Definiere ABLT_H und deklariere Funktion
- 2 AbLT.h wird zum $(1 + n)$ -ten Mal eingebunden:
 - ▶ ABLT_H ist schon definiert → überspringe alles bis zu #endif

Include / Header Guards

Vorlesung 4

- “One definition rule” (ODR): Ein Objekt darf genau ein mal definiert werden
- Mehrfacheinbindung kann zur Mehrfachdefinition führen → **header guards** to the rescue.

```
#ifndef POS_ST_H
#define POS_ST_H
struct pos_st {
    double x;
    double y;
};
#endif // ifndef(POS_ST_H)
```

⇒ 04/teilchen_pos_guarded

Vorschau: Vorlesung 5

- Mehrdimensionale Arrays
- Dynamische Speicherverwaltung
 - ▶ Sicherer Umgang mit Speicher!
- Dynamische Arrays

- Fragen zu Vorlesung 4?
- Dynamische Speicherverwaltung
- Mehrdimensionale statische Arrays
- Mehrdimensionale dynamische Arrays

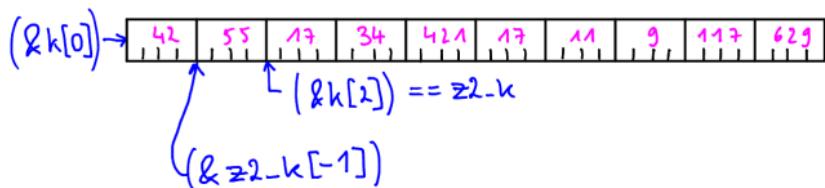
Zeigerarithmetik Quiz

Vorlesung 5

k ist ein int-Array mit 10 Elementen, initialisiert mit

{42, 55, 17, 34, 421, 17, 11, 9, 117, 629}. Wenn Sie jede Gleichheit verstehen, haben Sie Zeigerarithmetik verstanden.

int k[10]; int *z1_k = k; int *z2_k = &k[2];



- $42 == k[0] == *z1_k == *k == z1_k[0] == z2_k[-2]$
- $55 == k[1] == *z2_k - 1 == *(k + 1)$
- $34 == z2_k[1] == *(k + 3)$

⇒ $k[n]$ ist ein praktisches Kürzel für $*(k+n)$.

Dynamische Speicherverwaltung

Vorlesung 5

malloc und free

- Speicher wird in C mit malloc allokiert und mit free wieder freigegeben.
- Man sollte immer überprüfen, ob eine Allokation erfolgreich war.
- Wenn man den Speicher nicht mehr braucht, muss er freigegeben werden, sonst entstehen *memory leaks*.

```
int n = 300;
// Platz fuer 300 'double'-Variablen allokiieren
double *x = (double*)malloc( sizeof(double) * n );
if( x == NULL ){
    printf("Speicherallokation fuer 'x' fehlgeschlagen!\n");
    exit(32); // Rueckgabewert > 0 bedeutet Fehler
}
x[23] = 4.2; x[3] = 2.4; // 'x' verwenden
free(x); /* Speicher 'x' wird nicht mehr gebraucht
           * -> freigeben */
/* Programm geht weiter */
```

⇒ 05_01_test_malloc.c, 05_02_test_malloc_segfault.c

Speicherverwaltung: Memory Leaks

Vorlesung 5

- Wird ein Speicherbereich a einem Zeiger zugewiesen und dann ein weiterer Speicherbereich, b , dem gleichen Zeiger zugewiesen, geht die Adresse von a verloren. → Speicherleck (memory leak)
- Der Speicher ist belegt und kann, bis das Programm beendet wird, nicht freigegeben werden.
- Memory leaks sind Bugs, welche oft erst in den unangenehmsten Situationen auftauchen. Man sollte seine Programme überprüfen wenn möglich.

```
// Platz fuer 100 'double'-Variablen allokinieren
double *x = (double*)malloc( sizeof(double) * 100 );
/* x wird einem anderen Speicherbereich zugewiesen
 * -> Speicherleck */
x = (double*)malloc( sizeof(double) * 55 );
free(x); /* free'd nur "malloc( sizeof(double) * 55 )"
           * verloren: "malloc( sizeof(double) * 100 )" */
```

⇒ 05_03_memory_leak.c

Mehr Speicherverwaltung

Vorlesung 5

```
void* memmove(void *dest, void *src, size_t n);
```

Mit `memmove` werden n Bytes von `src` nach `dest` kopiert. Speicherbereiche dürfen überlappen.

```
void* memcpy(void *dest, void const *src, size_t n);
```

Mit `memcpy` werden n Bytes von `src` nach `dest` kopiert. Speicherbereiche dürfen **nicht** überlappen. → schneller als `memmove`.

```
void* memset(void *dest, int b, size_t n);
```

Mit `memset` werden n Bytes ab Speicherstelle `dest` auf den Bytewert `b` gesetzt. Oft hat man `b=0`.

⇒ 05_04_test_memmove_memset.c

Noch mehr Speicherverwaltung

Vorlesung 5

Einen allokierten Speicherbereich der Größe `n_old`, kann man nachträglich mit `realloc` vergrößern oder verkleinern.

```
void* realloc(void *ptr, size_t n_new);
```

- Der Rückgabewert und `ptr` müssen nicht übereinstimmen!
 - Schlägt `realloc` fehl, ist der Rückgabewert `NULL`, `ptr` bleibt aber weiterhin allokiert.
 - Wenn nach `ptr+n_old` kein zusammenhängender Speicherbereich der Größe `n_new-n_old` verfügbar ist, wird ein neuer Speicherbereich der Größe `n_new` allokiert, die Daten aus `ptr` werden dorthin kopiert, `ptr` freigegeben und ein Zeiger auf den neuen Speicherbereich zurückgegeben.
- Ist `n_new < n_old`, bleibt der Anfang von `ptr` bis `n_new` unverändert.

```
DATENTYP* temp = (DATENTYP*) realloc( ptr, n_new );  
if( temp != NULL ){ ptr = temp; }  
// ptr verwenden
```

⇒ 05_05_test_realloc.c

Mehrdimensionale Arrays

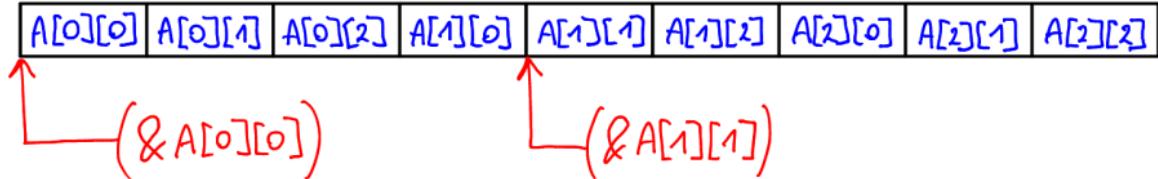
Vorlesung 5

Statische Mehrdimensionale Arrays

Ein statisches mehrdimensionales Array hat in C eine relativ einleuchtende Notation.

```
double A[3][3];      // 3x3 double Matrix  
double B[2][5][3];  // 2x5x3 Tensor mit 3 Indices
```

Es liegt am Stück in einer festgelegten Reihenfolge linear im Speicher.



Mehrdimensionale Arrays

Vorlesung 5

```
double B[2][5][3]; // 2x5x3 Tensor mit 3 Indices
```

Die Ordnung der Linearisierung ist Standard. Der am weitesten rechts liegende Index "läuft" am schnellsten.

B[0][0][0]	B[0][0][1]	B[0][0][2]	B[0][1][0]	B[0][1][1]	B[0][1][2]	B[0][2][0]	B[0][2][1]	B[0][2][2]
------------	------------	------------	------------	------------	------------	------------	------------	------------

Dynamische mehrdimensionale Arrays werden genauso angelegt, um den Zugriff effizient zu machen.

Linearisierung Mehrdimensionaler Arrays

Vorlesung 5

Wir wollen eine $N \times N$ Matrix linear im Speicher ablegen, wie verfahren wir?

```
int N = 42;
double *A = (double*)malloc( N*N*sizeof(double) ); // 42^2 doubles
```

Der Index i für den Zugriff auf A_{zs} , ergibt sich aus der Formel:

$$i = s + N \cdot z$$

Für ein Objekt mit 3 Indizes, $T_{i_0 i_1 i_2}$ der Größe $N_0 \times N_1 \times N_2$, ergibt sich:

$$\begin{aligned} i &= i_2 + N_2 \cdot (i_1 + (N_1 \cdot i_0)) \\ &= i_2 + N_2 \cdot i_1 + N_2 \cdot N_1 \cdot i_0, \end{aligned}$$

wobei $i_0 \in [0, N_0 - 1]$, $i_1 \in [0, N_1 - 1]$, $i_2 \in [0, N_2 - 1] \Rightarrow i \in [0, N_0 \cdot N_1 \cdot N_2 - 1]$

Für unsere Matrix A haben wir also:

```
A[ 1 + N*2 ] // A_{21}
```

Dynamische Mehrdimensionale Arrays

Vorlesung 5

Wir wollen eine $N_0 \times N_1$ Matrix erstellen, aber nicht als statisches Array. Wir wollen es uns aber auch erlauben, darauf mit `[] []` zuzugreifen. Wie gehen wir vor?

```
unsigned int N_0 = 27;
unsigned int N_1 = 13;
// Speicher fuer N_0*N_1 double-Werte
double *matrix_mem = (double*)malloc( N_0*N_1*sizeof(double) );
// Speicher fuer N_0 double-Zeiger
double **Matrix = (double**)malloc( N_0*sizeof(double*) );
for( unsigned int z = 0; z < N_0; z++ ){
    Matrix[z] = matrix_mem + N_1*z;
    // oder auch
    // Matrix[z] = &(matrix_mem[N_1*z]);
}
Matrix[2][1] = 2.2;
```

⇒ 05_06_lin_multidim_array.c und
05_07_dyn_multidim_array.c

Visualisierung: Dynamischer Matrixspeicher

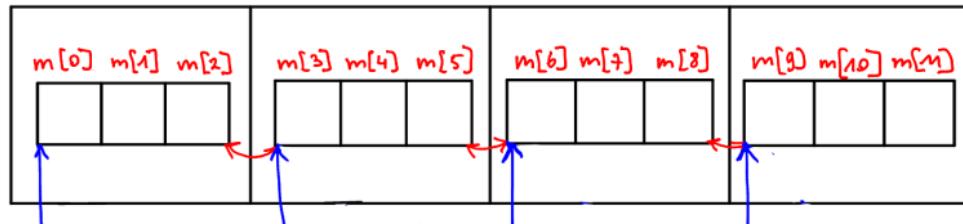
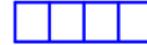
Vorlesung 5

4x3
Matrix
 M

$m = \text{malloc} (12 * \text{sizeof}(\text{double}))$;



$M = \text{malloc} (4 * \text{sizeof} (\text{double} *))$;



$$M[0] = m;$$

$$M[1] = m + 3;$$

$$M[2] = m + 2 \cdot 3;$$

$$M[3] = m + 3 \cdot 3;$$

$$\Rightarrow (M[1][2] == m[5]), (M[3][0] == m[9])$$

Vorsicht: Spaltendimension nicht automatisch beschränkt, da es sich bei $M[z][s]$ um ein Alias für $m + 3*z + s$ handelt $\Rightarrow M[0][3] == M[1][0]$

Eigene Typen anlegen: `typedef`

Vorlesung 5

Hatten in Vorlesung 4 Datenstrukturen kennengelernt. Mit `typedef` können wir ein Kürzel für einen komplexen Datentyp erstellen.

```
typedef double gleitzahl; // Alias 'gleitzahl' fuer 'double'  
gleitzahl x = 4.2;  
  
typedef struct teilchen_2d_t {  
    double x;  
    double y;  
    double v_x;  
    double v_y;  
    double m;  
    int ladung;  
} teilchen_2d_t; // <- typedef kombiniert mit struct Definition  
  
teilchen_2d_t teilchen; // Objekt des Typs 'teilchen_2d_t'  
teilchen_2d_t *gasarray = (teilchen_2d_t*)malloc( 10000*sizeof(  
    teilchen_2d_t) );  
void zeitschritt_gas( teilchen_2d_t* gas ){ ... }
```

Oft hebt man selbst definierte Datentypen durch ein `_t` im Quelltext hervor.

Vorschau auf Vorlesung 6

- Input / Output mit Dateien

Heute

Vorlesung 6

- Fragen zu Vorlesung 5?
- Input / Output aus / in Dateien

Eingabe und Ausgabe: fopen

Vorlesung 6

C bietet eine Zahl von Funktionen zur Eingabe und Ausgabe, zunächst muss eine Datei jedoch immer geöffnet werden.

```
FILE* fopen(char const *pfad, char const *modus);
```

Modus ist ein **string**, anhand dessen entschieden wird, welche Dateioperationen durchgeführt werden.

modus	Bedeutung
"w"	(erstellen) (über-)schreiben
"r"	lesen
"a"	(schreiben) erstellen oder hinzufügen
"w+"	erstellen oder überschreiben + lesen
"r+"	lesen + schreiben (nicht erstellen!)
"a+"	wie "a" + lesen (Lesezeiger am Anfang der Datei!)

Der Rückgabewert von fopen ist bei Erfolg ein *Dateizeiger* auf einen *Stream*, ansonsten ein Nullpointer. → auf NULL prüfen!

Weitere Details zu Modus: <https://goo.gl/Uvykxm>

Eingabe und Ausgabe: fopen

Vorlesung 6

```
FILE* ausgabedatei = fopen("/pfad/zu/den/daten/daten.txt", "r");
if( ausgabedatei != NULL ){
    // Datei bereit, ausgelesen zu werden
} else {
    // Datei existiert nicht oder sonstiger Fehler!
}
```

pfad kann hierbei absolut sein:

- UNIX: /home/user/pfad/zu/den/daten/daten.txt
- Windows: C:\Eigene Dateien\Dokumente\daten.txt

oder relativ zum Verzeichnis, in dem das Programm ausgeführt wurde:

- UNIX: pfad/zu/den/daten/daten.txt
- Windows: pfad\zu\den\daten\daten.txt

Im zweiten Fall wird der Pfad des momentanen Verzeichnisses vorgestellt.

Erst, nachdem die Datei geschlossen wurde, kann sichergestellt sein, dass auch alles geschrieben wurde. → fclose

⇒ (06_01_test_fopen_fclose.c)

Formatierte Textdateien lesen mit `fscanf`

(s) `scanf` hatten wir kurz erwähnt, als es um Kommandozeilenargumente und das Einlesen von Daten von der Konsole ging. `fscanf` ist eine ähnliche Funktion für Dateistreams.

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

- Rückgabewert: Anzahl gelesener Felder
- Auch unvollständiges Lesen bewegt den Dateicursor
- → Rückgabewerte überprüfen und Fehler identifizieren!

⇒ 06_02_test_fscanf.c

Mit fprintf Textdateien schreiben

Vorlesung 6

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

fprintf funktioniert wie printf, mit dem Unterschied, dass ein Dateizeiger übergeben werden muss. Um genau zu steuern, wie die Ausgabe auszusehen hat können die Platzhalter durch Parameter gesteuert werden.

```
% + - 0 BREITE . PRAEZISION ll/l/h [d,u,s,c,p,g,e,f]
```

format, argument Ausgabe (. = Leerzeichen)

%.2f, 2.3453	2.35
%5d, 32	...32
%05d, 32	00032
%05d, 999932	999932
%+05d, 32	+0032 // Auffuellen mit 0 & Vorzeichen
%-5d, 32, 1	32...1....// nicht gleichzeitig moeglich
%8.3f, 42.3453	..42.345

Mit fwrite / fread Binärdateien schreiben und lesen

Vorlesung 6

Datenstrukturen in C können direkt blockweise ausgeschrieben und eingelesen werden, solange man einen entsprechenden Speicherbereich allokiert hat.

```
#include <stdio.h>
size_t fwrite(void* data, size_t blockgr, size_t anzahl, FILE* fp);
size_t fread(void* data, size_t blockgr, size_t anzahl, FILE* fp);
```

modus	Bedeutung
fopen(..., "wb")	(erstellen) (über-)schreiben (Binärmodus)
fopen(..., "rb")	lesen (Binärmodus)

Um zu prüfen, ob das Ende der Datei erreicht wurde oder ein Fehler aufgetreten ist, werden feof und ferror genutzt.

```
int feof(FILE* fp);
int ferror(FILE* fp);
```

⇒ 06_0[3,4]_test_fwrite.c und 06_0[3,4]test_fread.c

Systemabhängigkeit von fwrite / fread

Vorlesung 6

Endianness und **Padding** machen Binärdaten *inkompatibel* zwischen verschiedenen Architekturen!

- Ob das grösste oder das kleinste Bit der Binärdarstellung einer Zahl im Speicher an erster/letzter Stelle steht, hängt von der Architektur ab...
 - Big-endian: kleinstes Bit zuerst
 - Little-endian: grösstes Bit zuerst (z.B. Intel)
 - Stellt auch für einfache Arrays ein Problem dar...
- Datenstrukturen (**struct**) enthalten Padding (leere Speicherstellen)
 - Paddinggrößen sind architekturnspezifisch... **struct** binär speichern kann problematisch sein... ⇒ 06_04_sizeof_struct.c

```
struct test {  
    char zeichen;  
    int zahl;  
    double gleitzahl;  
};
```

`sizeof(test) != sizeof(char) + sizeof(int) + sizeof(double)`

Portables binäres I/O benötigt also mehr Arbeit...

Dateien Byte-weise auslesen / schreiben

Vorlesung 6

Manchmal praktisch, einen Stream Zeichen für Zeichen zu lesen oder zu schreiben.
(Vorsicht: oft langsamer als in Blöcken).

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

Rückgabewerte

- `fgetc`: Zeichen als `(int)(unsigned char)`, `-1` bei *end of file* oder Fehler.
- `fputc`: wie `fgetc`

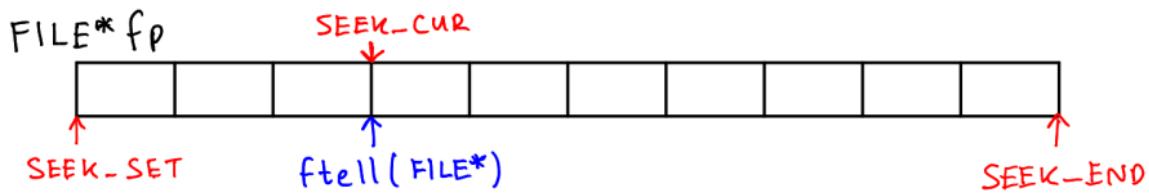
⇒ 06_05_test_fgetc_fputc.c

Den Dateicursor bewegen mit fseek

Jede Lese- und Schreiboperation bewegt den Dateicursor des Streams. Um diesen Dateicursor selbst zu bewegen, wird fseek genutzt.

Der Cursor bewegt sich offset Bytes ab Position: $\text{origin} \in \{\text{SEEK_SET}, \text{SEEK_CUR}, \text{SEEK_END}\}$.

```
int fseek(FILE *stream, long offset, int origin);
```



Um herauszufinden, wo der Dateicursor sich gerade befindet, kann man ftell benutzen.

```
long ftell(FILE *stream);
```

⇒ 06_06_test_fseek.c

Zeilenweise Datei auslesen: `getline`

Vorlesung 6

Mit dem POSIX-Standard 2008 wurde die Funktion

```
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

eingeführt. Mit ihr kann man relativ sicher Dateien Zeile für Zeile (also von Zeilenumbruch bis Zeilenumbruch) auslesen.

- `lineptr`: die Adresse eines Pointers auf `char` (kann als `NULL` übergeben werden)
- `n`: Adresse einer Zählervariable des Typs `size_t`
- `stream`: ein Dateistream
- `getline` kümmert sich selbstständig um die Speicherverwaltung für den Pufferspeicher `lineptr`.
- Beim letzten Aufruf muss man `lineptr` freigeben.

⇒ 06_07_getline.c

Vorschau auf Vorlesung 7

- Algorithmische Komplexität
- Sind Arrays für bestimmte Aufgaben die beste Datenstruktur?
- Doppelt verkettete Listen
- Funktionenpointer
- Komplexe Zahlen

Algorithmische Komplexität und *Big-O*-Notation

Vorlesung 7

- Die *algorithmische Zeitkomplexität* ist ein Maß für die Anzahl an Rechenschritten, die ein Algorithmus benötigt, um für n Datenelemente ein Ergebnis zu liefern.
- Wir nutzen die *asymptotische Zeitkomplexität*, also wenn n sehr groß ist, um Algorithmen miteinander zu vergleichen.
- Sortieralgorithmen sind ausgezeichnete Beispiele, an denen man Komplexität verstehen lernen kann.
- Auch die Wahl der Datenstruktur kann für bestimmte Aufgaben einen Einfluss auf die Gesamtkomplexität haben.

Zeitkomplexität von Einfügensortieren

Vorlesung 7

Betrachten wir zunächst die Kosten von Einfügensortieren aus Vorlesung 1.

Algorithmus 2 Einfügensortieren

Ausführungen Kostengrund

Input: Lists U, S

Output: List S

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

c_1	n	Zählervariable
c_2	n	Zuweisung
c_3	n	Zuweisung
c_4	$\sum_{i=1}^{n-1} t_i$	Vergleich
c_5	$\sum_{i=1}^{n-1} t_i$	Vergleich
c_6	$\sum_{i=1}^{n-1} (t_i - 1)$	Zuweisungen
—	—	—
—	—	—
—	—	—
—	—	—

t_i sind situationsabhängig und die Kosten steigen (natürlich) mit wachsendem n , aber wie genau?

Zeitkomplexität von Einfügensortieren

Vorlesung 7

Die Gesamtkosten, $T(n)$, ergeben sich also aus:

$$T(n) = (c_1 + c_2 + c_3) \cdot (n) + (c_4 + c_5) \cdot \sum_{i=1}^{n-1} t_i + (c_6) \cdot \sum_{i=1}^{n-1} (t_i - 1)$$

Im schlimmsten Fall ist $t_i = (i + 1)$ und man erhält

$$\sum_{i=1}^{n-1} (i + 1) = \frac{n(n + 1)}{2} - 1$$

$$\sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}$$

Im Limes $n \rightarrow \infty$ dominieren die letzten beiden Terme und wir sehen:

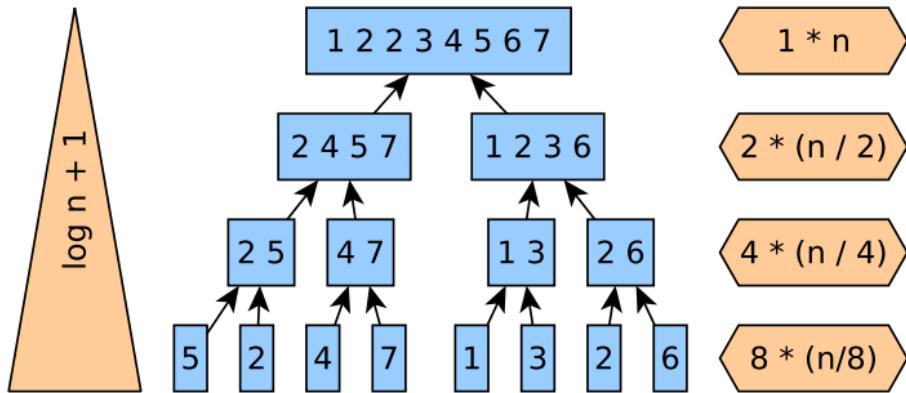
Einfügensortieren ist ein $O(n^2)$ Algorithmus. Es gibt ein $k > 0$, für das gilt:

$$0 \leq T(n) \leq kf(n^2)$$

und $T(n)$ ist nach oben durch $kf(n^2)$ beschränkt. Untere Schranke?

Zeitkomplexität von Mergesort

Vorlesung 7



- Wir teilen $\log_2 n$ mal auf (also $\log_2 n + 1$ Ebenen)
- Jeder Aufruf benötigt cn Rechenschritte

$$T(n) = cn \cdot (\log_2 n + 1) \rightarrow O(n \log_2 n)$$

Im Gegensatz zu Einfügensortieren ist Mergesort also ein $O(n \log_2 n)$ -Algorithmus

Zeitkomplexität von Mergesort ganz konkret

Vorlesung 7

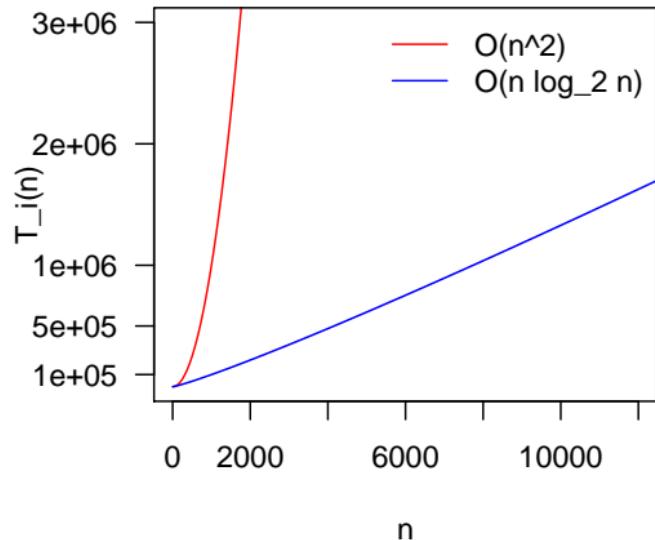
Wir haben einen Algorithmus mit $T_1(n) = c_1 n^2$ und einen Algorithmus mit $T_2(n) = c_2 n \log_2 n$.

- Nehmen wir an, dass $c_2 = 10 \cdot c_1$: die $O(n^2)$ Implementierung könnte rechnerisch einfacher sein und pro Schritt, 10 mal weniger Zeit brauchen.
- Für $n = 10000$ haben wir:

$$T_1(10^4) = c_1 \cdot 10^8$$

$$\begin{aligned} T_2(10^4) &= c_2 \cdot 10^4 \cdot \log_2 10^4 \\ &= 10 \cdot c_1 \cdot 10^4 \cdot 13.3 \sim 10^6 \cdot c_1 \end{aligned}$$

- Der Unterschied zwischen $O(n \log_2 n)$ und $O(n^2)$ ist gigantisch.
⇒ Wenn man einen besseren Algorithmus benutzen kann, ist dies fast immer besser, als einen schlechteren Algorithmus durch Optimierung schneller zu machen...



Kategorisierung von Algorithmen

Vorlesung 7

Man kann Algorithmen allgemein nach ihrer asymptotischen Laufzeit kategorisieren. Die O -Notation ist dabei nicht die einzige Möglichkeit:

- O -Notation: obere Schranke für die “worst-case” Laufzeit → der Algorithmus wird nie schlechter skalieren als

$$0 \leq T(n) \leq kf(n)$$

- Θ -Notation: obere und untere Schranken im asymptotischen Limes → das Skalierverhalten der Laufzeit ist

$$k_1f(n) \leq T(n) \leq k_2f(n)$$

- Ω -Notation: untere Schranke im asymptotischen Limes → die Laufzeit skaliert mindestens wie

$$kf(n) \leq T(n)$$

Komplexität von Operationen auf Datenstrukturen

Vorlesung 7

Zielsetzung

Wir wollen aus einem Array ein Element entfernen oder in ein Array ein Element einfügen.

Einfügen bei Index i in ein Array der Größe n

- Array vergrößern → `realloc` eventuell mit vollständiger Kopie → $O(n)$
- Ab Index i , Elemente um 1 Stelle nach rechts verschieben → $O(n)$
- Zielelement bei Index i einfügen → $O(1)$

Schlimmstensfalls also $O(n)!$

Elemente bei Index i entfernen aus einem Array der Größe n

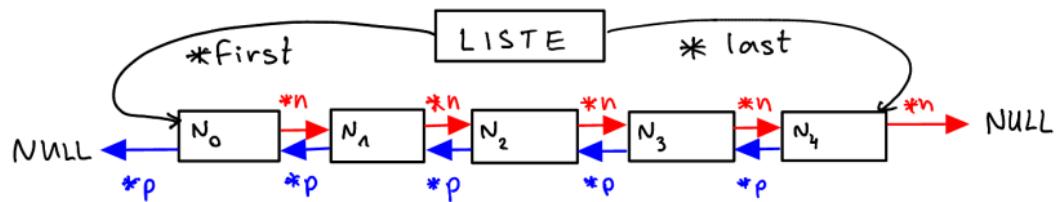
- Ab Index $i + 1$, Elemente um 1 Stelle nach links verschieben → $O(n)$
- (eventuell `realloc` um Speicher freizugeben)

Schlimmstenfalls also auch $O(n)!$

Anwendung: Doppelt verkettete Liste

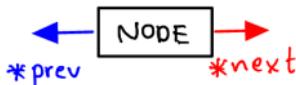
Vorlesung 7

Algorithmen, welche oft Elemente einfügen oder entfernen müssen, werden durch Arrays langsam... (besonders Arrays von komplexen Datenstrukturen)



Lösungsansatz: Doppelt verkettete Liste

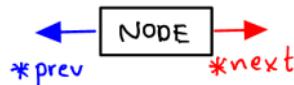
Mit einer doppelt verketteten Liste, müssen nur jeweils zwei Zeiger gesetzt werden:
Einfügen und Entfernen sind $O(1)$! \Rightarrow Unabhängig von der Größe der Liste, man sagt auch "konstante Zeit".



- Als Listenelement dient eine Datenstruktur mit zwei Zeigern und Speicherplatz für die eigentlichen Daten.

Doppelt verkettete Liste: Listenelemente

Vorlesung 7



dll.h

```
#ifndef DLL_H
#define DLL_H

typedef struct dll_node_t {
    struct dll_node_t *next;
    struct dll_node_t *prev;
    double             data;
} dll_node_t;

typedef struct dll_t {
    dll_node_t *first;
    dll_node_t *last;
} dll_t;
```

```
dll_t *dll_create();

void   dll_free(dll_t *list);

dll_node_t*
dll_insert( dll_t *list,
            dll_node_t *cursor,
            double       data );

void
dll_delete( dll_t *list,
            dll_node_t *delnode);

#endif // ifdef(DLL_H)
```

Doppelt verkettete Liste durchlaufen

Vorlesung 7

In Übung 7 werden Sie eine doppelt verkettete Liste implementieren. Um eine solche Liste zu durchlaufen, kann man folgendes Konstrukt nutzen.

```
dll_t* list = dll_create(); /* erstellen und dann fuellen ... */  
/* vorwaerts oder rueckwaerts durchlaufen */  
for( dll_node_t *ptr = list->first; ptr != NULL; ptr = ptr->next ){}  
for( dll_node_t *ptr = list->last; ptr != NULL; ptr = ptr->prev ){}
```

Es gibt viele weitere Datenstrukturen die vom Prinzip her der ähnlich sind.

- Einfach verkettete Listen/Ringe: kann nur in eine Richtung durchlaufen werden.
- Stapel: Elemente werden aufgestapelt und von oben her abgearbeitet
- Schlange: Elemente werden hinten angestellt und vorne abgearbeitet
- Drei- und n -fach verkettete Listen: für Baumstrukturen und Graphen

Doppelt verkettete Liste: Nachteile?

Vorlesung 7

- Welche Nachteile hat eine doppelt verkettete Liste?

Doppelt verkettete Liste: Nachteile

Vorlesung 7

- Welche Nachteile hat eine doppelt verkettete Liste?
- ⇒ Wenn wir das n te Element wollen, müssen wir durch die ganze Liste laufen und abzählen, bis wir n erreicht haben.
- ! Eine solche Liste kann also nur sequentiell effizient durchlaufen werden. Springender Zugriff hat Kosten $O(n)$...

Funktionenpointer

Vorlesung 7

Ein Ziel ist es, möglichst generisch zu programmieren: eine Funktion soll viele Zwecke erfüllen können.

Beispiel

Wir haben eine Funktion ableitung_cos, welche eine Annäherung an die Ableitung des Cosinus berechnet.

```
double ableitung_cos(double const x, double const delta){  
    return (cos(x+delta)-cos(x))/delta;  
}
```

Müssen wir hunderte praktisch identischer Funktionen für alle möglichen Ableitungen implementieren?

Ein Funktionenpointer ist ein Zeiger auf eine Funktion, ein Datentyp also, der für die komplette Signatur der Funktion einstehen kann.

Funktionenpointer: Deklaration

Vorlesung 7

Der Funktionenpointer hat als "Datentyp" die vollständige Signatur der Funktion, auf die er zeigen soll.

```
ret_t (*fp)(arg1_t, arg2_t)
```

⇒ fp ist der Variablename eines Zeigers auf eine Funktion mit Rückgabewert ret_t und Argumententypen arg1_t, arg2_t (es können natürlich beliebig viele Argumente sein)

Gibt es eine Funktion mit dieser Signatur, kann man fp auf diese Funktion zeigen lassen.

```
ret_t func(arg1_t a1, arg2_t a2){ .... }  
fp = func;  
// oder  
fp = &func; // (gleichwertige Zuweisung, Adressoperator muss nicht geschrieben werden)
```

Jetzt kann func über fp aufgerufen werden, aber was bringt das?

Funktionenpointer: Anwendung

Vorlesung 7

Wir können jetzt die Ableitungsfunktion verallgemeinern.

```
double ableitung(double const x, double const delta,
                  double (*fp)(double) ){
    return ((*fp)(x+delta)-(*fp)(x))/delta;
}

double abl_wert = ableitung( 1.2, 0.0001, cos );
```

Funktionenpointer können auch auf Gleichheit geprüft werden. Insbesondere könnte man überprüfen, ob ein Funktionspointer auf eine bestimmte Funktion zeigt.

```
double ableitung(double const x, double const epsilon,
                  double (*fp)(double) ){
    if( fp == tan ){
        /* ohje, das wird schwierig... */
    }
    return ((*fp)(x+epsilon)-(*fp)(x))/epsilon;
}
```

Funktionenpointer

Vorlesung 7

```
unser_struct_t erste_funktion(double, int, int, char, double);
unser_struct_t zweite_funktion(double, int, int, char, double);
unser_struct_t (*fp)(double, int, int, char, double);

double void_funktion(void);
double (*void_fp)(void);

unser_struct_t ergebnis, ergebnis2;

fp = erste_funktion;
ergebnis = (*fp)( arg0, arg1, arg2, arg3, arg4 ); // ruft erste_funktion auf

fp = zweite_funktion;
ergebnis2 = (*fp)( arg0, arg1, arg2, arg3, arg4 ); // ruft zweite_funktion auf

fp = void_funktion; // Warnung, aber kein Fehler...

void_fp = void_funktion;
double x = (*void_fp)();
```

Der Funktionenpointer erlaubt es, den Aufruf einer Funktion situationsabhängig zu machen. Achtung: Funktionenpointer sind tückisch und fehleranfällig...

⇒ (07_[00,01]_void_fpinter.c)

Komplexe Zahlen

Vorlesung 7

Seit C99 gibt es in C einen Datentypen für komplexe Zahlen. Es gibt davon zwei Ausführungen mit unterschiedlicher Fließkommagenaugigkeit.

```
#include <complex.h>
double _Complex z = 3.0 + 2.1*I;
float _Complex w = 4.2 + 4.5*I;
```

Das Symbol I entspricht hier $\sqrt{-1}$.

```
double _Complex x = z * w;
```

Arithmetische Operationen sind auf dem `_Complex` Datentypen wie gewohnt definiert.

Komplexe Zahlen

Vorlesung 7

Zugriff auf Real- und Imaginärteil sowie Betrag und Argument erfolgen über die Funktionen:

```
double creal(const _Complex double); // Realteil  
double cimag(const _Complex double); // Imaginaerteil  
double cabs(const _Complex double); // Betrag  
double carg(const _Complex double); // Argument (Bogenmass)  
_Complex double conj(const _Complex double); // Konjugierte
```

Ausgabe mit [s,f]printf erfolgt also in zwei Teilen:

```
printf("z = %f + i %f\n", creal(z), cimag(z));
```

Weitere Funktionen:

- cexp, clog, csqrt, cpow
- csin, ccos, ctan, casin, cacos, catan
- csinh, ccosh, ctanh, casinh, cacosh, catanh

Vorschau Vorlesung 8

- Verwendung Externer Bibliotheken
- Makefiles

- Quicksort aus der C Standardbibliothek verwenden
- Externe Bibliotheken einbinden
- Makefiles
- Sicher programmieren mit *Assertions*
- Zuverlässige Zeitmessung

Quicksort aus der C-Standardbibliothek

Vorlesung 8

- Haben gestern Sortieralgorithmen verglichen und in den Übungen insgesamt drei implementiert.
- Muss man jetzt für jeden Datentyp den Sortieralgorithmus neu schreiben?

⇒ *Quicksort (qsort)* ist in der C-Standardbibliothek vorhanden.

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(void const *, void const *));
```

- base: void-Zeiger auf zu sortierende Daten
- nmemb: Anzahl Datenelemente
- size: Größe (in Bytes) pro Datenelement
- compar: Funktion, welche die Vergleiche durchführt

Quicksort aus der C-Standardbibliothek

Vorlesung 8

```
int compar( const void *links, const void *rechts){ ... }
```

- Wir müssen eine Funktion schreiben, die zwei Zeiger auf Elemente aus dem base-Array erhält und dann die Vergleiche macht.
- Was auch immer der Datentyp ist, den wir vergleichen, wir wollen dafür eine Sortierfolge festlegen (also, ein “größer als”, “kleiner als”)
- Rückgabewert:
 - < 0 wenn links kleiner als rechts interpretiert wird
 - == 0 wenn links gleich groß als rechts interpretiert wird
 - > 0 wenn links größer als rechts interpretiert wird
- Wir können die Sortierfunktion jederzeit ändern!

⇒ 08_01/08_01_quicksort.c

Externe Bibliotheken

Vorlesung 8

Einbinden und Kompilieren

Eine "externe" Bibliothek hatten wir schon kurz kennengelernt und genutzt:
`math.h`.

```
#include <math.h>
```

Um auf die Funktionen aus der Bibliothek dann auch Zugriff zu erhalten, muss diese auch eingelinkt werden:

```
gcc -o programm main.o modul1.o modul2.o -lm
```

Das Kürzel `-l` bedeutet: suche in den Systembibliothekspfaden nach einer Bibliothek Namens `m`. Oftmals müssen auch weitere Bibliothekspfade angegeben werden. Dies wird mit `-Lpfad` erreicht, wobei `pfad` durch den den zu verwendenden Pfad ersetzt wird.

Zufallszahlen mit GSL

Vorlesung 8

In Übung 5 wurde ein Pseudozufallszahlengenerator entwickelt und mit Conway's Game of Life getestet. Den Standardgenerator in C sollte man nicht benutzen. Gute Alternativen finden sich in der GNU Scientific Library (GSL).

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
int main (void) {
    const unsigned int n = 10;
    const gsl_rng_type * generator_type;
    gsl_rng * r;
    gsl_rng_env_setup();
    generator_type = gsl_rng_ranlxd2;
    r = gsl_rng_alloc(generator_type);
    gsl_rng_set(r, 12345);
    double u = 0.0;
    for(unsigned int i = 0; i < n; i++) {
        u = gsl_rng_uniform (r);
        printf ("% .5f\n", u);
    }
    gsl_rng_free(r);
    return 0;
}
```

Kompilieren mit externen Bibliotheken

Vorlesung 8

Wie kompilieren wir jetzt?

- Zunächst muss gcc (als Compiler [-c]), die Headerdateien für die Bibliotheken finden.
 - `gcc -c -Ipfad`
 - Das `-I` steht für "include path"
- Beim Linken muss gcc die Bibliothekspfade kennen, und wir müssen die notwendigen Bibliotheken einbinden.
 - `gcc -Lpfad -lbib`
 - `-l` und `-L` stehen für "library" und "library path"
 - Bibliotheken haben auf UNIX-Systemen Dateinamen der Form: `lib[...]`, hier also `libgsl.a` und `libgsl.so`

```
$ gcc -c -I/usr/include/ 08_02_test_gsl_rng.c
$ gcc -o 08_02_test_gsl_rng 08_02_test_gsl_rng.o -lgsl -lgslcblas -L/usr/lib/
x86_64-linux-gnu/
```

Die zweite eingebundene Bibliothek erklären wir gleich.

⇒ `08_02_test_gsl_rng.c` und `compile_08_02.sh`

Einfache Makefiles

Vorlesung 8

Ein Makefile ist eine Datei in der die Regeln zum Komplizieren eines Programms beschrieben sind. Die genaue Struktur ist dabei relativ offen, da es sich nur um Regeln zur Erstellung von Dateien und deren Abhängigkeiten handelt.

```
regel: abhaenigkeit1 abhaengigkeit2 abhaengikeit3  
kommando1  
kommando2
```

- Um `regel` (auch `target` oder `Ziel` genannt) zu erstellen, überprüft `make` zunächst ob die Abhängigkeiten existieren. Wenn ja, werden die Kommandos ausgeführt. Wenn nicht, wird versucht Regeln zu finden um die Abhängigkeiten zu erstellen.
- Ist eine Abhängigkeit, oder eine Abhängigkeit einer Abhängigkeit (z.B. der C-Quelltext zu einem Modul) neuer als das momentan vorhandene Ziel, wird das Ziel erneut kompiliert.
- Target und Abhängigkeiten sind durch einen Doppelpunkt getrennt. In der nächsten Zeile ist ein Tabulator (wichtig, keine einfachen Leerzeichen!)

Makefiles: Beispiel 0

Vorlesung 8

```
programm: modul1.o modul2.o programm.o
          gcc -o programm modul1.o modul2.o programm.o

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

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

programm.o: programm.c modul1.h modul2.h
          gcc -std=c99 -Wall -Wpedantic -c -o programm.o programm.c
```

Hier werden für jedes Ziel einzeln, alle Regeln manuell erstellt und make führt die Kommandos dann einfach aus. Man hätte natürlich auch einfach ein Shellskript schreiben können. Man gewinnt lediglich, dass nur jene Teile kompiliert werden, die auch wirklich kompiliert werden müssen ⇒ Verallgemeinerungen machen Makefiles erst mächtig.

⇒ makefiles/beispiel0

Makefiles: Beispiel 1

Vorlesung 8

Auf Variablen wird in einem Makefile mit einem Dollarzeichen zugegriffen. Es gibt einige spezielle Variablen, aber man kann natürlich selbst welche definieren.

```
test_printf: test_printf.c  
    gcc -std=c99 -Wall -Wpedantic -o $@ $<
```

- \$@ ist ein Platzhalter für den Namen des Targets. (hier `test_printf`)
- \$< steht für die erste Abhängigkeit von links. (hier `test_printf.c`)

Kompliert wird jetzt mit:

```
$ make
```

make wird das kommando

```
gcc -std=c99 -Wall -Wpedantic -o test_printf test_printf.c
```

aufrufen und die ausführbare Datei `test_printf` wird dadurch erzeugt werden.

⇒ `makefiles/beispiel1`

Makefiles: Beispiel 2

Vorlesung 8

```
# Regeln fuer main-Objektdateien
programm1.o: programm1.c modul1.h modul2.h
    gcc -std=c99 -Wall -Wpedantic -c $<

programm2.o: programm2.c modul2.h modul3.h
    gcc -std=c99 -Wall -Wpedantic -c $<

# Regel fuer die Module
%.o : %.c %.h
    gcc -std=c99 -Wall -Wpedantic -c $<

programm1: programm1.o modul1.o modul2.o
    gcc -std=c99 -Wall -Wpedantic -o $@ $^

programm2: programm2.o modul2.o modul3.o
    gcc -std=c99 -Wall -Wpedantic -o $@ $^
```

\$^ ist ein Kürzel für alle Abhängigkeiten. Das Kommando:

```
$ make programm1
```

wird erst modul1.o, modul2.o und programm1.o kompilieren und dann den Linker aufrufen.

Makefiles: Beispiel 3

Vorlesung 8

Für die ganzen Beispielprogramme, die in der Regel aus einer Datei bestehen, habe ich folgendes Makefile genutzt.

```
SOURCES := $(wildcard *.c)

all: $(basename $(SOURCES))

$(basename $(SOURCES)): % : %.c Makefile
    gcc -g -ggdb -Wall -Wpedantic -std=c99 -o $@ $<

.PHONY: clean
clean:
    rm $(basename $(SOURCES))
```

- make führt, wenn es ohne Kommandozeilenargumente aufgerufen wird, die erste Regel aus. Das ist hier `all`. Es werden also alle Programme erzeugt, deren C-Dateien sich in der Variable `$(SOURCES)` wiederfinden.

Makefiles: Einfache Substitutionen

Vorlesung 8

```
SOURCES := $(wildcard *.c)

all: $(basename $(SOURCES))

$(basename $(SOURCES)): % : %.c Makefile
    gcc -g -ggdb -Wall -Wpedantic -std=c99 -o $@ $<

.PHONY: clean
clean:
    rm $(basename $(SOURCES))
```

- Mit “VARIABELNAME :=” werden eigene Variablen definiert.
- \$(wildcard *.c) sucht im momentanen Verzeichnis nach allen C-Dateien. Die Dateinamen werden in der Variable \$(SOURCES) abgelegt.
- \$(basename \$(SOURCES)) entfernt alle Dateiendungen der Dateinamen in der Variable \$(SOURCES), hier .c

`$(basename programm1.c programm2.c) -> programm1 programm2`
⇒ (makefiles/beispiel3)

Makefiles: Das Prozentzeichen

Vorlesung 8

Das Prozentzeichen erlaubt es, Regeln zu automatisieren oder zu verketten.

```
%.o : %.c %.h  
        gcc -std=c99 -Wall -Wpedantic -c $<
```

Für jede Datei des Typs “name.o” wird eine Abhängigkeit von name.c und name.h aufgestellt. name.o wird dann mit der Regel

```
gcc -std=c99 -Wall -Wpedantic -c $<
```

kompiliert.

```
$(basename $(SOURCES)) : % : %.c Makefile  
        gcc -g -ggdb -Wall -Wpedantic -std=c99 -o $@ $<
```

- Für jedes Objekt in \$(basename \$(SOURCES)) wird eine neue Regel erstellt (erstes %)
- Für diese neuen Regeln werden Abhängigkeiten der Form name.c erstellt und das Kompilierkommando entsprechend ausgeführt.

Makefiles: *phony* targets und aufräumen

Vorlesung 8

```
.PHONY: clean  
clean:  
    rm $(basename $(SOURCES))
```

- Ein *phony* target kann immer ausgeführt werden, es wird nicht überprüft ob die Datei `clean` schon existiert. Hier wird ja auch gar keine Datei erzeugt!
- Es ist konventionell, ein solches target mit dem Namen `clean` zu haben, welches alle generierten Dateien aufräumt.

Um ein komplexeres Projekt aus mehreren Modulen mit Einbindung von Bibliotheken und, unter Umständen, mehreren ausführbaren Programmen zu kompilieren, kann uns dieses Beispiel dienen.

⇒ nächste Folie

Makefiles: Beispiel 4

Vorlesung 8

```
CC := gcc -std=c99 -Wall -Wpedantic
LD := gcc -std=c99 -Wall -Wpedantic
SOURCES := $(wildcard *.c)
HEADERS := $(wildcard *.h)
MODULES := $(patsubst %.c, %.o, $(SOURCES))
PROGRAMS := teilchen
MODULES := $(filter-out $(addsuffix .o, $(PROGRAMS)), $(MODULES))
LIBS := -lm -lgsl -lgslcblas

all: $(PROGRAMS)

$(addsuffix .o, $(PROGRAMS)): % : $(addsuffix .c, $(PROGRAMS)) $(HEADERS) Makefile
    $(CC) -c $(patsubst %.o, %.c, $@)

%.o: %.c %.h Makefile
    $(CC) -c $<

$(PROGRAMS): % : %.o $(MODULES) Makefile
    $(LD) -o $@ $< $(MODULES) $(LIBS)

.PHONY: clean
clean:
    rm -f $(MODULES) $(PROGRAMS) $(addsuffix .o, $(PROGRAMS))
```

⇒ (makefiles/beispiel[4,5]) haben ausführliche Kommentare, die jede Zeile nochmal erläutern.

- Makefiles Handreferenz (sehr praktisch!): <https://goo.gl/kizu5D>

Makefiles und Build-systeme

Vorlesung 8

- Ein großes Softwarepaket kann aus tausenden Quelldateien bestehen und muss auf verschiedenen Architekturen übersetzt werden.
- Bibliothekenpfade oder Abhängigkeiten können vom System abhängen.
- Dazu gibt es Build-systeme wie CMake oder Autotools, die (unter Anleitung) "automatisch" Makefiles für ein Programm erstellen.
- Viel zu umfangreich für eine Vorlesung...

CMake intro (sehr kurz): <https://goo.gl/ugbG8K>

CMake INTRO (50 Folien): <https://goo.gl/nMFw5>

CMake Orientierung (gut!): <https://goo.gl/kJmv3C>

Autotools Intro (ziemlich gut): <https://goo.gl/SNGujC>

Abbruchbedingungen mit *assertions*

Vorlesung 8

- Assertions (Zusicherungen, Sicherstellungen) sind ein Konstrukt, welches es erlaubt, gut lesbare Abbruchkriterien für Ausnahmesituationen zu definieren.
- Wenn eine Bedingung nicht gegeben ist, an dieser Stelle im Code aber auf jeden Fall gegeben sein sollte (nicht-NUL Argumente in einer Funktion, z.B.), kann man assert nutzen, um das Programm dort abzubrechen.
- Können mit gcc -DNDEBUG [...] ausgeschaltet werden.

```
#include <cassert.h>
int func(FILE* fp){
    assert( fp != NULL );
    /* ... */
    return 0;
}
int main(void){
    FILE* fp = fopen("gibtbesnicht.txt","r");
    func(fp);
}
```

⇒ (08_03_test_assert.c)

Zuverlässige Zeitmessung mit gettimeofday

Vorlesung 8

Zeitmessung mit `clock()` ist sehr unzuverlässig. `gettimeofday` wird nicht von anderen Programmen beeinflusst und hat eine Präzision von μs .

```
#include <sys/time.h>
struct timeval anfang, ende;
gettimeofday(&anfang, NULL);
/* Zeit vergeht durch schwierige Rechnungen */
gettimeofday(&ende, NULL);
double sec = (double)(ende.tv_sec-anfang.tv_sec);
double usec = (double)(ende.tv_usec-anfang.tv_usec);
printf("Es sind %f Sekunden vergangen\n", sec+1.0e-6*usec);
```

⇒ (08_04_test_gettimeofday.c)

Das `struct timeval` ist wie folgt definiert:

```
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};
```

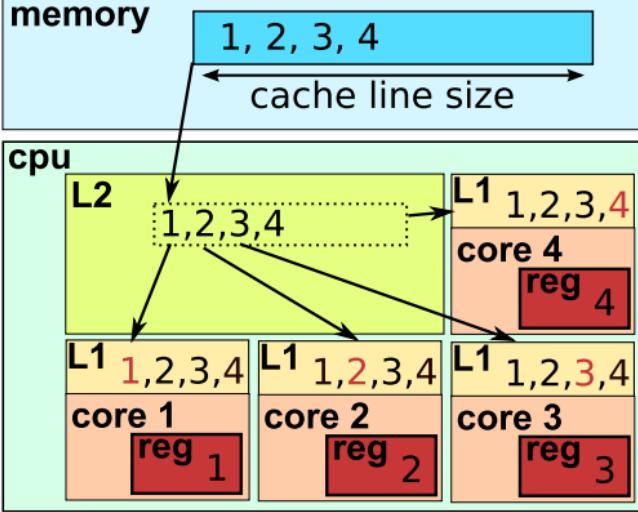
Vorschau auf Vorlesung 9

Mal sehen, was wir alles schaffen:

- Nadelöhr verstehen mit gprof (Performancetuning)
- Optimierungsflaggen des Compilers
- OpenMP-Parallelisierung (multi-threading, Verteilung auf mehrere Threads)
- MPI-Parallelisierung (Verteilung auf mehrere Prozesse)

Der Aufbau einer CPU

Vorlesung 9



Algorithmus ist beschränkt durch

- Ausführbare Rechenschritte pro Zeiteinheit der CPU
- Bandbreite:
Speicher → Cache → Register
- Parallelisierbarkeit des Problems

- Jedes Core hat eine begrenzte Anzahl an Registern, auf denen arithmetische Operationen ausgeführt werden.
- Jedes Core hat einen kleinen aber schnellen *Level 1 cache*
- Die Cores teilen sich einen etwas größeren aber langsameren *Level 2 cache*
- Unter Umständen noch ein *Level 3 cache* vorhanden
- Daten werden aus dem Speicher in Blöcken, der *cache line* in die Hierarchie von Caches bis zu den Registern geladen

Ein einfaches numerisches Problem

Vorlesung 9

Die Matrix M hat Z Zeilen und S Spalten. Der Vektor \vec{y} hat Z Elemente, der Vektor \vec{x} hat S . Wir werden anhand der Matrix-Vektor Multiplikation

$$y_i = \sum_{j=1}^S M_{ij}x_j$$

und der Vektornorm

$$d = (\vec{y}, \vec{y}) = \sum_{i=1}^Z y_i \cdot y_i$$

sehen, wie man die Ausführgeschwindigkeit eines numerischen Programms analysiert und verbessert.

- Die Matrix-Vektor Multiplikation benötigt:
 - pro Zeile S Multiplikationen und $S - 1$ Additionen
 - insgesamt also $Z \cdot (2S - 1)$ arithmetische Operationen.
- Die Normberechnung benötigt
 - Z Multiplikationen und $Z - 1$ Additionen $\rightarrow 2Z - 1$ Operationen

Matrix-Vektor Multiplikation

Vorlesung 9

Eine Implementierung der Matrix-Vektor Multiplikation könnte so aussehen:

```
for( int i_z = 0; i_z < Z; ++i_z ){
    for( int i_s = 0; i_s < S; ++i_s ){
        y[i_z] += M[i_z][i_s] * x[i_s];
    }
}
```

Floating Point Operarations und Speicherzugriffe - FLOPs, LOADs und STOREs

- FLOPs: 2 arithmetische Operationen (Multiplikation und Akkumulation)
 - LOADs: 24 Bytes ($M[i_z][i_s]$, $x[i_s]$ und $y[i_z]$)
 - STOREs: 8 Bytes ($y[i_z]$)
-
- $x[i_s]$ wird mehrmals verwendet \Rightarrow Effizienzgewinn durch Caches
 - $y[i_z]$ wird (meist) in einem Register verbleiben

Vektornorm

Vorlesung 9

Und die der Vektornorm so:

```
double summe = 0.0;
for( int i_s = 0; i_s < s; ++i_s ){
    summe += x[i_s] * x[i_s];
}
```

Floating Point Operarations und Speicherzugriffe - FLOPs, LOADs und STOREs

- FLOPs: 2 arithmetische Operationen (Multiplikation und Akkumulation)
 - LOADs: 16 Bytes ($x[i_s]$ und summe)
 - STOREs: 8 Bytes (summe)
-
- $x[i_s]$ wird nur ein Mal verwendet \Rightarrow Performance beschränkt durch Speicherbandbreite
 - summe wird bis zum Ende der Schleife in einem Register verbleiben
- \Rightarrow Benchmarkcode: 09_01_plain

Nadelöhre verstehen mit gprof

Vorlesung 9

gprof (GNU profiler) ist ein Programm, mit dem man verstehen kann, in welchen Teilen eines anderen Programms am meisten Zeit verbracht wird.

- Zunächst muss man anpassen, wie man kompiliert.

```
compile: gcc -c [...] -> gcc -c -pg [...]  
link:     gcc -o [...] -> gcc -pg -o [...]
```

- Dann wird das Programm ganz normal ausgeführt. gprof schreibt dann die Datei `gmon.out` ins momentane Verzeichnis.
- Jetzt wird mit gprof analysiert.

```
$ gprof name_des_programms
```

- Besonders bei komplizierten Programmen sieht man, welche Funktionen am dringsten optimiert werden müssen. Man spricht von **hotspots**.

gprof Ausgabe

Vorlesung 9

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
99.74	3.62	3.62	200	18.10	18.10	mult_Mv
0.55	3.64	0.02	1	20.06	20.06	matrix_rand
0.00	3.64	0.00	409	0.00	0.00	fatal_error
0.00	3.64	0.00	100	0.00	0.00	sq_norm
[...]						

Spalten von links nach rechts

- Relativer Anteil an der Gesamtzeit, verbracht in der Funktion mit Namen name
- Aufsummierte Zeit in allen bisher aufgeführten Funktionen
- Absolute Zeit in dieser Funktion
- Anzahl Aufrufe dieser Funktion
- Millisekunden pro Aufruf im Inneren der Funktion selbst
- Millisekunden pro Aufruf im Inneren der Funktion **und** der darin aufgerufenen Funktionen
- Name der Funktion

⇒ mal ausprobieren!

Erste Optimierung: Compilerflaggen

Vorlesung 9

Compiler können versuchen, bestimmte Optimierungen vorzunehmen. Man übergibt hierzu entsprechende Kommandozeilenargumente.

- Optimierungslevel hochschrauben `-Ox` (großes O, keine Null)

```
gcc -c -O3
```

- Architektspezifische Optimierungen einschalten `-mtune=Architektur`
`-march=Architektur`. Auf meinem Rechner:

```
gcc -c -O3 -mtune=broadwell -march=broadwell -mfma
```

- Info dazu mit

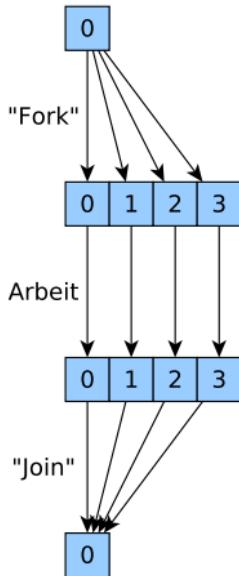
```
$ cat /proc/cpuinfo | less
```

⇒ Makefile in `09_02_compiler_flags`

Zweite Optimierung: Multi-threading mit OpenMP

Vorlesung 9

OpenMP ist ein Standard, welcher es erlaubt, Rechenarbeit mithilfe von *Threads* auf mehrere Rechenkerne zu verteilen.



- 1 Programm startet ganz normal auf einem Rechenkern.
 - 2 Es findet sich parallelisierbare Arbeit → *fork* verteilt Arbeit auf *Threads*
 - 3 Threads rechnen unabhängig voneinander.
 - 4 Threads werden wieder zusammengeführt (*join*). Unter Umständen müssen teure Synchronisierungen getätigkt werden.
- **Achtung:** Threads können sich bei bestimmten Operationen in die Quere kommen, überschreiben sich gegenseitig Speicherstellen → *race conditions* → falsche Ergebnisse!

OpenMP Blitz-Intro 1/2

Vorlesung 9

Wir wollen Arbeit in einer Schleife über mehrere Threads verteilen.

```
#pragma omp parallel for
for(int i_z = 0; i_z < Z; ++i_z ){
    for(int i_s = 0; i_s < S; ++i_s ){
        y[i_z] += M[i_z][i_s] * x[i_s];
    }
}
```

- Abzählbereich $[0, Z - 1]$ wird in N_{threads} Teile aufgeteilt.
- Jeder Thread führt einen Teil der Schleife durch.
- Die Iterationen der Schleife müssen unabhängig sein.

```
x[0] = 2.3;
// geht nicht: Abhängigkeit zwischen Iterationen!
#pragma omp parallel for
for(int i_z = 1; i_z < Z; ++i_z ){
    x[i_z] = funktion(x[i_z-1]);
}
```

OpenMP Blitz-Intro 2/2

Vorlesung 9

Bei der Vektornorm müssen wir aufpassen!

```
double summe = 0.0;
#pragma omp parallel for reduction(+:summe)
for(int i_s = 0; i_s < S; ++i_s ){
    summe += x[i_s] * x[i_s];
}
```

- Alle Threads schreiben in die gleiche Variable `summe`.
- `reduction(+:summe)` gibt an, dass jeder Thread eine eigene solche Variable benötigt.
- Am Ende der Schleife wird dann automatisch zusammengezählt.

Wir müssen noch kompilieren:

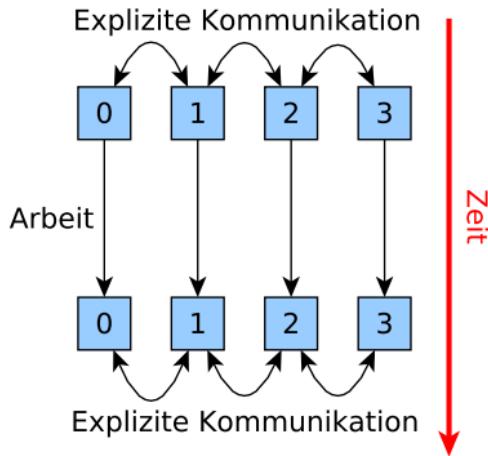
```
$ gcc -c [...] -> $ gcc -c -fopenmp [...] // compile
$ gcc -o [...] -> $ gcc -fopenmp -o [...] // link
```

⇒ `mult_Mv` und `sq_norm` in `09_03_plain_openmp/matrix.c` und `09_04_compiler_flags_openmp/matrix.c`

Dritte Optimierung: Verteilung über mehrere Rechner mit MPI

Vorlesung 9

- Supercomputer bestehen aus mehr oder weniger normalen Rechnern, die über ein sehr schnelles Netzwerk miteinander vernetzt sind.
- MPI, das *Message Passing Interface* bietet Funktionalität, um numerische Aufgaben über solche Rechner zu verteilen.



- 1 Mehrere Kopien (*Ranks*) des Programms starten gleichzeitig.
 - 2 Jedes Rank arbeitet von vorneherein an einem Teilproblem.
 - 3 Es wird explizit kommuniziert, wenn notwendig.
- **Achtung:** Aufteilung manchmal höchst nicht-trivial...

MPI Blitz-Intro 1/3

Vorlesung 9

```
#include <mpi.h>
int main(int argc, char** argv){
    int nranks, myrank;
    MPI_Init(&argc,&argv);
    // MPI_COMM_WORLD: Sammlung aller Ranks
    MPI_Comm_size(MPI_COMM_WORLD, &nranks);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("This is rank id=%d, out of %d total ranks!\n", myrank,
        nranks);
    MPI_Finalize();
    return 0;
}
```

- Kompilieren und linken mit mpicc anstelle von gcc.

```
$ gcc -c [...] -> $ mpicc -c [,,,] // compile
$ gcc -o [...] -> $ mpicc -o [...] // link
```

- Ausführen mit

```
$ mpirun -np N programm
```

wobei N die Anzahl gewünschter MPI Ranks ist.

MPI Blitz-Intro 2/3

Vorlesung 9

- Die "lokale" Problemgröße ist ein Teil der Gesamtproblemgröße

```
int main(void){  
    [...]  
    unsigned int problemgroesse = Gesamtgroesse / nranks;  
    double * vektor = malloc( sizeof(double)*problemgroesse );  
    [...]  
}
```

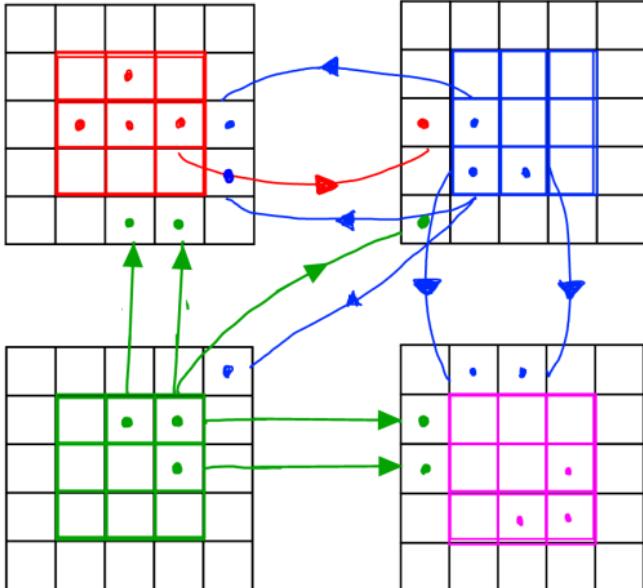
- Es wird zwischen Ranks kommuniziert, wenn notwendig

```
// vier Elemente von vektor an einen Nachbarn schicken  
MPI_Send(&(vektor[send_offset]), 4, MPI_DOUBLE,  
         nachbar_rechts, 0, MPI_COMM_WORLD);  
// und vier Elemente von einem anderen Nachbarn erhalten  
MPI_Recv(&(vektor[recv_offset]), 4, MPI_DOUBLE,  
         nachbar_links, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

MPI Blitz-Intro 3/3

Vorlesung 9

Wie würden wir das Game of Life auf vier Rechner aufteilen?



- Jeder Punkt hat acht Nachbarn.
- Hier sind die lokalen Gitter 3×3
- Wenn das Gitter in 2D auf vier Rechner verteilt ist, müssen wir vor jeder Iteration Nachbarpunkte von anderen Rechnern über das Netzwerk kopieren.
- Jeder Rechner hat Speicher für einen Rand von $4 \times 3 + 4$ fremden Nachbarn.
- Um so größer das lokale Gitter, um so günstiger ist die Kommunikation relativ zur Gesamlaufzeit.

MPI Matrix-Vektor Produkt & Vektornorm

Vorlesung 9

- Glücklicherweise müssen wir bei unserem Matrix-Vektor Produkt nicht kommunizieren. (Ausser, dass wir vorher sicherstellen müssen, dass die rechten Seiten \vec{x} auf allen Ranks gleich sind → MPI_BCast).
- Bei der Berechnung der Vektornorm jedoch schon! Hier schreiben die Ranks zwar nicht in die gleiche Variable (weil ja das ganze Programm mehrfach dupliziert aufgerufen wird), aber wir müssen mit MPI_Allreduce die Summen aufaddieren.

⇒ 09_05_plain_mpi/matrix.c,
09_05_plain_mpi/matrix_benchmark.c

⇒ 09_06_mpi_compiler_flags/Makefile

Achtung: Damit das parallele Programm die gleichen Ergebnisse liefert, wie das serielle Programm, mussten wir sicherstellen, dass die gleichen Zufallszahlen genutzt werden.

Referenzen zur Parallelisierung

Vorlesung 9

- Kursmaterialien zum MPI / OpenMP “Einführungskurs” am Forschungszentrum Jülich <https://goo.gl/Vtdhnd> (sehr umfangreich)
- “Hands-on OpenMP Introduction” YouTube + Folien:
<http://tinyurl.com/OpenMP-Tutorial>,
http://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf
(umfangreich und relativ klar)
- Liste weiter Tutorials zu OpenMP:
<http://www.openmp.org/resources/tutorials-articles/>
- Brauchbares MPI Intro <http://www2.in.tum.de/hp/file?fid=323>
- Umfangreiches MPI Tutorial <https://goo.gl/gYHcAx>

Andere Programmiersprachen

Vorlesung 9

Kurze Liste von Programmiersprachen, die in der Physik (und anderswo) Anwendung finden.

- Bash scripting: Shellskripte sind wohl das Werkzeug, das man im täglichen Gebrauch am häufigsten verwendet!
 - <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- R: mächtiges Werkzeug in der Datenanalyse, Statistik, *machine learning* → wird z.B. in der Mastervorlesung *Computational Physics* genutzt. Als Programmiersprache etwas inkonsistent, aber die Fähigkeiten machen das locker wett.
 - *R in Action* (Robert Kabacoff)
 - *The Art of R Programming* (Norman Matloff)
 - *Advanced R* (Hadley Wickham)
- Python + PANDAS: Python ist eine sehr konsistente Skriptsprache. PANDAS implementiert viele Konzepte aus R für Datenanalyse und Statistik in Python.
 - *Python for Data Analysis* (Wes McKinney)
- C++: Ursprünglich eine von Stroustrup erfundene objekt-orientierte Erweiterung von C, ist C++ jetzt eine mächtige, hoch generische und gleichzeitig – in den meisten Fällen – schnelle Programmiersprache.
 - *A Tour of C++* (Bjarne Stroustrup) - umfangreiche Einführung
 - *The C++ Programming Language* (Bjarne Stroustrup) - Referenzwerk
 - *Effective Modern C++* (Scott Meyers) - 42 sehr gut geschriebene Tipps, wie man C++ gut nutzt (nachdem man es mal gelernt hat)

That's all folks!