Programmiertechnik

Alexander Weiße

Institut für Physik, Universität Greifswald

http://theorie2.physik.uni-greifswald.de/member/weisse/protec/index.html

Montags 10:00-11:30 Uhr, Raum A 202

Einleitung

Ziele der Veranstaltung:

- Grundlegende Programmierkenntnisse
- Wichtige Algorithmen
- Tieferes Verständnis von Computern

Was heißt programmieren?

- Man bringt Computer dazu, eine bestimmte Aufgabe zu erfüllen

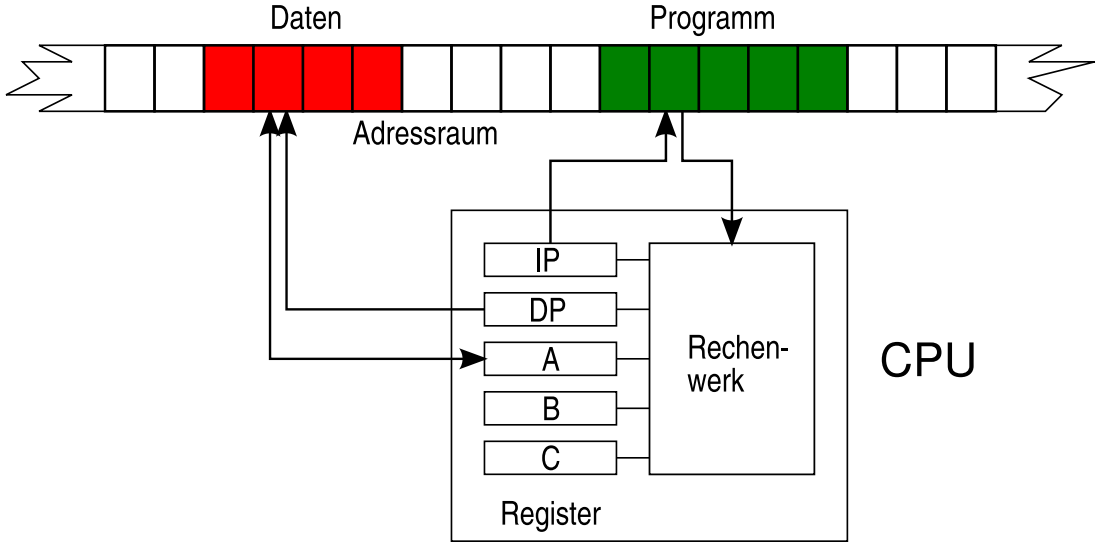
Wozu will man programmieren?

- Existierende Programme dienen bestimmten, eingeschränkten Zwecken
  $\longrightarrow$ Neue Probleme erfordern neue Programme
- Viele bestehende Programme können vom Nutzer erweitert werden oder haben eigene eingebaute Programmiersprachen, mit denen neue Funktionen geschaffen werden können

Gliederung

Schematischer Aufbau eines Computers

- Die meisten Computer folgen dem VON NEUMANN Design:



- Eine Recheneinheit (CPU) verarbeitet Daten *und* Programm aus einem gemeinsamen Speicher
- Innerhalb der CPU gibt es *Register*, die Daten und Adressen enthalten und mit einfachen Operationen verändert werden können.
- Manche Register erfüllen spezielle Funktionen, z.B.
  - Programm-Zeiger (IP): Adresse des auszuführenden Befehls
  - Daten-Zeiger (DP): Adresse zu lesender oder schreibender Daten
- Angeschlossene Geräte (Tastatur, Maus, Bildschirm, . . .) belegen bestimmte Speicherbereiche

Welche Operationen beherrscht eine CPU?

Eine CPU kann nur wenige elementare Operationen ausführen, dieses jedoch sehr schnell. Typische Klassen von Operationen sind:

- Daten vom Speicher in Register übertragen und umgekehrt
- Elementare Rechenoperationen mit Registern (Addition, Subtraktion, Multiplikation, Division, Binäroperationen)
- Hoch- und Herunterzählen spezieller Register (z.B. IP, DP)
- Vergleich von Registern
- Sprünge im Programmablauf (d.h. Veränderung von IP in Abhängigkeit vom Ergebnis vorangegangener Vergleiche)

Programmieren bedeutet, all die vielen komplexen Aufgaben, die ein Computer lösen kann, auf diese elementaren Operationen zurückzuführen. Dabei helfen *Programmiersprachen*.

Gliederung

Programmiersprachen

- Die elementaren Befehle an die CPU sind im Speicher als Zahlen gespeichert. Direkte Eingabe und eine Problembeschreibung mit Hilfe dieser Zahlen ist für Menschen extrem unpraktisch (früher: Lochkarten oder -streifen)
- Erste Erleichterung bieten Assembler-Sprachen:
  - Kurzworte für CPU-Befehle
  - Macros, Labels
  - Vorteile von Assembler: hardware-nah, schnell
- Praktisch programmiert wird meist in Hochsprachen:
  - Bsp.: C/C++, Fortran, Pascal, Basic, Java, ...
  - Algorithmen und Datenstrukturen werden in einer dem Menschen leichter zugänglichen Form dargestellt
  - Oft sind komplexe Aufgaben zu einfachen Befehlen/Funktionen zusammengefaßt (letztes Semester: octave, maxima, ...)
- Es gibt heute tausende Programmiersprachen. Neben Universalsprachen (wie C/C++) existieren auch viele Sprachen für spezielle Anwendungen (z.B. PostScript).

Beispiel: Assembler I

```
# -- Machine type IA32
# mark_description "Intel(R) C++ Compiler for applications r
Build 20070913 %s";
# mark_description "-S -o hello.s";
        .file "hello.c"
        .text
..TXTST0:
# -- Begin  main
# mark_begin;
       .align    2,0x90
        .globl main
main:
..B1.1:                                 # Preds ..B1.0
        pushl     $3
#3.16
        call      __intel_new_proc_init
#3.16
                                        # LOE ebx ebp esi edi
```

Beispiel: Assembler II

```
..B1.5:                                 # Preds ..B1.1
        pushl     $_2__STRING.0.0
#5.10
        call      printf
#5.3
                                        # LOE ebx ebp esi edi
..B1.6:                                 # Preds ..B1.5
        addl      $8, %esp
#5.3
                                        # LOE ebx ebp esi edi
..B1.2:                                 # Preds ..B1.6
        xorl      %eax, %eax
#7.10
        ret
#7.10
        .align    2,0x90
                                        # LOE
# mark_end;
```

Beispiel: Assembler III

```
        .type   main,@function
        .size   main,.-main
        .data
# -- End  main
        .section .rodata.str1.4, "aMS",@progbits,1
        .align 4
        .align 4
__2__STRING.0.0:
        .byte   72
        .byte   101
        .byte   108
        .byte   108
        .byte   111
        .byte   44
        .byte   32
        .byte   119
        .byte   111
        .byte   114
```

Beispiel: Assembler IV

```
        .byte   108
        .byte   100
        .byte   33
        .byte   10
        .byte   0
        .type   _2__STRING.0.0,@object
        .size   _2__STRING.0.0,15
        .data
        .section .note.GNU-stack, ""
# End
```

Beispiel: C

```c
#include <stdio.h>

int main(void) {

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

    return 0;

}
```

Beispiel: Fortran

```
program HelloWorld
   write (*,*) 'Hello, world!'
end program HelloWorld
```

Beispiel: PostScript

```
50 500 moveto
/Helvetica findfont 80 scalefont setfont
(Hello, world!) show
```

Gliederung

Warum C?

In der Vorlesung soll vorrangig auf C eingegangen werden. Warum?

- C ist Hochsprache und trotzdem maschinennah
- C ist universell einsetzbar und kann in schnellen Maschinencode übersetzt werden
- Die meisten aktuellen Betriebssysteme sind in C programmiert (Unix, Windows, Mac OS X)
- Viele andere Programmiersprachen orientieren sich an C oder sind damit verwandt, z.B. C++, Java, AWK, Perl, ... (C $\hat{=}$ Latein)
- C ist standardisiert und sehr portabel. Es gibt viele Compiler für C.
- In C kann strukturiert und klar programmiert werden (leider gilt auch das Gegenteil)

Geschichte von C

- Als erste Programmiersprachen entstanden in den 1950ern Lisp und Fortran (**For**mula **Tran**slator)
- Aus Fortran entwickelte sich Algol (**Algo**rithmic **L**anguage)
- Aus Algol entstanden CPL (**C**ommon **P**rogramming **L**anguage), BCPL (Basic CPL), schließlich B
- Um 1971 verbesserten Ken Thompson und Dennis Ritchie von den Bell Labs die Sprache B, unter anderem um portable Versionen von Unix zu programmieren. Es entstand C.
- Eine erste Beschreibung und lange Zeit der Standard für C war:
  Brian W. Kernighan, Dennis M. Ritchie
  The C Programming Language
  Prentice Hall, 1978
- Der erste offizielle Standard für C erschien 1989 (ANSI C)
- Ein erweiterter Standard folgte 1999 (C99)

Praktische Schritte beim Programmieren

1. Der Programmtext muß in den Rechner:
   Man bemüht einen Editor (z.B. Emacs, gedit, ...)
2. Das Programm muß in maschinenlesbare Form gebracht werden. Dazu gibt es zwei Möglichkeiten:
   - 2.1 Kompilieren: Das Programm wird als ganzes in Maschinensprache übersetzt und danach gestartet.
     Typisch für Sprachen wie C, Fortran, etc.
   - 2.2 Interpretieren: Das Programm wird Schritt für Schritt übersetzt und ausgeführt.
     Typisch für Skriptsprachen wie Perl, AWK, Python

Es gibt auch Programme, die diese Schritte zusammenfassen, sogenannte integrierte Entwicklungsumgebungen (IDE). Beispiele:

Linux	Windows
Geany	Dev C++
Anjuta	Local C Compiler
Eclipse	
KDevelop	

Allgemeine Eigenschaften von C

- C Programme können in freier Form eingegeben werden. Leerzeichen und Zeilenumbrüche können an beliebiger Stelle stehen.
- Einzelne Befehle werden mit Semikolon abgeschlossen
- Programme werden in Funktionen gegliedert, die Programmausführung beginnt mit dem Aufruf der Funktion `main()`
- Variablen haben stets einen Datentyp
- C hat relativ wenig reservierte Wörter. Für die meisten komplizierteren Operationen (Ein- und Ausgabe, Mathematik etc.) gibt es Funktionen in der sog. Standard-Bibliothek oder vom Betriebssystem

Das zweite C Programm: `kreis.c`

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

int main(void) {

   double radius, flaeche;

   /* Lies Radius */
   printf(" Radius = ");
   scanf("%lg", &radius);

   flaeche = M_PI*radius*radius;

   /* Gib Kreisflaeche aus */
   printf(" Flaeche = %lg\n", flaeche);

   return 0;

}
```

Anmerkungen zu `kreis.c`

Dieses kurze Programm verdeutlicht bereits einige sehr allgemeine Eigenschaften von Programmen:

- Es folgt der Struktur

$$\boxed{\text{Eingabe}} \rightarrow \boxed{\text{Berechnung}} \rightarrow \boxed{\text{Ausgabe}}$$

- Aus den Eingabe-Daten werden mittels eines Algorithmus die Ausgabe-Daten berechnet ($A = \pi r^2$)
- Es gibt typisierte Variablen (`radius`, `flaeche`) und Konstanten (`M_PI`)

Außerdem wurden stilistische Grundregeln befolgt:

- Variablen haben verständliche Namen
- Wichtige Programmschritte sind mit Kommentaren versehen
- Durch Leerzeichen und Umbrüche ist das Programm strukturiert

Gliederung

Programmstruktur

- Größere C-Programme bestehen meist aus mehreren Dateien, die vom Compiler *einzeln* übersetzt werden
- Innerhalb einer Datei (.c) können ein oder mehrere Funktionen und globale Variable definiert werden.
- Damit Funktionen aus einer anderen Datei benutzt werden können, müssen diese in der aktuellen Datei deklariert werden
- Mechanismus: Header-Datei (.h), die nur Funktionsnamen und Datentypen von Argumenten und Rückgabewerten enthält
- Der sog. PRÄPROZESSOR-Befehl `#include` dient zum Einbinden der Header-Dateien
- Spitze Klammern <> laden System-Dateien, mit Anführungszeichen "" werden lokale Header geladen

Programmstruktur – Beispiel

futest.c

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

int main(void) {

   double x, y;

   printf(" x = ");
   scanf("%lg", &x);

   y = myfunct(x);

   printf(" y = %lg\n", y);

   return 0;

}
```

myfunct.h

```c
double myfunct(double);
```

myfunct.c

```c
double myfunct(double x) {

   return x*x;

}
```

Variablen & Konstanten I

- Variablen sind benannte Speicherbereiche für Daten, die vom Programm verändert werden sollen. Je nach Inhalt haben sie einen bestimmten Typ. Es gibt folgende Grundtypen:

`char`	einzelner Buchstabe
`int`	ganze Zahl
`float`	einfach genaue Fließkommazahl
`double`	doppelt genaue Fließkommazahl

- Diese einfachen Typen können durch bestimmte Schlüsselworte geändert/erweitert werden (Übung: Ausgabe der Typlänge mittels `sizeof()`)

`short int`	ganze Zahl mit weniger Bits
`long int`	ganze Zahl mit mehr Bits
`long long int`	ganze Zahl mit viel mehr Bits
`unsigned int`	vorzeichenlose ganze Zahl

- Später wird gezeigt, wie komplizierte Datentypen aus einfachen zusammengesetzt werden (z.B. Typ `student` ;-)

Variablen & Konstanten II

- Eine Variablen-Deklaration hat folgende Form:

```
typ name [= Wert];
```

  Beispiele:

```
double x = 5.0;
int a, b, n = 10;
```

- Variablennamen bestehen aus Buchstaben oder Zahlen, und müssen mit einem Buchstaben beginnen. Auch Unterstrich _ ist erlaubter Buchstabe (oft bei internen Variablen). Groß- und Kleinschreibung wird unterschieden.
- Deklarationen gelten nur innerhalb einer Funktion. Variablen, die außerhalb einer Funktion deklariert werden, gelten innerhalb der Datei.
- Wert-Zuweisungen an Variablen anderen Typs werden vom Compiler automatisch umgewandelt oder vom Programmierer erzwungen:

```
int n, m;
double x = 3.59;

n = x;
m = (int) x;
```

Variablen & Konstanten III

- Konstanten können im Programm direkt eingegeben werden:

`5.0`	double 5
`1.0e-2`	double 0.01
`10`	int 10
`0x1a`	int, hexadezimal (=26)
`013`	int, octal (=11)
`4.0f`	float 4
`"Ein Text"`	Zeichenkette

- Praktischer sind meist benannte Konstanten, die über den Präprozessor-Befehl `#define` festgelegt werden:

```
#define PI 3.1415926
#define N 1000
```

- Viele Programmierer verwenden Großbuchstaben für Konstanten und Kleinbuchstaben für Variablen
- Namen von Variablen und Konstanten sollten nicht mit reservierten Worten zusammenfallen

Reservierte Worte

ANSI C89 / ISO C90

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

ISO C99

`_Bool`	`_Imaginary`	`restrict`
`_Complex`	`inline`	

Einfache Operationen

Mit Variablen und Konstanten können wir rechnen ...

Rechnen: `+ - * / ++ --`

Rest: `%`

Binär: `& | ^ << >> ~`

	and (&)	or (|)	xor (^)
0 op 0	0	0	0
0 op 1	0	1	1
1 op 0	0	1	1
1 op 1	1	1	0

Logik: `&& || !`

Vergleiche: `== != < > <= >=`

Zuweisung: `= += *= -= /= ...`

Beispiel – Darstellung bel. Basis

```c
#include <stdio.h>

#define basis 5

int main(void) {

   int x, i;

   printf(" x = ");
   scanf("%i", &x);

   i = 0;
   while(x>0) {
      printf("%i * %i^%i\n", x % basis, basis, i);
      x = x / basis;
      i++;
   }

   return 0;

}
```

Beispiel – Binärdarstellung

```c
#include <stdio.h>

int main(void) {

   int i, n, x;

   printf(" x = ");
   scanf("%i", &x);

   n = 8*sizeof(int);

   for(i=n-1; i>=0; i--)
      printf("%i", (x>>i) & 1);
   printf("\n");

   return 0;

}
```

Operatoren

`() [] -> .`	links nach rechts
`! ~ ++ -- + - * (type) sizeof`	rechts nach links
`* / %`	links nach rechts
`+ -`	links nach rechts
`<< >>`	links nach rechts
`< <= > >=`	links nach rechts
`== !=`	links nach rechts
`&`	links nach rechts
`^`	links nach rechts
`|`	links nach rechts
`&&`	links nach rechts
`||`	links nach rechts
`?:`	rechts nach links
`= += -= *= /= %= &= ^= |= <<= >>=`	rechts nach links
`,`	links nach rechts

Operatoren haben eine Rangfolge und eine Richtung der Auswertung

Gliederung

Kontrollstrukturen

- Es gibt nahezu kein Programm, daß mit einer einfachen linearen Befehlsfolge auskommt.
- Üblicherweise gibt es Programmteile, die nur unter bestimmten Voraussetzungen ausgeführt werden sollen.
- Ein anderes wichtiges Strukturelement sind Schleife, d.h. Befehle die mehrfach hintereinander ausgeführt werden.
- C bietet dafür verschiedene Programmstrukturen:

`if() { } else { }`	Wenn ... dann ... sonst
`switch() { case ...: }`	Fallunterscheidung
`do { } while();`	Tue ... solange
`while() { }`	Solange ... tue ...
`for(..;..;..) { }`	Schleife von ... bis

Bedingte Ausführung

- Grundstruktur:

```
if(bedingung) {
   ...;
} else {
   ...;
}
```

- Beispiel:

```
if((a<5) && (b==3)) {
   c = 5*a;
 } else {
   c = 7*b;
}
```

- Kurzform für bedingte Zuweisung:

```
c = ((a<5) && (b==3)) ? 5*a : 7*b;
```

Fallunterscheidung

Gelegentlich wird eine Variable auf mehrere Werte getestet. Statt

```
if(a==1) {
  fall1();
} else if (a==5) {
  fall5();
} else {
  sonstiges();
}
```

gibt es die übersichtlichere Form:

```
switch(a) {
 case 1:
   fall1();
   break;
 case 5:
   fall5();
   break;
 default:
   sonstiges();
   break;
}
```

Schleifen I

- Im Prinzip können alle Schleifen mit Hilfe von `for()` programmiert werden.
- `do ... while` Konstruktionen sind aber oft übersichtlicher

```
do {
   ...;
 } while(bedingung);

while(bedingung) {
   ...;
 }
```

- Für Bedingungen gelten die selben Regeln wie bei `if() ... else ...`

Schleifen II

- Die am häufigsten verwendete Schleifenstruktur ist

```
for(Initialisierung; Bedingung; Inkrement) {
  ...;
}
```

- Kanonisches Beispiel:

```
for(i=0; i<n; i++) {
  funktion(i);
}
```

- Dies ist äquivalent zu:

```
i=0;
while(i<n) {
  funktion(i);
  i++;
}
```

- Initialisierung, Bedingung und Inkrement können beliebige andere Befehle enthalten. Die Schleife entspricht dann einer entsprechend komplizierteren `while`-Schleife

Übung 1

Setzen Sie sich mit den in der Vorlesung behandelten Beispiel-Programmen auseinander.

Schreiben Sie ein einfaches Programm das prüft, ob eine gegebene Zahl prim ist.

Gliederung

Standard-Bibliothek

- Viele häufig benutzte Funktionen werden von der sog. Standard-Bibliothek zur Verfügung gestellt, die jedem C-Compiler beiliegen muss. Unzählige weitere Funktionen liefert das jeweilige Betriebssystem.
- Übersicht:

assert.h	Hilfsfunktionen zur Fehlersuche
complex.h	Definition komplexer Zahlen und Funktionen
ctype.h	Klassifikation von Buchstaben (isalpha, islower, isdigit)
errno.h	Fehlercodes von Bibliotheksfunkionen
fenv.h	Fehler-Behandlung bei Fließkommaoperationen, Genauigkeit
float.h	Charakteristik von Fließkommazahlen
inttypes.h	Charakteristik von ganzen Zahlen
iso646.h	Alternative Schreibweise für bestimmte Schlüsselworte (`and` statt `&&`)
limits.h	Schranken für ganze Zahlen

Standard-Bibliothek

locale.h	Landesspezifische Einstellungen für Zahlen/Währungen
math.h	Mathematische Funktionen
setjmp.h	Nichtlokale Sprünge über Funktionsgrenzen hinweg
signal.h	Reaktion auf externe Signale (kill, halt, etc.)
stdarg.h	Mechanismen für Funktionen mit beliebig vielen Argumenten
stdbool.h	Boolsche Variable und Werte
stddef.h	Definition best. Zeigertypen
stdint.h	Ganzzahl-Typen bestimmter Länge
stdio.h	Die wichtigsten Ein-/Ausgabe-Funktionen
stdlib.h	Verschiedene oft benutzte Funktionen (Typumwandlung, Dyn. Speicher, Sortieren, ...)
string.h	Funktionen für Zeichenketten
time.h	Zeitfunktionen
wchar.h	Funktionen für Sonderzeichen, nicht-europäische Zeichensätze
wctype.h	Weitere Funktionen dazu, Umwandlungen

stdio.h

STDIO.H stellt Funktionen zur Ein- und Ausgabe von Daten bereit:

- Öffnen und Schließen eine Datei:

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *fp);
```

  Hier bezeichnet `FILE` den Datentyp "Datei", `path` den Dateinamen und `mode` den Zugriffsmodus auf die Datei:

mode	Bedeutung
"r"	Lesen
"w"	Schreiben
"a"	Anhängen

- Formatierte Ausgabe:

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

  `stream` ist die zuvor geöffnete Datei, `format` bestimmt das Ausgabe-Format, danach folgen die Daten.

- Spezialfall: Standard-Ausgabe (`STDIO`)

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

stdio.h II

- Die Formatangabe ist eine Zeichenkette aus normalem Text und Beschreibungen der auszugebenden Daten, die jeweils mit "%" beginnen.
  Struktur: `%[flag][breite.genauigkeit][typ]`

flag	Bedeutung
#	Nutze alternative Darstellung, z.B. 0x vor hex-Zahlen
0	gibt führende Nullen aus
-	linksbündige Ausgabe || ' '	Leerzeichen vor positiver Zahl
+	immer Vorzeichen (+/-) vor Zahl

Breite/Gen.	Bedeutung
Zahlen	Feldbreite in Bytes
*	Feldbreite folgt als nächstes Argument nach den Daten
*m$	Feldbreite folgt als m-tes Argument

stdio.h III

- Datentyp:

typ	Bedeutung
i, d	`int` dezimal
o	`int` octal
x,X	`int` hexadezimal
f,e,g	`double` ohne, mit und mit automatischem Exponent (1.0e2)
c	`char` einzelner Buchstabe
s	`char *` Zeichenkette
p	`viod *` Zeiger/Adresse

- Für modifizierte Grundtypen gibt's modifizierte Formatbeschreibungen:

typ	Bedeutung
hhi	`signed char`
hi	`short int`
i	`int`
li	`long int`
lli	`long long int`

stdio.h IV

- Formatierte Eingabe:

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

  Wie bei `printf` steht `stream` für die geöffnete Datei und `format` beschreibt die einzulesenden Daten. Die nachfolgenden Argumente sind meist Zeiger auf Variablen (Adressen).
- Spezialfall: Standard-Eingabe (`STDIO`)

  ```
  int scanf(const char *format, ...);
  ```
- Achtung: Die Formatangaben bei den `scanf`-Funktionen unterscheiden sich teilweise von denjenigen für `printf`. Beispiel:

  ```
  double x;

  scanf(" %lg", &x);
  printf(" x = %g\n");
  ```

  Näheres unter "`man scanf`"

stdio.h V

- Weitere Funktionen:

`feof()`	Melde Dateiende
`ferror()`	Melde Fehler beim Zugriff
`fflush()`	Leeren von Zwischenspeichern
`remove()`	Datei löschen
`rename()`	Datei umbenennen
`fgetc()`	Lies einzelnes Zeichen
`fputc()`	Schreibe einzelnes Zeichen
`fgets()`	Lies Zeichenkette
`fputs()`	Schreibe Zeichenkette
`fread()`	Lies Datei direkt den Speicher
`fwrite()`	Schreibe Speicherinhalt in Datei
`fseek()`	setze Lese-Position in der Datei
`ftell()`	ermittle aktuelle Position
...	weitere siehe Standard oder WIKI

stdio.h – Beispiel

```c
#include <stdio.h>

#define N 50

int main(void) {

  int i;
  double x, s;

  FILE *iofile;

  iofile = fopen("bspout.dat", "w");

  if(iofile==NULL) {
    printf("Fehler beim Oeffnen!");
  } else {

    s = 0.0;
    x = 1.0;

    for(i=1; i<N; i++) {
      s += x;
      x /= (double) i;
      fprintf(iofile, " %03x %+25.20g\n", i, s);
    }

    fclose(iofile);
  }

  return 0;
}
```

stdio.h – Beispiel

```
#include <stdio.h>

int main(void) {

   int i, n;
   double x;

   FILE *iofile;

   iofile = fopen("bspout.dat", "r");

   if(iofile==0) {
      printf("Fehler beim Oeffnen!\n");
   } else {

      while(!feof(iofile)) {
         n = fscanf(iofile ," %x %lg", &i, &x);
         if(n==2) printf("%i %.20g\n", i, x);
      }
      fclose(iofile);
   }

   return 0;
}
```

stdlib.h I

STDLIB.H stellt verschiedene Hilfsfunktionen zur Zahlenumwandlung, Speicherbelegung, etc. zur Verfügung

- Umwandlung von Zeichenketten in Zahlen:

`atoi()`	Zeichenkette in `int`
`atol()`	Zeichenkette in `long`
`atof()`	Zeichenkette in `double`
`strtol()`	Zeichenkette bel. Basis in `long`
`strtod()`	Zeichenkette bel. Basis in `double`

- Reservierung von Speicher:

`malloc()`	Reservierung einer bestimmten Anzahl Bytes
`calloc()`	Reservierung für mehrere Elemente eines Typs
`free()`	Aufhebung einer Reservierung

- System-Befehle:

`exit()`	normale Beendigung des Programms
`abort()`	Abbruch des Programms
`system()`	Übergabe eines Befehls ans System
`getenv()`	Auslesen und Umgebungsvariablen

stdlib.h II

- Zufallszahlen:

`rand()`	ganze Zufallszahl zwischen 0 und RAND_MAX
`srand()`	Initialisierung des Pseudo-Zufallszahlengenerators
`drand48()`	reelle Zufallszahl $\in [0, 1)$
`srand48()`	Initialisierung des Pseudo-Zufallszahlengenerators
...	weitere Funktionen für andere Typen

- Suchen und Sortieren:

`bsearch()`	Suche in einer geordneten Liste
`qsort()`	Sortieren einer Liste

math.h I

MATH.H stellt die gebräuchlichsten Mathematikfunktionen zur Verfügung

- Exponentialfkt. & Logarithmen

`exp(x)`	e^x
`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$
`pow(x,y)`	x^y
`sqrt(x)`	$\sqrt{x}$

- Runden, etc.

`floor(x)`	größte ganze Zahl kleiner als x
`ceil(x)`	kleinste ganze Zahl größer als x
`round(x)`	gerundet auf ganze Zahl
`fabs(x)`	$\lvert x \rvert$

- Konstanten

`M_PI`	π
`M_E`	e
`M_LN2`	$\ln 2$
`M_SQRT2`	$\sqrt{2}$
...	sowie Vielfache und Inverse davon

`math.h` II

- Winkelfunktionen

`sin(x)`	$\sin x$
`cos(x)`	$\cos x$
`tan(x)`	$\tan x$
`asin(x)`	$\sin^{-1}(x) \in [-\pi/2, \pi/2]$
`acos(x)`	$\cos^{-1}(x) \in [0, \pi]$
`atan(x)`	$\tan^{-1}(x) \in [-\pi/2, \pi/2]$
`atan2(x,y)`	$\tan^{-1}(y/x) \in [-\pi, \pi]$
	(Kartesische K.$\rightarrow$ Polarkoord.)

- Hyperbolische Funktionen

`sinh(x)`	$\sinh x$
`cosh(x)`	$\cosh x$
`tanh(x)`	$\tanh x$
`asinh(x)`	$\sinh^{-1} x$
`acosh(x)`	$\cosh^{-1} x$
`atanh(x)`	$\tanh^{-1} x$

complex.h

- COMPLEX.H definiert komplexe Varianten der Funktionen aus `math.h`, z.B. `csin(z)`, `cexp(z)`, ...
- Weitere Funktionen zum Umgang mit kompl. Zahlen:

`creal(z)`	Realteil von z
`cimag(z)`	Imaginärteil von z
`cabs(z)`	Betrag von z
`carg(z)`	Argument ϕ von $z = r\,\mathrm{e}^{\mathrm{i}\,\phi}$

- Deklaration:

```
double _Complex z = 1.5 + 12.8i;
double complex z = 1.5 + 12.8*I;
```

string.h

- STRING.H enthält verschiedene Funktionen zum Umgang mit Zeichenketten (näheres dazu später)

`strcpy()`	Kopiere Zeichenkette
`strcat()`	Anhängen einer Zeichenkette
`strcmp()`	Vergleich
`strstr()`	Suche nach Zeichenkette
`strlen()`	Länge der Zeichenkette
`...`	weitere Suchfunktionen

- Außerdem werden Funktionen zum Kopieren und Verschieben von Speicherinhalten zur Verfügung gestellt.

`memcpy()`	Kopieren von Speicherinhalten
`memmove()`	Verschieben von Speicherinhalten
`memcmp()`	Vergleichen von Speicherinhalten
`memset()`	Speicherbereich auf bestimmten Wert setzen
`...`	

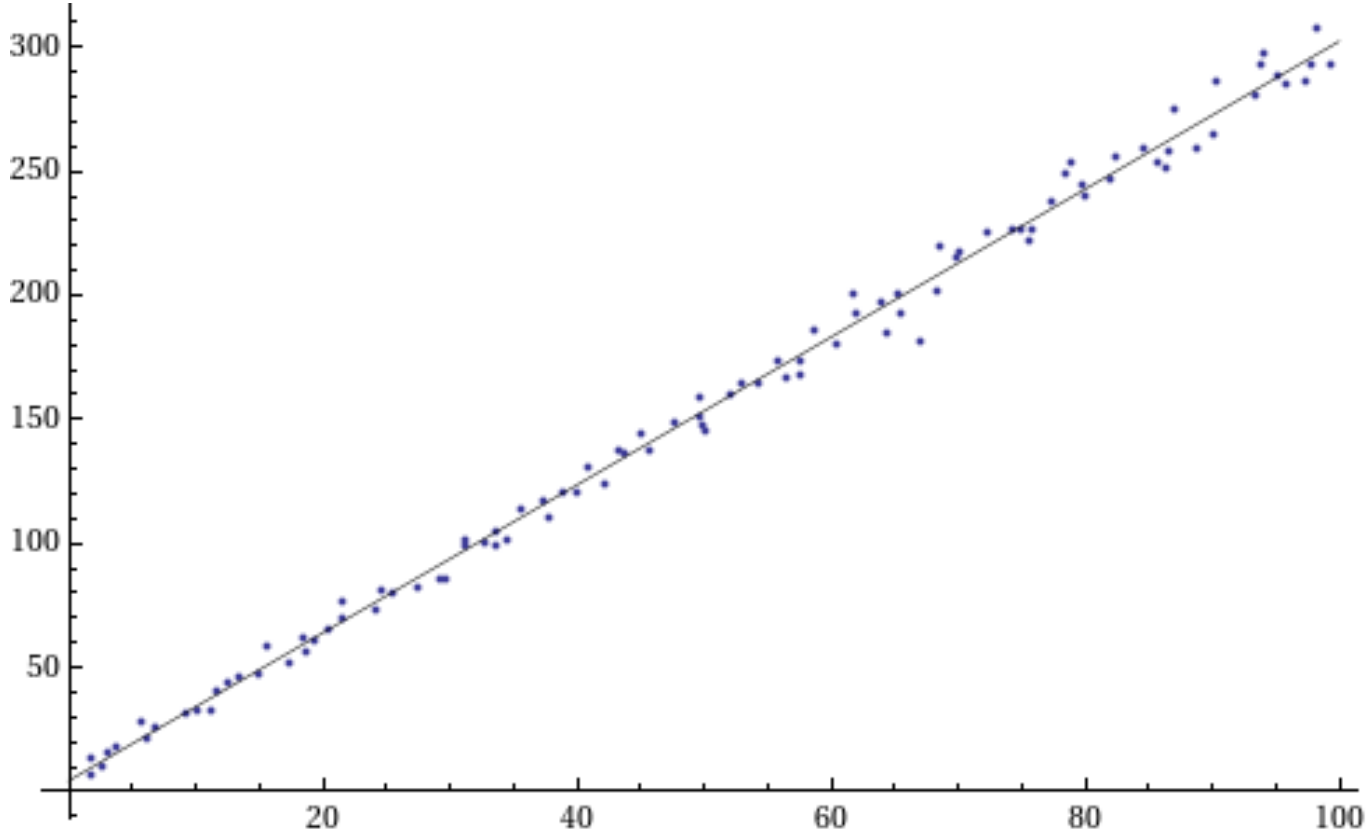
time.h

TIME.H stellt Funktionen für Datum und Zeit zur Verfügung

`clock()`	CPU Zeit seit Programmstart
`time()`	Aktuelle Kalenderzeit (internes Format)
`localtime()`	Umrechnung in strukturiertes Format und lokale Zeit
`difftime()`	Zeitdifferenz in Sekunden
`asctime()`	Zeitangabe als Zeichenkette
`strftime()`	Umwandlung Zeit $\rightarrow$ freies Textformat
...	

Übung 2

Schreiben Sie ein Programm, das aus einer Datei einen Datensatz aus mehreren (x, y) Werten einliest und eine Gerade daran anpaßt. Machen Sie sich zuerst klar, was zu berechnen ist, bevor Sie anfangen zu programmieren.



Gliederung

Funktionen

- Im Abschnitt über die Standard-Bibliothek haben wir bereits eine Vielzahl von Funktionen kennengelernt.
- Die Zerlegung in einzelne Funktionen ist ein wichtiges Element zur Strukturierung eines Programms.
- Funktionen fassen wiederholt auftretende Programmschritte zusammen.
- Funktionen kapseln verschiedene Teile eines Programms gegeneinander ab. Der Nutzer einer Funktion muß nichts über die Interna der Funktion wissen (black box).
- *Geschickt konstruierte* Funktionen führen zu wiederverwendbarem Code.
- Bei C dienen Funktionen ohne Rückgabewert auch als Unterprogramme. Andere Programmiersprachen unterscheiden gelegentlich Prozeduren und Funktionen (Pascal, Fortran).

Funktionen – Deklaration

- Die Grundstruktur einer Funktion besteht aus:

```
typ name(typ var1, typ var2, ...) {
 typ wert;
 ...
 wert = ...
 return wert;
}
```

- Die Funktionsargumente werden als Werte übergeben, d.h. die Variablen `var1, var2, ...` enthalten Kopien der übergebenen Werte. `var1, var2, ...` können innerhalb der Funktion verändert werden, ohne daß dies Auswirkungen auf das aufrufende Programm hat.
- Innerhalb der Funktion deklarierte Variablen (einschließlich der Argumente) gelten nur lokal innerhalb der Funktion. Ihre Werte werden bei *jedem* Aufruf neu initialisiert (Ausnahme: Deklaration mit `static`)
- Sollen Daten außerhalb der Funktion verändert werden, müssen Zeiger, d.h. Adressen von Daten, übergeben werden (siehe später).

Funktionen – Beispiele I

```c
#include <stdio.h>

/* pruefe, ob x prim ist */
int primQ(int x) {

   int d, p;

   d = 2;
   p = 1;
   while((d<x) && (p==1)) {
      if((x % d)==0) p=0;
      d++;
   }

   return p;
}
```

Funktionen – Beispiele II

```c
/* Hauptprogramm */
int main(void) {

   int x;

   printf(" x = ");
   scanf("%i", &x);

   if(primQ(x)) {
      printf(" %i ist prim\n", x);
   } else {
      printf(" %i ist nicht prim\n", x);
   }

   return 0;

}
```

Gliederung

Datenstrukturen

- Die geschickte Anordnung und die Strukturierung von Daten sind entscheidende Faktoren für erfolgreiche Programmierung
- Strukturierte Daten führen zu übersichtlichen Algorithmen
- Die Anordnung von Daten hat großen Einfluß auf die Geschwindigkeit eines Programms
- C bietet verschiedene Möglichkeiten zur Datenstrukturierung:
  - Arrays = Listen gleichartiger Daten
  - Strukturen = Kombination verschiedener Daten zu neuem Objekt
  - Unionen = Container für Daten variablen Typs
- Diese Strukturtypen können kombiniert und verschachtelt werden
- Mittels `typedef` können die neuen Datenobjekte eigene Namen erhalten.
- Funktionen können die neuen Datentypen zurückliefern oder als Argument erwarten

Arrays & Zeiger

- Arrays sind eine Aneinanderreihung mehrerer Objekte des gleichen Typs
- Ein statisches Array (fester Länge n) wird wie folgt deklariert:

```
typ name[n];
typ name2[] = {1,2,3};
```

- Auf die Elemente des Arrays wird mittels eines ganzzahligen Index zugegriffen, der die Werte 0 bis $n-1$ annehmen darf:

```
name[0] = 5;
name[1] = 3;
...
```

- Achtung: C-Compiler überprüfen meist nicht, ob die Feldgrenzen eingehalten werden. Feldüberschreitungen sind eine häufige Ursache für Programmfehler, Anfälligkeit für Viren, etc.
- Die Variable, die ein Array beschreibt, ist ein sogenannter Zeiger. Diese Art von Variablen enthält die Speicheradresse von Daten.
- Im Falle eines Array enthält `name` die Adresse des ersten Elements, `name[0]`

Arbeiten mit Zeigern

- Zeiger werden oft auch unabhängig von Arrays verwendet.
- Deklaration:

```
typ *name;
```

- Um Daten an die Adresse, auf die ein Zeiger verweist, zu schreiben oder von dort zu lesen, verwendet man `*`:

```
*name = wert;
```

- Um die Adresse zu bestimmen, an der der Inhalt einer Variablen abgelegt ist, verwendet man `&`:

```
typ a, *p;
p = &a;
```

- Zeiger kennen den Typ der Daten, auf die sie zeigen (außer `void *`)
- Mit Zeigern kann gerechnet werden:

```
typ a[5]
*(a+3) = 1;
```

- Es gibt auch Zeiger auf Funktionen (d.h. auf den Anfang des entsprechenden Programmcodes).

Zeiger – Beispiel

```c
#include <stdio.h>

int main(void) {

   double a[5], *p;
   int i;

   for(i=0; i<5; i++) a[i] = i;
   for(p=a; p<(a+5); p++) printf(" %g\n", *p);

   return 0;
}
```

Zeiger auf Funktionen – Beispiel

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

int main(void) {

   int i;
   double (*fct[3])(double);

   fct[0] = sin;
   fct[1] = cos;
   fct[2] = tan;

   for(i=0; i<3; i++)
      printf(" %g\n", (*fct[i])(0.5));

   return 0;
}
```

Zeichenketten

- Zeichenketten sind Arrays von 1-Byte-Ganzzahlen (`char`). Das letzte Element enthält stets den Wert 0.
- Deklaration:

```
char text[100];
char text2[] = "Ein Text.";
```

- Konstante Zeichenketten werden in `""` eingeschlossen, der Compiler berechnet die richtige Länge und hängt 0 an.
- Zuweisung einzelner Zeichen erfolgt über:

```
text[0] = 'A';
text[1] = '\0'
```

- Die Standard-Bibliothek enthält mit `string.h` eine Reihe von Funktionen für Zeichenketten, z.B. Kopieren, Suchen, Vergleichen ...

Zeichenketten – Beispiel

```c
#include <stdio.h>
#include <string.h>

#define MAXLEN 200

int main(void) {

   char text[] = "Hallo,", text2[MAXLEN], text3[MAXLEN];

   strncpy(text2, "Welt.", MAXLEN);

   sprintf(text3, "Ich sage: %s %s\n", text, text2);

   printf("Mein Text: %s", text3);
   printf("Laenge = %i\n", strlen(text3));

   return 0;
}
```

Strukturen I

- Strukturen fassen mehrere Objekte verschiedenen Typs zu einem neuen Objekt zusammen
- Definition:

```
struct name {
   typ elem1;
   typ elem2;
}
```

- Die Elemente einer Struktur können beliebigen Datentyp haben, insbesondere kann es sich wieder um Strukturen handeln
- Deklaration eines Objekts vom neuen Typ:

```
struct name meine_variable;
```

- Zugriff auf die Elemente des neuen Objects:

```
meine_variable.elem1 = 10;
```

- Zeiger auf Strukturen werden wie andere Zeiger deklariert:

```
struct name *mein_zeiger;
```

Strukturen II

- Für den Zugriff auf die Elemente einer Struktur, auf die der Zeiger zeigt, gibt es eine Kurznotation:

```
(*mein_zeiger).elem1 = 10;
mein_zeiger->elem1 = 10;
```

- Die Deklaration eines Struktur-Objekte über `struct name variable;` wirkt schwerfällig. Eleganter ist die Verwendung von `typedef`:

```
typedef struct [name] { ... } neuer_typ;
```

- Beispiel:

```
#include <stdio.h>

typedef struct {
  double x, y, z;
} punkt;

double dot(punkt a, punkt b) {
  return a.x*b.x + a.y*b.y + a.z*b.z;
}
```

Strukturen III

```c
/* Hauptteil, Arbeit mit punkt */
int main(void) {

   punkt a = {1,2,3}, b;

   b.x = 0.0;
   b.y = 1.0;
   b.z = 0.0;

   printf(" a.b = %g\n", dot(a,b));

   return 0;
}
```

Unionen

- Unionen beschreiben Objekte, die verschiedenen Typ haben können.
- Definition:

```
union name {
   int ival;
   double dval;
}
```

- Deklaration entsprechender Variablen:

```
union name meine_union;
```

- Wie bei Strukturen kann auch der elegantere Weg über `typedef` gegangen werden.
- Der Zugriff auf den Inhalt einer Union erfolgt auch ähnlich wie bei `struct`:

```
meine_union.ival = 10;
meine_union.dval = 5.0;
```

- Achtung: Der Programmierer muß selbst darauf achten (und Buch darüber führen), welchen Typ die Union gerade enthält.
- Unionen sind ein eher selten verwendetes Programmelement.

Übung 3

Schreiben Sie eine Funktion, die die Zerlegung einer ganzen Zahl in Primfaktoren berechnet. Rufen Sie die Funktion aus einem Hauptprogramm auf.

Gliederung

Weitere Sprachelemente I

Die bisher behandelten Sprachelemente von C genügen für die meisten Programmieraufgaben. Der Vollständigkeit halber seien hier noch weitere Elemente erläutert:

Modifikatoren:

- `auto type var;` definiert eine automatische Variable, die am Anfang eines Blocks bereitgestellt und am Ende gelöscht wird. Alle normalen Variablen sind automatisch, d.h. `auto` ist redundant.
- `extern type var;` oder `extern type func(...);` deklarieren Funktionen, die in einer anderen Datei (extern) definiert werden.
- `const type var = val;` definiert eine Variable, deren Wert nicht verändert wird.
- `register type var;` bittet den Compiler, ein CPU-Register für `var` zu benutzen. Meist unnötig, da der Compiler diese Entscheidungen beim Optimieren besser trifft.

Weitere Sprachelemente II

- `volatile type var;` definiert eine Variable, deren Wert von außen (ohne Zutun des Programms) verändert werden kann (z.B. Hardware-Register). Der Compiler wird an solchen Variablen keine leichtfertigen Optimierungen vornehmen.
- `static type var;` definiert lokale Variablen, deren Wert über mehrere Funktionsaufrufe erhalten bleibt (Gegenteil von `auto`). Bei globalen Variablen führt `static` dazu, daß sie von außen (andere Dateien) nicht sichtbar sind.
- `type * restrict var;` teilt dem Compiler mit, daß `var` der einzige Zeiger ist, der auf den betreffenden Speicherbereich zeigt. Dies erleichtert bestimmte Optimierungen.
- `inline type func(...);` bittet den Compiler, zeitraubende Sprünge zu vermeiden und den Code der Funktion direkt an die Stelle des Aufrufs zu kopieren. Kann gelegentlich Zeit sparen, wird aber beim Optimieren oft schon automatisch gemacht.

Weitere Sprachelemente III

Datentypen:

- `enum { a, b, c};` definiert ganzzahlige Konstanten, die vom Compiler automatisch aufsteigende Werte zugewiesen bekommen. Mit `enum { a=10, b=32, c=85};` können auch feste Werte zugewiesen werden. `enum` eignet sich für `switch()` und als Ersatz für `#define` (Vorteil: Typüberprüfung)
- `_Bool var;` definiert eine Variable vom Typ Wahrheitswert (wahr oder falsch, 0 oder 1). Neu in C99. Mit dem Header `stdbool.h` kann auch die elegantere Schreibweise `bool var;` verwendet werden.

Sprungbefehle:

- Mit der `switch` Anweisung hatten wir bereits den Befehl `break` kennengelernt, der den `switch`-Block verläßt. `break` funktioniert auf gleiche Weise auch bei `for()` und `do ... while() ...;`

Weitere Sprachelemente IV

- `continue` unterbricht ebenfalls Schleifen, springt aber nicht ganz hinaus, sondern zur Bedingung von `while()` oder zum Inkrement von `for()`
- Mit `goto name;` kann zu einer beliebigen Stelle gesprungen werden, die mit dem Label `name: befehl;` markiert ist. Die Verwendung von `goto` gilt oft als schlechter Programmierstil.

Präprozessor I

- Bisher wurde die Übersetzung eines C-Programms in ausführbaren Code als ein Schritt angesehen. Tatsächlich zerfällt sie aber in mehrere Teile:

  Preprocessing: Der Programmtext wird vorverarbeitet. Dabei werden mit `#include` aufgelistete Dateien eingefügt, mit `#define` benannte Konstanten ersetzt, Kommentare entfernt, und mehr ...

  Compiling: Der entstandene reine C-Code wird in Maschinensprache übersetzt. Allerdings ist das Programm noch nicht in einer Form, die das Betriebssystem versteht.

  Linking: Das übersetzte Programm wird mit Systemdateien verbunden (to link), d.h. der entsprechende Code wird direkt angehängt (statisch gelinkt) oder Verweise auf Systemdateien eingebaut (dynamisch gelinkt). Das Ergebnis wird in das systemspezifische Format ausführbarer Dateien gebracht.

Präprozessor II

Wie angedeutet, bietet der Präprozessor weitere Möglichkeiten den Übersetzungsprozeß zu beeinflussen:

- Alle Präprozessor-Befehle beginnen mit `#`
- `#include <name>` bindet die Datei `name` an dieser Stelle ein. Gesucht wird in *Systempfaden*.
- `#include "name"` bindet die Datei `name` an dieser Stelle ein. Gesucht wird im *aktuellen (Nutzer-)Pfad*.
- `#define name ersatztext` definiert eine Konstante. Der Präprozessor sucht nach `name` und ersetzt durch `ersatztext` (ähnlich wie die Suche-Ersetze-Funktion eines Editors).
- Der Ersatztext kann auch echten C-Code enthalten:
  ```
  #define forever for(;;)
  ```
- Darüberhinaus kann `#define` mit Parametern umgehen. Damit können sogenannte Makros definiert werden:
  ```
  #define max(A,B) ((A)>(B) ? (A) : (B))
  ```

Präprozessor III

- Makros entsprechen minimalistischen Funktionen, die direkt in den Programmtext eingefügt werden. Argumente haben keinen festen Typ.
- Achtung: Bei Konstruktion und Aufruf von Makros ist auf korrekte Klammerung und die tatsächliche Auswertung des entstehenden Ausdrucks zu achten:

```
#define square(x) x*x /* FALSCH */
max(i++,j++) /* FALSCH */
```

- Bedingte Übersetzung: Abhängig von vorher definierten Konstanten kann gegebener Code eingebaut werden, oder nicht.

```
#define SYS 2
#if SYS == 0
    Programmcode
#elif SYS == 1
    Programmcode
#else
    Programmcode
#endif
```

Präprozessor IV

- Neben `#if` gibt es verschiedene Varianten, die prüfen, ob eine Konstante definiert ist, oder nicht:

```
#if defined(CONST)
#ifdef CONST
#if !defined(CONST)
#ifndef CONST
```

- Konstanten können auch über Optionen beim Aufruf des Compilers definiert werden:

```
gcc -DCONST=5 ...
```

- Im Programm kann man anschließend testen, ob eine Konstante bereits definiert ist, oder die Definition andernfalls nachholen:

```
#ifndef CONST
#define CONST 4
#endif
```

- Der Präprozessor wird auch verwendet, um den Compiler zu steuern, z.B. zur Parallelverarbeitung mit OpenMP

```
#pragma omp ...
```

Kommandozeilen-Parameter I

- In einzelnen Beispiel-Programmen und bei der Arbeit mit Octave wurden Kommandozeilen-Parameter verwendet.
- Wie erläutert, ruft das Betriebsystem beim Programmstart die Funktion `main()` auf. Dabei werden die komplette Kommandozeile als Array von Zeichenketten und die Länge des Arrays als Argumente übergeben:

```
#include <stdio.h>

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

   int i;

   for(i=0; i<argc; i++)
      printf("%3i %s\n", i, argv[i]);

   return 0;
}
```

Kommandozeilen-Parameter II

- Nach Übersetzung liefert "`a.out 1 5776.3 hallo.txt`":

```
0 a.out
1 1
2 5776.3
3 hallo.txt
```

- Die Namen `argc` und `argv` sind verbreitete Konvention. Selbstverständlich kann man auch andere Bezeichnungen verwenden.
- `argc` ist immer mindestens 1, da immer der Programmname als `argv[0]` übergeben wird.
- Die übergebenen Zeichenketten können mit Bibliotheksfunktionen in andere Datentypen umgewandelt oder direkt verwendet werden:

```
i = atoi(argv[1]);
d = atof(argv[2]);
input = fopen(argv[3],"r");
```

Dynamischer Speicher I

- Bislang wurden nur Arrays benutzt, deren Länge zur Zeit der Übersetzung feststand.

```
double x[10], y[] = {1,2,3.4};
```

- Die erlaubte Dimension solcher festen Arrays ist begrenzt (mein Laptop z.B. 1000000). Die maximale Länge ist viel kleiner als der tatsächlich verfügbare Speicher.
- Bei den meisten Programmen ist die Dimension von Arrays variabel (Eingabeparameter). Außerdem besteht gelegentlich großer Speicherbedarf.
- Ausweg: dynamische Speicherbelegung, d.h. Anforderung freien Speichers vom Betriebsystem zur Laufzeit.
- Die Standard-Bibliothek liefert in `stdlib.h` entsprechende Funktionen:

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Dynamischer Speicher II

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

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

  int i, n;
  double *x;

  if(argc>1) {
    n = atoi(argv[1]);
  } else {
    printf("Fehler! n angeben.\n");
    exit(EXIT_FAILURE);
  }

  x = (double *) calloc(n, sizeof(double));
```

Dynamischer Speicher III

```
    if(x==NULL) {
        printf("Fehler! Nicht genug Speicher.\n");
        exit(EXIT_FAILURE);
    }

    for(i=0; i<n; i++) x[i] = i;

    free(x);

    return 0;
}
```

Übung 4

Definieren Sie einen neuen Datentyp für rationale Zahlen. Schreiben Sie Funktionen zum Erzeugen & Kürzen (Stichwort: Euklidischer Algorithmus), Addieren und Multiplizieren solcher Zahlen.

Berechnen Sie:

$$\frac{8883}{5454} + \frac{432}{32} = ?$$
$$\frac{8883}{5454} \times \frac{432}{32} = ?$$
$$\frac{554554}{32409} + \frac{744}{837} = ?$$

Gliederung

Programm-Design I

- Bevor wir uns eingehender mit Algorithmen und komplizierteren Programmen befassen, sollen noch einmal allgemeine Hinweise zur Programmentwicklung gegeben werden.
- Angenommen, wir verstehen das phys. Problem, kennen passende Algorithmen, besitzen einen geeigneten Computer:

  Wie entsteht daraus ein Programm?

  Wie sollte das Programm beschaffen sein?
- Wichtige Anforderungen: Ein Programm sollte...
  - einfach und lesbar sein. Die Funktion jedes Teils sollte klar und verständlich sein.
  - sich selbst dokumentieren, so daß der Programmierer und andere es verstehen.
  - leicht zu benutzen sein.
  - einfach zu ändern und auf andere Rechner zu portieren sein.
  - an andere weitergebbar und veränderbar sein.
  - richtige Ergebnisse liefern.

Programm-Design II

- Diese Ziele können durch einen modularen, sogenannten top-down Programmierstil erreicht werden:
  1. Beim modularen Programmieren wird jede Aufgabe in Teilaufgaben zerlegt. Dieser Prozeß wird solange fortgesetzt, bis kleinere, überschaubare Teilaufgaben definiert sind. Programme werden dadurch übersichtlicher und einfacher. Außerdem kann Arbeit auf mehrere Programmierer verteilt werden.
     - 1.1 Schreiben Sie mehrere kleine Unterprogramme, die begrenzte Aufgaben erfüllen.
     - 1.2 Jede Untereinheit sollte wohldefinierte Aus- und Eingabewerte haben, die als Argumente übergeben werden
     - 1.3 Unterprogramme sollten weitgehend unabhängig voneinander sein. Dadurch können sie einzeln getestet und wiederverwendet werden.
     - 1.4 *Aber:* Die Zerlegung in Unterprogramme sollte nicht übertrieben werden. Viele Funktionsaufrufe kosten Zeit. (vgl. `inline`)
  2. Schieben Sie das eigentliche Programmieren, d.h. die Eingabe des Programms, so weit wie möglich hinaus. Konzentrieren Sie sich darauf, das Problem zu definieren und zu verstehen. Überlegen Sie sich, welche logischen Schritte zu seiner Lösung erforderlich sind.

Programm-Design III

3. Wählen Sie den verlässlichsten und einfachsten Algorithmus. Wichtig sind zuerst richtige Ergebnisse, danach Geschwindigkeit.
4. Seien Sie sich bewußt, daß die Eignung eines Algorithmus auch von der verfügbaren Hardware abhängt (Skalar-, Parallel-, Vektorcomputer)
5. Klare und einfache Programme haben am Ende meist weniger Fehler. Das schreiben klarer Programme dauert zwar länger, man spart aber bei der Fehlersuche (die leider einen großen Teil der Programmierarbeit ausmacht).
6. Die Planung des Programms sollte von oben nach unten (*top-down*) erfolgen, d.h. man identifiziert zuerst die wesentlichen Aufgaben und behält stets das *big picture* im Auge:
   - 6.1 Entwerfen Sie die Grobstruktur des Programms. Ordnen Sie die Hauptaufgaben in der Reihenfolge an, in der sie auszuführen sind.
   - 6.2 Zerlegen Sie die Hauptaufgaben in Teilaufgaben, dies führt auf sinnvolle Unterprogramme oder Gruppen von Unterprogrammen.
   - 6.3 Falls nötig, verfeinern Sie den Entwurf weiter.
7. Versuchen Sie, den Programmablauf linear zu halten. Umherspringen ist sehr verwirrend.

Programm-Design IV

- Setzen Sie den Programmentwurf in ein strukturiertes Programm um. Strukturierung bedeutet, daß
  1. die Daten sinnvoll angeordnet sind, d.h. sich z.B. am physikalischen Inhalt orientieren und zum verwendeten Algorithmus passen,
  2. der Programmablauf mit den üblichen Elementen strukturiert ist (Funktionen, Schleifen, Fallunterscheidungen),
  3. der Code übersichtlich editiert und kommentiert ist.
- Beachten Sie folgende praktischen Ratschläge:
  1. Bewahren Sie stets eine lauffähige Version Ihres Programms. Machen Sie Änderungen an einer Kopie.
  2. Halten Sie sich an den Standard der verwendeten Programmiersprache. Verzichten Sie auf spezielle Spracherweiterungen bestimmter Compiler. Der Code wird dadurch portabel und langlebiger.
  3. Fügen Sie ausreichend Kommentare ein. Geben Sie zu jedem Unterprogramm bzw. jeder Funktion eine kurze Beschreibung an.
  4. Vergeben Sie sinnvolle Variablennamen und folgen Sie den üblichen Konventionen der Benennung phys. Größen (z.B. `mass` oder `temp`). Erläutern Sie die Variablen in Kommentaren.

Programm-Design V

5. Vermeiden Sie globale Variablen. Bei diesen ist oft unklar, woher Sie ihren Wert bekommen, wer zugreift etc.
6. Beachten Sie, daß auch Compiler fehlerhaft sein können. Testen Sie Ihr Programm mit verschiedenen Optimierungsstufen und eventuell mit verschiedenen Compilern oder unterschiedlichen Computern.

Übung 5

- Als kleine Zusammenfassung des C-Grundkurses soll ein einfaches Problem Schritt für Schritt in ein Programm umgesetzt werden: Die Umrechnung von Temperaturen von Grad Celsius in Grad Fahrenheit und umgekehrt.
- Ziel: Ein Programm mit folgender Fertigkeit:

  Eingabe: 85.5 F
  Ausgabe: 29.72 C

  Eingabe: 45.2 C
  Ausgabe: 113.36 F

Gliederung

Sortieren

- Das Sortieren von Zahlen oder Zeichenketten ist eine regelmäßig zu lösende Aufgabe. Beispiel: Sortieren von Spielkarten.
- Anhand von Sortieralgorithmen können auch Untersuchungen zur Laufzeit und zum Resourcenverbrauch von Algorithmen gemacht werden.
  - Viele einfache Algorithmen benötigen N^2 Vergleiche, um N Objekte zu sortieren.
  - Die besten Algorithmen kommen mit $N \log N$ Vergleichen aus. Dies ist typisch für Algorithmen, die auf dem Prinzip „Teile und Herrsche“ beruhen.
- Insbesondere beim Sortieren von und Suchen nach Worten werden wir neue Datenstrukturen kennenlernen (Verkettete Listen, Bäume)
  - Sortieren erfordert das Umordnen von Objekten im Speicher. Bei Verwendung verketteter Listen muß nur die Verkettung geändert werden.
  - Geordnete Objekte können in schnell zu durchsuchenden Bäumen gespeichert werden.

Selection sort

Einer der EINFACHSTEN SORTIERALGORITHMEN für eine Liste von Objekten kann wie folgt beschrieben werden:

1. Suche das kleinste Element der Liste.
2. Vertausche es mit dem ersten Element der Liste.
3. Wiederhole die Prozedur mit der verbleibenden Liste ohne das erste Element, d.h. gehe zu Schritt 1 (sofern es noch unsortierte Objekte gibt).

Frage: Wieviele Vergleiche und wieviele Vertauschungen werden zum Sortieren einer Liste von N Objekten benötigt?

Antwort: Etwa $N^2/2$ Vergleiche und N Vertauschungen.

Insertion sort

- Ein WEITERER SORTIERALGORITHMUS wird von vielen Spielern benutzt, um die Karten auf der Hand zu ordnen:
  1. Gehe die unsortieren Objekte der Reihe nach durch.
  2. Füge jedes Element an der richtigen Stelle in die Liste der bereits sortierten ein.
- Dieser Algorithmus erfordert die Suche im bereits sortierten Teil der Liste.
- Außerdem müssen beim Einfügen die Elemente oberhalb des einzufügenden bewegt werden.
- Aufwand: $N^2/2$

Quicksort I

- Der vermutlich am häufigsten verwendete Sortieralgorithmus ist Quicksort, der 1960 von C.A.R. Hoare entwickelt wurde.
- Die Performance von Quicksort ist im Detail verstanden. Es handelt sich um eines der schnellsten allgemein verwendbaren Verfahren.
- Trotzdem läßt sich Quicksort relativ leicht umsetzen:
  1. Wähle ein Element x aus der Liste.
  2. Bilde zwei Stapel: einen aus allen Elementen, die kleiner als x sind, einen zweiten mit allen anderen Elementen.
  3. Wende Quicksort auf die beiden Stapel an, d.h. gehe jeweils zu Schritt 1. Das Verfahren bricht ab, wenn nur noch Stapel mit weniger als zwei Elementen übrig sind.
- Quicksort folgt dem Prinzip „Teile und Herrsche“, d.h. ein Problem der Größe N wird in zwei Teilprobleme der Größe $N/2$ aufgeteilt (ungefähr). Diese Aufteilung wird fortgesetzt, bis nur noch triviale Probleme übrig bleiben.

Quicksort II

- Nimmt man an, daß die Aufteilung der N-elementigen Liste in zwei Stapel ungefähr N Operationen benötigt, ergibt sich die Gesamtlaufzeit aus der Rekursion:

$$T(N) = 2T(N/2) + N$$

- Die fortgesetzte Zweiteilung wird etwa $\log_2(N)$-mal ausgeführt. Der Gesamtaufwand ist also von der Größenordung

$$T(N) \approx N \log_2 N$$

- Ein ähnliches Verhalten beobachtet man auch bei anderen hocheffizienten Algorithmen, z.B. Fast Fourier Transformation (FFT)

Suchen

- Fast noch elementarer und häufiger benötigt als Sortieren ist das Suchen nach Daten.
- Die Suche kann erleichtert werden, wenn die Daten in geeigneter Form gespeichert sind. Beispiele sind geordnete Listen und binäre Bäume.
- Der einfachste Algorithmus ist die Sequentielle Suche: Angenommen die Daten sind als Array gespeichert, dann gehe das Array von Anfang bis Ende durch und stoppe, falls der gesuchte Wert gefunden wurde.
- Offenbar sind im schlechtesten Fall N Schritte nötig (insbesondere bei erfolgloser Suche), im Mittel ungefähr $N/2$.

Verkettete Listen I

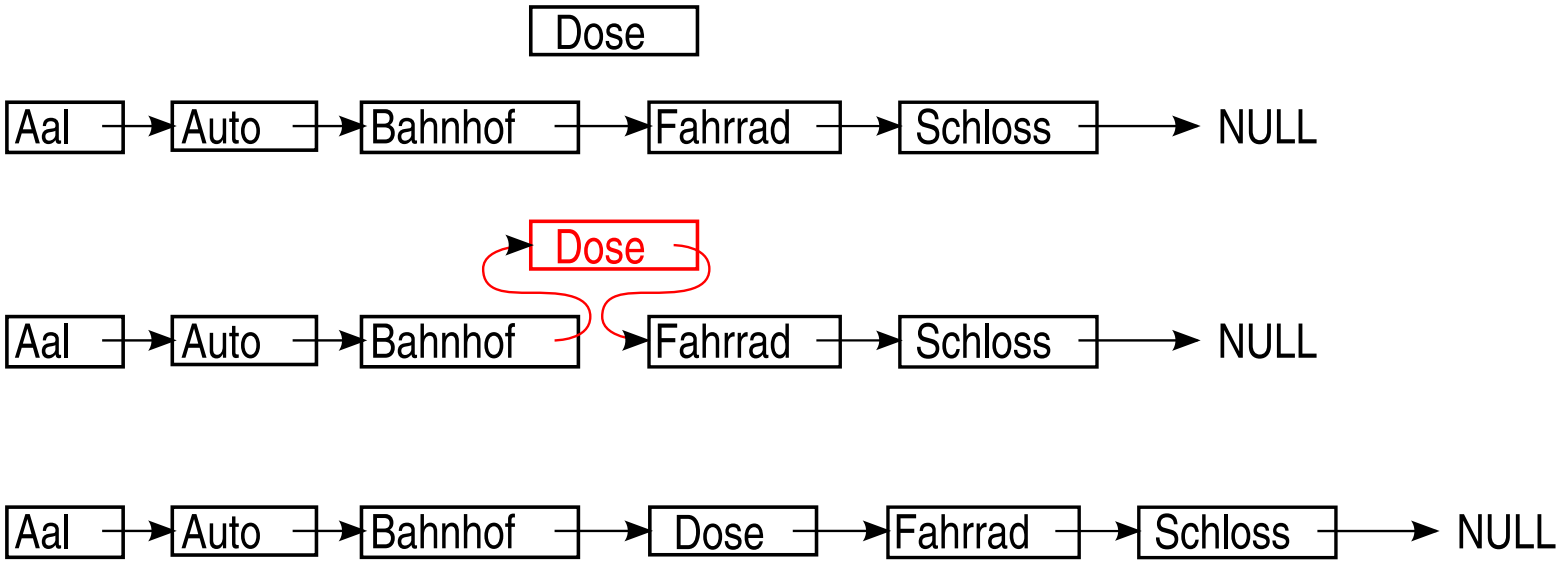
- Zahlen oder andere Objekte einheitlicher Länge lassen sich gut in Arrays abspeichern. Für Text oder einzelne Worte sind verkettete Listen häufig die bessere Wahl.
- Jedes Element einer verketteten Liste besteht aus der gespeicherten Information und einem Zeiger auf das nächste Element der Liste. Bsp.:

```
typedef struct listelem listelem;
struct listelem {
  char *text;
  listelem *next;
}
```

- Die Liste beginnt mit einem Zeiger auf das erste Element. Alle weiteren werden ermittelt, indem jeweils dem Zeiger `next` gefolgt wird. Das Ende der Liste wird durch einen leeren Zeiger (`next = NULL`) angezeigt.

Verkettete Listen II

- Ist die Liste geordnet, kann eine Suche abgebrochen werden, sobald ein Element größer als das gesuchte angetroffen wird.
- Eine geordnete Liste kann leicht erzeugt werden, indem zunächst nach dem hinzuzufügenden Element gesucht wird. Ist es noch nicht vorhanden, bricht die Suche bei nächstgrößeren Element ab.
- Das neue Element kann an dieser Stelle eingefügt werden, indem lediglich die betreffenden Zeiger angepaßt werden. Ein umkopieren der restlichen Elemente ist *nicht* erforderlich (vgl. Insertion sort).



Übung 6

Schreiben Sie ein Programm, das ein Array von N reellen Zahlen ordnet. Beginnen Sie mit einem einfachen Algorithmus wie *selection sort* oder *insertion sort*. Versuchen Sie dann *Quicksort*.

Binärsuche

- Zurück zu gewöhnlichen Arrays: Wenn die N Elemente des Arrays geordnet sind, kann die Suche nach einem Element x wesentlich beschleunigt werden, indem wieder das Prinzip „Teile und Herrsche“ angewendet wird:
  1. Vergleiche x mit dem Element $N/2$.
  2. Ist x größer, fahre mit der oberen Hälfte der Liste fort, andernfalls mit der unteren.
  3. Wiederhole die Prozedur mit der Teilliste.
- Da nur mit dem mittleren Element der jeweiligen Teilliste verglichen wird, erfordert diese sog. Binärsuche lediglich um die $\log_2 N$ Operationen.
- Sind die Elemente des Arrays *gleichmäßig verteilt*, kann die Suche noch weiter verbessert werden: Statt die Liste in der Mitte zu teilen, bestimmen wir den Schnittpunkt durch Interpolation.
  1. Berechne aus erstem und letztem Element der Liste eine Interpolationsgerade und schätze die Position von x.
  2. Vergleiche x mit dem Element an der geschätzten Position und fahre mit der oberen oder unteren Teilliste fort.
- Der Aufwand schrumpft damit auf $\log_2 \log_2 N$.

Binärsuche – Code

Die Suche in einem geordneten Array ganzer Zahlen kann mit folgender Funktion realisiert werden:

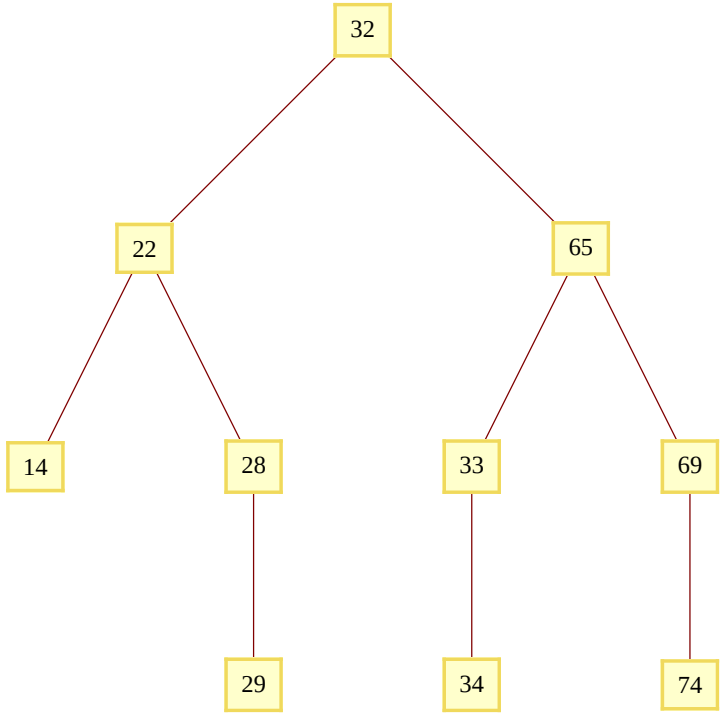
```
int binsearch(int x, int n, int *d) {

  int i, l, r;

  l = 0;
  r = n-1;

  do {
    i = (r+l)/2;
    if(x < d[i]) {
      r = i-1;
    } else if(x > d[i]) {
      l = i+1;
    }
  } while (x!=d[i] && l<=r);

  return (x==d[i]) ? i : -1;
}
```

Binärsuche – Beispiel

- Angenommen, wir suchen in der Liste:

  `14 22 28 29 32 33 34 65 69 74`

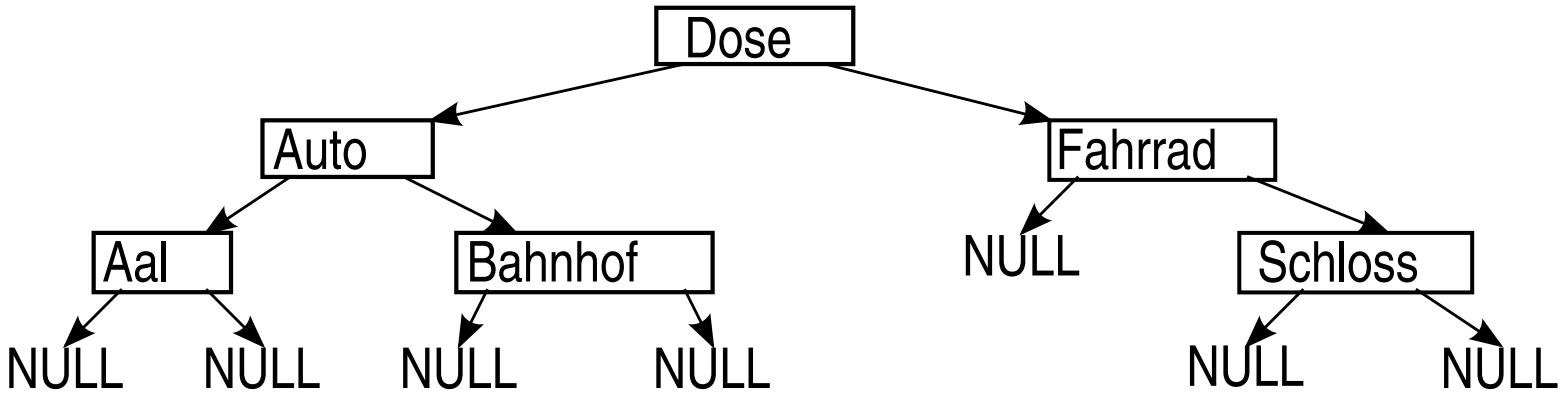
- Die Funktion `binsearch()` vergleicht der Reihe nach mit folgenden Zahlen



- Diese Struktur führt uns auf verwandte Algorithmen für Textdaten

Binäre Suchbäume

- Auf geordnete verkettete Listen läßt sich Binärsuche nicht direkt anwenden, da die Mitte nicht ermittelt werden kann.
- Ausweg: Speicherung der Daten als BINÄRER BAUM. Jeder Knoten hat nicht nur einen Zeiger `next`, sondern *zwei*: `links` und `rechts`.



- Die Suche im Baum funktioniert ähnlich wie in einer Liste. Allerdings folgt man dem linken Zeiger, wenn das gesuchte Element kleiner als der aktuelle Eintrag am Knoten ist, oder dem rechten, wenn es größer ist.
- Stößt man auf das gesuchte Element, endet die Suche. Stößt man auf `NULL`, wird ein neuer Knoten eingebaut.
- Ist der Baum balanciert, wächst der Suchaufwand wie $\log_2 N$.

Hashing I

- Speichern und Suchen von Daten in binären Bäumen sind bereits sehr effizient. Bei größeren Datenmengen sind jedoch noch relativ viele Vergleiche nötig und man wünscht sich einen schnelleren Zugriff auf die Daten.
- Praktisch wäre eine direkte Umrechnung vom Datensatz auf die Position im Speicher. Durch eine sog. HASH-FUNKTION wird dies näherungsweise erreicht.
- Eine Hash-Funktion sollte jedem Datensatz einen Indexwert aus dem Bereich $[0, M-1]$ zuordnen. Gibt es mehr als M Datensätze, sollte die Verteilung auf die Hashwerte möglichst gleichmäßig sein, d.h. so wenige Datensätze wie möglich sollten den gleichen Index liefern.
- Hash-Funktionen sind oft ähnlich konstruiert, wie Generatoren von Pseudo-Zufallszahlen (lineare Kongruenz). Beispiel einer Hash-Funktion für Zeichenketten:

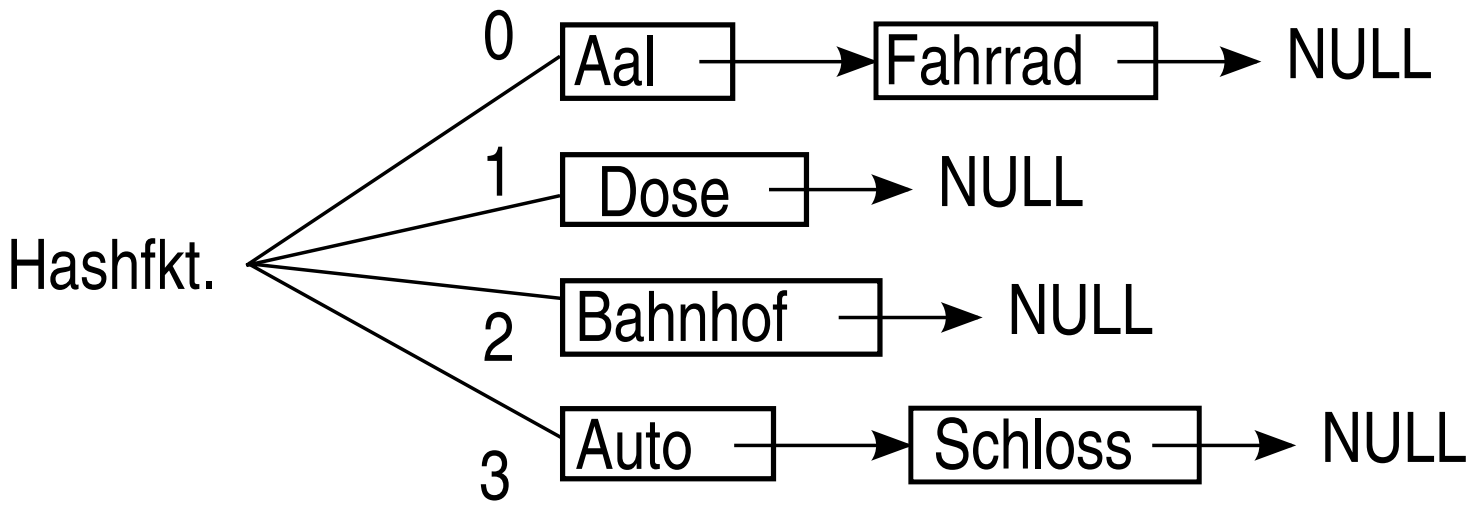
Hashing II

```
int hashfun(char *s) {

  int h;

  for(h=0; *s; s++) h = (F*h + (*s)) % M;

  return h;
}
```

- Der Faktor F entspricht der Größe des Zeichensatzes (z.B. 256) und M sollte prim sein. Alternativ kann auch F prim sein (z.B. 31 oder 37) und M beliebig.
- Auch kryptographische Algorithmen eignen sich oft als Hash-Funktion.
- Die Eignung einer Hash-Funktion hängt auch von den Daten ab.
- Um Text/Worte zu speichern und zu finden, legt man ein Array von verketteten Listen an. Der Hashwert gibt an, zu welcher Liste ein Datensatz gehört. Innerhalb der (kurzen) Liste von Datensätzen mit gleichem Hash kann sequentiell gesucht werden.

Hashing III

- Schematische Darstellung einer Hash-Struktur:



- Hier wurden die Worte aus dem Binärbaum in einen Hash der Länge $M = 4$ geordnet ($F = 31$). Die längsten Teillisten haben zwei Elemente, d.h. eine sequentielle Suche kommt schnell zum Ziel.
- Hashing wird oft bei großen Datenbanken verwendet. Die Suche erfordert nur $O(1)$ Operationen.
- Im Gegensatz zum Binärbaum ist eine sortierte Ausgabe der Daten nicht leicht zu realisieren.

Übung 7

Schreiben Sie ein Programm, das die Häufigkeit von Worten in einem Text analysiert. Die gefundenen Worte und ihr Zähler sollen in einem Baum gespeichert werden, so daß am Ende eine sortierte Liste ausgegeben werden kann.

Gliederung

Zahlendarstellung und Wertebereich I

- Bevor wir uns mit einfachen numerischen Verfahren beschäftigen, sollen zunächst die Eigenschaften von Zahlen und Fehler in numerischen Rechnungen diskutiert werden.
- Zahlen im Computer haben stets einen endlichen Wertebereich.
- Ganze Zahlen umfassen eine bestimmte Anzahl Bits, genauere Informationen liefert `sizeof`:

Typ	Bytes	Bits
short	2	16
int	4	32
long	4	32
long long	8	64

- Von N Bits wird eins für das Vorzeichen verwendet (sofern nicht `unsigned` definiert), d.h. eine Variable x unterliegt der Beschränkung:

$$-2^{N-1} \leq x < 2^{N-1}$$

Zahlendarstellung und Wertebereich II

- Wird dieser Bereich überschritten, findet ein sog. Überlauf statt. Überzählige Stellen werden abgeschnitten.
- Negative ganze Zahlen werden mit Hilfe des binären Komplements dargestellt:

```
  42 = 00101010
-42 = ~(42) + 1 = 11010110
```

- Dies hat den Vorteil, daß die gleichen elementaren Rechenoperationen für negative und positive Zahlen gelten:

```
 -1  =   11111111
  1  =   00000001
-----------------
  0  =  100000000
```

- Bei einem Überlauf werden aus positiven Zahlen plötzlich negative.

Zahlendarstellung und Wertebereich III

- Fliesskommazahlen bestehen aus drei Elementen: Vorzeichen, Mantisse und Exponent:

$$x = (-1)^{\text{Vorzeichen}} \times \text{Mantisse} \times 2^{\text{Exponent}}$$

- Je nach Datentyp werden für die drei Teile unterschiedlich viele Bits reserviert:

Typ	Vorz.	Mantisse	Exponent
float	1	23	8
double	1	52	11

- Der Exponent wird so gewählt, daß vor dem Komma stets 1 steht. Diese 1 wird nicht gespeichert (phantom bit). Außerdem enthält der Exponent konstanten Aufschlag $2^{N_e-1} - 1$:

$$\pi = 1.10010010000110\cdots \times 2^1 = \underbrace{0}_{\textit{sign}}\,\underbrace{10000000000}_{\textit{exponent}}\,\underbrace{10010010000110\ldots}_{\textit{mantissa}}$$

Zahlendarstellung und Wertebereich IV

- Der Betrag einer Fließkommazahl unterliegt also der Beschränkung:

$$\texttt{float}: \quad 10^{-38} \lesssim |x| \lesssim 10^{38}$$
$$\texttt{double}: \quad 10^{-308} \lesssim |x| \lesssim 10^{308}$$

- Wird dieser Bereich überschritten, erhält man auf „gutartigen“ Systemen den Wert `inf`, manchmal aber auch Unsinn
- Bei Unterschreitung des Zahlenbereichs wird auf „gutartigen“ Systemen von der Konvention der 1 vorm Komma abgewichen. Auf Kosten der Stellenzahl sind dann auch kleinere Zahlen der Form $0.00000001 \ldots 2^{-1022}$ möglich. Manche Rechnersysteme springen auf Null.
- Division durch Null führt meist auf den speziellen Wert `nan` (not a number).

Genauigkeit

- Mit Fließkommazahlen kann man über viele Größenordnungen mit der gleichen relativen Genauigkeit rechnen.
- Obwohl sich sehr kleine Zahlen in der Nähe von Null darstellen lassen, sagt dies nichts über den absoluten Fehler einer Rechnung aus.
- Ein besseres Maß für den absoluten Fehler liefert die Maschinengenauigkeit, die definiert ist als kleinstes ϵ mit

$$1 + \epsilon \neq 1$$

- Für die Standardtypen findet man:

$$\texttt{float}: \quad \epsilon \approx 10^{-7}$$

$$\texttt{double}: \quad \epsilon \approx 10^{-16}$$

Gliederung

Tricks für genauere Rechnungen I

- Bei unvorsichtiger Programmierweise kann sich die endliche Maschinengenauigkeit unangenehm auswirken. Man beachte deshalb folgende Hinweise:
- Differenzen ähnlich großer Zahlen sind ungenau und sollten vermieden werden:

$$e^{-x} = \sum_{k=0}^{\infty} \frac{(-x)^k}{k!} \xrightarrow{\text{besser}} 1/e^x$$

- Potentiell gefährliche Ausdrücke sollten umformuliert werden:

$$\frac{1-\cos x}{x^2} \xrightarrow{\text{besser}} \frac{1}{2}\left[\frac{\sin x/2}{x/2}\right]^2$$

- Summen sollten mit dem kleinsten Beitrag begonnen werden:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$$

Tricks für genauere Rechnungen II

- Gelegentlich helfen auch Tricks wie KOMPENSIERTE SUMMATION:

$$s = a + b \qquad \text{wird zu} \qquad \begin{aligned} t &= a + b \\ e &= (t - a) - b \\ s &= t - e \end{aligned}$$

Summen I

- Bei den oben als Beispiel angegebenen Summen handelte es sich um unendliche Summen. Eine numerische Rechnung muß naturgemäß nach endliche vielen Termen abgebrochen werden. Wo?
- Ein mögliches Kriterium ist das Verhältnis des kleinsten berücksichtigten Terms zur Gesamtsumme.
- Noch genauere Ergebnisse erhält man durch eine Abschätzung des Restgliedes $R(N)$:

$$\sum_{k=0}^{\infty} f(k) = \sum_{k=0}^{N} f(k) + R(N)$$

Summen II

- Für eine im Intervall $[0,1]$ $2n$-mal stetig differenzierbare Funktion $f(x)$ und eine Konstante $0 < \theta < 1$ gilt:

$$\int_0^1 f(t)\,dt = \frac{1}{2}\left[f(1)+f(0)\right] - \sum_{k=1}^{n} \frac{B_{2k}}{(2k)!}\left[f^{(2k-1)}(1) - f^{(2k-1)}(0)\right] + \frac{B_{2n}}{(2n)!} f^{(2n)}(\theta)\,,$$

  wobei B_n die Bernoulli-Zahlen bezeichnet.

- Dieser Ausdruck folgt durch mehrfache partielle Integration unter Verwendung der Bernoulli-Polynome mit $B_n'(x) = nB_{n-1}(x)$ (und $B_n = B_n(0)$).

Summen III

- Wird ein Intervall $[a, b]$ in m gleichgroße Intervalle der Länge $h = (b-a)/m$ zerlegt, ergibt sich die Euler-Maclaurin-Formel:

$$\sum_{k=0}^{m} f(a+hk) = \frac{1}{h}\int_a^b f(t)\,dt \;+\; \frac{1}{2}\left[f(a)+f(b)\right] + \sum_{k=1}^{n-1} \frac{h^{2k-1}B_{2k}}{(2k)!}\left[f^{(2k-1)}(b) - f^{(2k-1)}(a)\right] + \frac{h^{2n}B_{2n}}{(2n)!}\sum_{k=1}^{m-1} f^{(2n)}(a+kh+\theta h)$$

- Das Integral und wenige Terme der Euler-Maclaurin-Reihe liefern eine sehr genaue Abschätzung des Restglieds $R(N)$.
- Diese Methode wird häufig benutzt, um spezielle Funktionen zu berechnen, die durch unendliche Summen definiert sind.

Übung 8

- Untersuchen Sie, wann Fließkommazahlen vom Typ `float` bzw. `double` über- bzw. unterlaufen.
- Bestimmen Sie die Maschinengenauigkeit für beide Datentypen.
- Berechnen Sie numerisch $(1 - \cos x)/x^2$ für kleine Werte x. Wird für $x \to 0$ der richtige Grenzwert erreicht?
- Falls noch viel Zeit ist: Schreiben Sie ein Programm zur Berechnung der Riemannschen Zeta-Funktion $\zeta(x)$ für $x > 1$.

Gliederung

Differenzieren I

- Die wichtigsten Formeln zur NUMERISCHEN BERECHNUNG VON ABLEITUNGEN wurden bereits im letzten Semester vorgestellt. Grundlage ist üblicherweise eine Taylor-Entwicklung:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \dots$$

- Hier noch einmal eine Zusammenfassung sog. 3-Punkt-Formeln:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h) \qquad \text{Vorwärts-Diff.}$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h) \qquad \text{Rückwärts-Diff.}$$

$$f'(x) = \frac{f(x+h/2) - f(x-h/2)}{h} + O(h^2) \qquad \text{Zentrale Diff.}$$

- Die zentrale Differenz liefert eine genauere Approximation der Ableitung.

Differenzieren II

- Man kann auch mehr als zwei Punkte zur Berechnung der Ableitung heranziehen, um den Fehler weiter zu reduzieren:

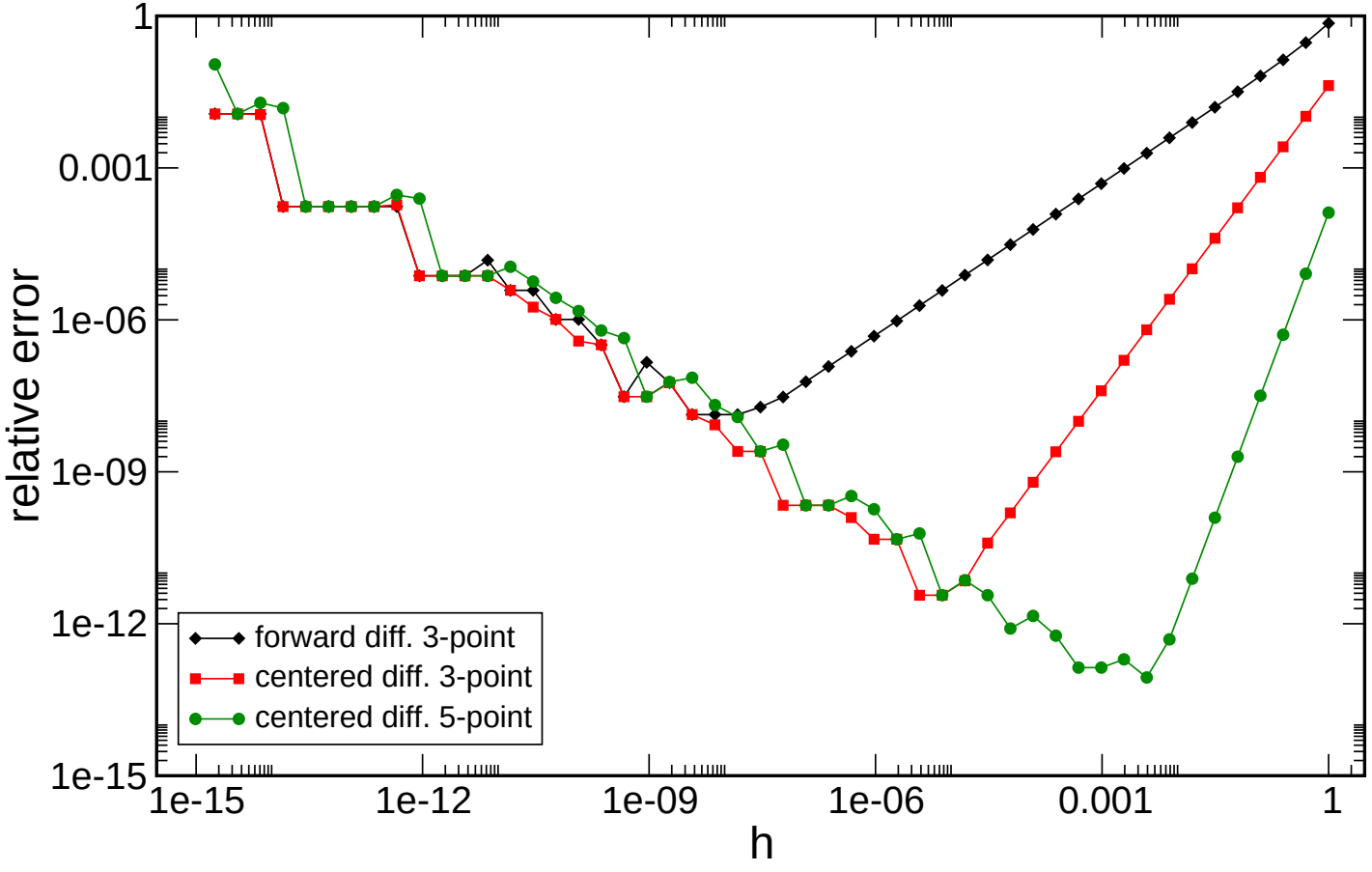
$$\frac{f(x+h/4)-f(x-h/4)}{h/2} = f'(x) + \frac{h^2}{96} f^{(3)}(x) + \dots$$

$$\frac{f(x+h/2)-f(x-h/2)}{h} = f'(x) + \underbrace{\frac{h^2}{24} f^{(3)}(x)}_{O(h^2)} + \dots$$

$$\frac{8[f(x+h/4)-f(x-h/4)] - [f(x+h/2)-f(x-h/2)]}{3h} = f'(x) - \underbrace{\frac{h^4}{7680} f^{(5)}(x)}_{O(h^4)} + \dots$$

Differenzieren III

- Achtung: In den Ableitungsformeln werden Differenzen ähnlich großer Zahlen gebildet. Für zu kleines h wächst der Fehler des berechneten $f'(x)$ wieder an, im schlimmsten Fall ergibt sich $f'(x) = 0$.



Differenzieren IV

- Höhere Ableitungen und Differentialoperatoren wie Δ erhält man durch Verschachtelung der behandelten Ausdrücke für erste Ableitungen.
- Beispiele:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

$$f''(x) = \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2} + O(h^4)$$

$$\Delta f(x,y) = \frac{f(x+h,y) + f(x-h,y) + f(x,y+h) + f(x,y-h) - 4f(x,y)}{h^2} + O(h^2)$$

Gliederung

Integrieren I

- Bei der NUMERISCHEN INTEGRATION geht man im Prinzip zur ursprünglichen Definition des Integrals als Summe zurück:

$$\int_a^b w(x) f(x)\, dx = \sum_{i=0}^{n} w_i f(x_i) + R(n)$$

  Hierbei ist $w(x) > 0$ eine Gewichtsfunktion; im einfachsten Fall gilt $w(x) = 1$.

- Eine Schätzung

$$Q_n(f) = \sum_{i=0}^{n} w_i f(x_i)$$

  heißt Interpolations-Quadraturformel, wenn sie für Polynome bis zum Grad n exakt ist, d.h. $R(n) = 0$.

Integrieren II

- Zu vorgegebenen Stützstellen gibt es genau eine solche Formel mit

$$w_i = \int_a^b w(x) L_i(x)\, dx$$

  Hierbei steht $L_i(x)$ für die Lagrangeschen Interpolationspolynome,

$$L_i(x) = \prod_{\substack{k=0 \\ k \neq n}}^{n} \frac{x - x_k}{x_i - x_k}$$

- Interpolations-Quadraturformeln mit äquidistanten Stützstellen heißen Newton-Cotes-Formeln

Integrieren III

- Beispiel: Bei der Trapez-Regel werden für das Intervall $[0, 1]$ die Stützstellen $x_0 = 0$ und $x_1 = 1$ gewählt. Mit $w(x) = 1$ ergeben sich $w_0 = w_1 = 1/2$, also:

$$\int_0^1 f(x)\,dx = [f(0) + f(1)]/2$$

- Bekanntermaßen ist diese Formel für Polynome vom Grade 1 (Geraden) exakt.
- Durch Aneinanderreihung mehrerer gleichgroßer Intervalle erhält man die erweiterte Trapezregel für viele Stützstellen:

$$\int_a^b f(x)\,dx = h\left(\frac{f_0}{2} + f_1 + \cdots + f_{n-1} + \frac{f_n}{2}\right)$$

Integrieren IV

- Die nächsthöhere Newton-Cotes-Formel ist die Simpson-Regel:

$$\int_0^1 f(x)\,dx = [f(0) + 4f(1/2) + f(1)]/6$$

- Noch genauere Integrationsformeln ergeben sich, wenn man die Stützstellen nicht fest vorgibt, sondern optimal wählt ...
- Eine Schätzung

$$G_n(f) = \sum_{i=0}^{n} w_i f(x_i)$$

heißt QUADRATURFORMEL VOM GAUSSSCHEN TYP, wenn sie für Polynome bis zum Grad $2n+1$ exakt ist, d.h. $R(n) = 0$.

- Zu jeder vorgegebenen Gewichtsfunktion $w(x)$ gibt es genau eine Gaußsche Quadraturformel.

Integrieren V

- Die Stützstellen sind nicht mehr frei wählbar, sondern müssen den n Nullstellen des $(n+1)$-ten bzgl. $w(x)$ orthogonalen Polynoms $p_{n+1}(x)$ entsprechen.
- Die Gewichte ergeben sich aus:

$$w_i = \frac{1}{p_n(x_i)p'_{n+1}(x_i)} \int_a^b w(x)p_n(x)^2\,dx$$

- Je nach Wahl von $w(x)$ erhält man die nach den bekannten Klassen orthogonaler Polynome benannten Verfahren: Gauß-Legendre, Gauß-Chebyshev, Gauß-Laguerre, etc.
- Für die Berechnung der Nullstellen gibt es fertige Programme oder analytische Formeln (siehe z.B. Abramowitz, Stegun, *Handbook of Mathematical Functions*)

Übung 9

- Leiten Sie eine Funktion Ihrer Wahl numerisch ab. Vergleichen Sie verschiedene Ableitungsformeln.
- Integrieren Sie eine Funktion mit den erweiterten Trapez- und Simpson-Regeln. Falls Zeit bleibt, werden auch Codes zur Gauss-Integration vorgestellt.

Gliederung

Fast Fourier Transformation (FFT) I

- Die (Schnelle) Fourier Transformation ist einer der wichtigsten arithmetischen Algorithmen mit Anwendungen in fast allen Gebieten der Informationstechnologie und der Naturwissenschaften:
- Lineare Differentialgleichungen werden im Fourierraum zu leicht lösbaren algebraischen Gleichungen

$$\partial_x^2 f(x) = g(x) \quad \leftrightarrow \quad -k^2 f(k) = g(k)$$

- Spektroskopische Meßverfahren arbeiten im Fourierraum; Zeit- und ortsaufgelöste Daten werden im Fourierraum analysiert
- Die Faltung zweier Funktionen wird zur Multiplikation
- Viele Algorithmen der Bildverarbeitung und Kompression arbeiten im Fourierraum (z.B. jpeg)
- Die Multiplikation großer Zahlen und Polynome kann mit Fouriermethoden beschleunigt werden

Fast Fourier Transformation (FFT) II

- Die Grundformel für die sog. diskrete Fouriertransformation von N Datenpunkten lautet:

$$f_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \exp(-\frac{2\pi\,\mathrm{i}}{N} jk)\ f_j$$

$$f_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \exp(\frac{2\pi\,\mathrm{i}}{N} jk)\ f_k$$

$$\text{mit} \quad j, k = 0, \ldots, (N-1)$$

- Diese beiden Transformationen sind die Inverse der jeweils anderen (Beweis als Übung)
- Andere Formen der Transformation können meist auf diese Form zurückgeführt werden
  - diskretisierte Fourier-Integrale
  - diskrete Sinus- und Cosinus-Transformationen

Fast Fourier Transformation (FFT) III

- Noch übersichtlicher wird die obige Definition der Fouriertransformation mit der Abkürzung $w_N := \exp(2\pi\,\mathrm{i}\,/N)$ [$=N$-te Einheitswurzel]:

$$f_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} w_N^{-jk} f_j$$

$$f_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} w_N^{jk} f_k$$

- Formal entprechen diese Gleichungen Matrix-Vektor-Multiplikationen, man erwartet also einen Rechenaufwand von der Ordnung $O(N^2)$
- Es zeigt sich jedoch, daß eine Teile-und-Herrsche-Strategie auf einen Algorithmus mit einem Aufwand der Ordnung $O(N \log N)$ führt.
- Weite Verbreitung fand diese sog. *Fast* Fourier Transform (FFT) durch die Arbeiten von Cooley und Tukey in den 1960ern. Die Ursprünge des Verfahrens finden sich aber schon bei Gauß.

Fast Fourier Transformation (FFT) IV

- Wie können die Fourier-Summen in zwei unabhängige Teile zerlegt werden?

$$
\begin{aligned}
f_j &= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} w_N^{jk} f_k \\
&= \frac{1}{\sqrt{N}} \left(\sum_{q=0}^{N/2-1} w_N^{j(2q)} f_{2q} + \sum_{q=0}^{N/2-1} w_N^{j(2q+1)} f_{2q+1} \right) \\
&= \frac{1}{\sqrt{2}} \left(\frac{1}{\sqrt{N/2}} \sum_{q=0}^{N/2-1} w_{N/2}^{jq} f_{2q} + w_N^j \frac{1}{\sqrt{N/2}} \sum_{q=0}^{N/2-1} w_{N/2}^{jq} f_{2q+1} \right) \\
&= \frac{1}{\sqrt{2}} \left(f_j^{\text{gerade}} + w_N^j f_j^{\text{ungerade}} \right)
\end{aligned}
$$

Fast Fourier Transformation (FFT) V

- Die beiden Terme in der letzten Zeile ergeben sich aus den jeweils $N/2$ Datenpunkte umfassenden Transformationen aller geraden und aller ungeraden Datenpunkte. Man beachte dabei die Periodizität $f_j^{(\text{un})\text{gerade}} = f_{j+N/2}^{(\text{un})\text{gerade}}$, die aus der Definition der Einheitswurzel folgt, $w_{N/2}^{N/2} = 1$.
- Entspricht die Zahl der Datenpunkte einer Zweierpotenz, $N = 2^p$, kann diese Zerlegung bis zu Summen mit einem Term fortgeführt werden. Die Transformation einer einzelnen Zahl ($N = 1$) ist die Zahl selbst.
- Die obige Zerlegung der Fourier-Transformation in zwei Teile kann leicht in ein rekursives Programm übersetzt werden, dessen Laufzeit wie $N \log N$ skaliert (siehe Übung).
- Bei professionellen FFT-Implementierungen wird die Rekursion üblicherweise aufgelöst. Außerdem entfällt meist die Beschränkung der Datensatzlänge N auf Zweierpotenzen, da auch effiziente Algorithmen für kleine Primzahlen existieren.

Polynom-Multiplikation mit FFT I

- Die klassischen Anwendungen der Fourier-Transformation zur Datenanalyse und bei Differentialgleichungen sind vermutlich bekannt.
- Wir betrachten stattdessen das Problem der Polynom-Multiplikation:

$$
\begin{aligned}
p(x) &= p_0 + p_1 x + \cdots + p_n x^n \\
q(x) &= q_0 + q_1 x + \cdots + q_m x^m \\
r(x) &:= p(x)q(x) = r_0 + r_1 x + \cdots + r_{n+m} x^{n+m}
\end{aligned}
$$

  Gegeben: p_j, q_j, Gesucht: r_j

- Generell läßt sich ein Polynom vom Grade n exakt rekonstruieren, wenn man seinen Wert an $n+1$ verschiedenen Stützstellen kennt. Man benötigt dazu die früher eingeführten Langrangeschen Interpolationspolynome $L_i(x)$.

Polynom-Multiplikation mit FFT II

- Für das Produkt $r(x)$ berechnet man $p(x)$ und $q(x)$ an $N = n + m + 1$ verschiedenen Stellen, multipliziert die Ergebnisse und rekonstruiert daraus $r(x)$.
- Normale Algorithmen zur Auswertung von Polynomen (Horner-Schema) und zur Berechnung des Interpolationspolynoms skalieren wie $O(N^2)$
- Mit FFT läßt sich der Rechenaufwand reduzieren auf $O(N \log N)$. Man muß lediglich von beliebigen Stützstellen zu den Einheitswurzeln für $N = n + m + 1$ übergehen.
- Die Auswertung von $p(x)$ und $q(x)$ an den Stützstellen $x_j = w_N^j$ entspricht der Fourier-Rücktransformation der Koeffizienten p_j und q_j.
- Statt aus Interpolationspolynomen erhält man die Koeffizienten von $r(x)$ aus der Fourier-Hintransformation von $p(x_j)q(x_j)$.

Übung 10

- Beweisen Sie, daß die beiden Formeln für die diskrete Fourier-Transformation invers zueinander sind.
- Schreiben Sie eine Funktion, die die komplexe, diskrete Fourier-Transformation für einen Datensatz der Länge $N = 2^p$, $p \in \mathbb{N}_0$ berechnet.

Gliederung

Programmbibliotheken I

- Bei vielen wiederkehrenden Programmieraufgaben ist es sinnvoll, auf existierenden Code zurückzugreifen, statt das Rad neu zu erfinden. Man verwendet fertige Programmbibliotheken.
- Solche Bibliotheken gibt es für nahezu jedes Problem: Numerische Rechnungen, Datenstrukturen, Interfaces, Graphik, ...
- Bei der Entscheidung für eine bestimmte Bibliothek sollte man auf *Portabilität* (Welche Betriebsysteme, Rechner?), *Verbreitung* (Exotisch oder allgemeiner Standard? Gut gepflegt oder veraltet?) und *Lizenzfragen* (Freie Software oder kommerziell?) achten.
- Eine wichtige Bibliothek wurde bereits behandelt:
  Die C Standard-Bibliothek.
- Bevor weitere Bibliotheken vorgestellt werden, einige allgemeine Hinweise zur Arbeit mit Bibliotheken:

Programmbibliotheken II

- Einbau ins Programm:

```
#include <bibliothek.h>
...
bib_funktion(...);
...
```

- Kompilieren und Linken:

```
gcc [-Ipfad] programm.c [-Lpfad] -lbib -o programm
```

- Unter Linux gibt es statische und dynamische Bibliotheken, die entweder fest mit dem Programm verbunden oder zur Laufzeit zugeschaltet werden. Die Namensgebung folgt dem Muster `libname.a` (statisch) oder `libname.so` (dynamisch).
- Bei der Compiler-Option `-l` werden `lib` und die Endung weggelassen, also `-lname`. Die Umschaltung von dynamisch (Standard) zu statisch erfolgt mit der Option `-static`.
- Der Befehl `ldd programm` zeigt alle vom Programm verwendeten dynamischen Bibliotheken an.

GNU Scientific Library (GSL)

- Die GNU Scientific Library (GSL) ist eine umfangreiche Bibliothek für mathematische und numerische Probleme.
- Zur Verfügung gestellt werden Routinen für:
  - Spezielle Funktionen
  - Vektoren, Matrizen, Lineare Algebra (BLAS), Eigenprobleme (LAPACK)
  - Kombinatorik, Statistik
  - Ableitungen und Integration
  - Differentialgleichungen
  - Reihenentwicklungen (FFT, Chebyshev)
  - ...
- Eine ausführliche Dokumentation findet sich auf `charon` unter: `/usr/share/doc/gsl-doc-pdf/gsl-ref.pdf.gz`

FFTW

- Die "Fastest Fourier Transform in the West" (FFTW) ist eine sehr gute Bibliothek für diskrete Fourier-, Sinus- und Cosinus-Transformationen.
- Neben Zweierpotenzen kann FFTW auch gut mit anderen Datensatzlängen umgehen.
- Source-Code und Dokumentation finden sich auf der Webseite `http://www.fftw.org/`. Außerdem ist die Bibliothek in den meisten Linux-Distributionen enthalten.
- Auf `charon` findet sich die Dokumentation unter `/usr/share/doc/fftw3-doc/html/index.html`

Numerical Recipes

- Das Buch

  W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
  Numerical Recipes in C – The Art of Scientific Computing
  Cambridge University Press, 2nd edition 1992

  bietet dem Naturwissenschaftler einen guten Einstieg in fast alle Gebiete der numerischen Mathematik.
- Der Text des Buchs kann frei aus dem Internet bezogen werden (neuerdings leider nur noch mit speziellem Acrobat plugin), siehe `http://www.nr.com`.
- Die zum Buch gehörige Funktions-Bibliothek und Beispiel-Programme sind beliebt aber kostenpflichtig.
- Die C-Versionen der Codes sind (recht lieblose) Übersetzungen der Fortran-Versionen. Deshalb laufen Indizes oft von 1 bis N statt, wie in C üblich, von 0 bis $N - 1$.
- Manche Formulierungen des Buchs sind etwas plakativ, für zahlreiche Probleme gibt es bessere, freie Bibliotheken (z.B. FFTW)

BLAS

- Die Abkürzung BLAS steht für Basic Linear Algebra Subroutines.
- Dementsprechend umfaßt die Bibliothek Routinen für:
  - Vektoroperationen (Addition, Skalarprodukt)
  - Matrix-Vektor Multiplikation
  - Matrix-Operationen (Addition, Transposition, Produkte)
- Von BLAS gibt es viele (inbesondere freie) Versionen, die für spezielle Hardware und Prozessoren optimiert sind. Oft liefern Computer-Hersteller eigene, auf Höchstleistung getrimmte Versionen (z.B. Intel MKL).
- Moderne Versionen von BLAS nutzen mehrere Prozessoren eines Shared-Memory Computers (z.B. Multi-Core Rechner)
- Über eine einheitliche Schnittstelle können so die häufig auftretenden Grundoperationen mit der jeweils besten Performance ausgeführt werden.
- BLAS ist in Fortran programmiert, kann aber problemlos von C aus benutzt werden. Darüberhinaus gibt es das CBLAS Frontend.

LAPACK

- Aufbauend auf BLAS bietet das Linear Algebra Package (LAPACK) stabile Routinen für:
  - die Lösung von Gleichungssystemen,
  - Eigenwertprobleme,
  - Lineare Optimierungsprobleme (kleinste Quadrate).
- Da LAPACK auf BLAS zurückgreift, sind die Routinen meist gut auf die benutzte Hardware abgestimmt.
- Mit ScaLAPACK existiert eine Erweiterung für Parallel-Rechner mit verteiltem Speicher (distributed memory).
- Abgesehen von einigen speziell optimierten Versionen ist LAPACK freie Software.
- LAPACK ist ebenfalls in Fortran programmiert, aber gut von C aus zu benutzen.

Intel MKL

- Die Intel Math Kernel Library (MKL) ist eine sehr umfangreiche Sammlung mathematischer Routinen, die vom Hersteller speziell für die eigene Hardware optimiert wurde.
- Die Bibliothek umfaßt:
  - BLAS, sparse BLAS
  - LAPACK, ScaLAPACK
  - Mathematische Funktionen auf Datenarrays
  - Statistische Funktionen
  - Fourier Transformation
  - Differentialgleichungs-Löser
  - Routinen für Optimierungsprobleme
- Privatanwender können die Linux-Version kostenlos aus dem Netz beziehen und nutzen. Firmen und Universitäten zahlen moderate Lizenzgebühren.
- Wo Kompatibilität mit den großen freien Bibliotheken besteht, lohnt sich der Einsatz der MKL (BLAS, LAPACK). Bei den anderen Routinen ist Vorsicht geboten (Portabilität / Verfügbarkeit auf anderen Systemen).

GLib

- Der GNOME Toolkit ist eine umfangreiche Software-Bibliothek zur Programmierung graphischer Benutzeroberflächen. In Form der GLib sind viele grundlegende Funktionen zu einer selbständigen Bibliothek ausgegliedert.
- GLib umfaßt vorallem "informatische" Routinen und Datenstrukturen
  - Erweiterungen und Varianten grundlegender Datentypen
  - Strukturen wie Hashes und Listen
  - Funktionen zur Prozessverwaltung / Signal-Verarbeitung
  - Mustererkennung
  - Ein- und Ausgabe-Funktionen
  - ...
- GLib kann als eine Art STL für C aufgefaßt werden (vgl. Standard Template Library für C++).

Beispiel I

```
/*****************************************************************
   Polynom-Multiplikation mittels FFT
   Alex Weisse 2009

   Compilieren:  gcc -Wall polymult.c -lm -lfftw3 -o polymult
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>

#include <fftw3.h>

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

   int np, nq, nr, i;
   double inr;


   /* Einlesen des ersten Polynoms: n c[0] c[1] ... c[n-1] */
   printf(" p(x) = ");
   scanf(" %i", &np);
```

Beispiel II

```
double dp[np];

for(i=0; i<np; i++) scanf(" %lg", &dp[i]);

/* Einlesen des zweiten Polynoms: n c[0] c[1] ... c[n-1] */
printf(" q(x) = ");
scanf(" %i", &nq);

double dq[nq];

for(i=0; i<nq; i++) scanf(" %lg", &dq[i]);

/* Ordnung des Produkts + 1 */
nr = np+nq;
inr = 1.0/((double) nr);

/* Speicher und Plan fuer Hin- und Ruecktrafo*/
fftw_complex *in, *out;
fftw_plan hin, rueck;

in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nr);
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nr);
```

Beispiel III

```
/* Initialisierung der beiden FFTs */
hin = fftw_plan_dft_1d(nr, in, in, FFTW_FORWARD, FFTW_ESTIMATE);
rueck = fftw_plan_dft_1d(nr, out, out, FFTW_BACKWARD, FFTW_ESTIMATE)

/* Trafo des 1. Polynoms */
for(i=0; i<np; i++) in[i] = dp[i];
for(i=np; i<nr; i++) in[i] = 0.0;

fftw_execute(hin);

for(i=0; i<nr; i++) out[i] = in[i];

/* Trafo des 2. Polynoms */
for(i=0; i<nq; i++) in[i] = dq[i];
for(i=nq; i<nr; i++) in[i] = 0.0;

fftw_execute(hin);

for(i=0; i<nr; i++) out[i] *= in[i];

/* Ruecktrafo des Produkts */
fftw_execute(rueck);
```

Beispiel IV

```
  /* Ausgabe der Ergebnisse */
  printf("(%g ", dp[0]);
  for(i=1; i<np; i++) printf(" + %g*x^%i", dp[i], i);
  printf(")*(%g ", dq[0]);
  for(i=1; i<nq; i++) printf(" + %g*x^%i", dq[i], i);
  printf(") = \n");

  printf(" %g ", inr*creal(out[0]));
  for(i=1; i<nr-1; i++) printf(" + %g*x^%i", inr*creal(out[i]), i);
  printf("\n");

  /* Aufraeumen */
  fftw_destroy_plan(hin);
  fftw_destroy_plan(rueck);

  fftw_free(in);
  fftw_free(out);

  return EXIT_SUCCESS;
}
```

Literatur

- Brian W. Kernighan, Dennis M. Ritchie
  The C Programming Language
  Prentice Hall, 1978 und neuer
- Robert Sedgewick
  Algorithms
  Addison-Wesley, 1983 und neuer
- Brian W. Kernighan, Rob Pike
  The practice of programming
  Addison Wesley, 1999
- Donald E. Knuth
  The art of computer programming 1–3
  Addison Wesley, 3rd edition 1997
- W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
  Numerical Recipes in C
  Cambridge University Press, 2nd edition 1992
- Rubin H. Landau, Manuel J. Páez
  Computational Physics
  John Wiley, 1997