

# Prozedurale Programmieretechnik – C

Prof. Dr.-Ing. Torsten Finke

FOM

15. September 2010(Rev.: 1cb6e5e8250c)

# Überblick

Ankündigungen

Literatur

Compiler

Betriebssystem

Editor

C Programmierung (Revision: e2a356af8738 )

# Überblick

Ankündigungen

Literatur

Compiler

Betriebssystem

Editor

C Programmierung (Revision: e2a356af8738 )

# Überblick

Ankündigungen

Literatur

Compiler

Betriebssystem

Editor

C Programmierung (Revision: e2a356af8738 )

# Überblick

Ankündigungen

Literatur

Compiler

Betriebssystem

Editor

C Programmierung (Revision: e2a356af8738 )

# Überblick

Ankündigungen

Literatur

Compiler

Betriebssystem

Editor

C Programmierung (Revision: e2a356af8738 )

# Überblick

Ankündigungen

Literatur

Compiler

Betriebssystem

Editor

C Programmierung (Revision: e2a356af8738 )

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ **Einstieg/Übersicht**
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Inhalte

Script: <http://www.dr-torsten-finke.de/lehre/c>

- ▶ Einstieg/Übersicht
- ▶ Typen, Operatoren, Ausdrücke
- ▶ Kontrollstrukturen
- ▶ Funktionen
- ▶ Zeiger und Felder
- ▶ Strukturen
- ▶ Ein-/Ausgabe
- ▶ Betriebssystem-Schnittstelle

# Klausur

- ▶ Genaue Modalitäten noch in Arbeit
- ▶ Inhalte der Veranstaltung komplett relevant
- ▶ keine Hilfsmittel
- ▶ Auswahlklausur
- ▶ Formvorschriften

# Klausur

- ▶ Genaue Modalitäten noch in Arbeit
- ▶ Inhalte der Veranstaltung komplett relevant
- ▶ keine Hilfsmittel
- ▶ Auswahlklausur
- ▶ Formvorschriften

# Klausur

- ▶ Genaue Modalitäten noch in Arbeit
- ▶ Inhalte der Veranstaltung komplett relevant
- ▶ keine Hilfsmittel
- ▶ Auswahlklausur
- ▶ Formvorschriften

# Klausur

- ▶ Genaue Modalitäten noch in Arbeit
- ▶ Inhalte der Veranstaltung komplett relevant
- ▶ keine Hilfsmittel
- ▶ Auswahlklausur
- ▶ Formvorschriften

# Klausur

- ▶ Genaue Modalitäten noch in Arbeit
- ▶ Inhalte der Veranstaltung komplett relevant
- ▶ keine Hilfsmittel
- ▶ Auswahlklausur
- ▶ Formvorschriften

# Lob und Tadel

- ▶ Kritik, Anmerkungen, Fragen: am besten sofort, konkret, direkt!
- ▶ Evaluation – Sinn und Nutzen

# Lob und Tadel

- ▶ Kritik, Anmerkungen, Fragen: am besten sofort, konkret, direkt!
- ▶ Evaluation – Sinn und Nutzen

# Literatur

- ▶ BRIAN W. KERNIGHAN, DENNIS M. RITCHIE: C Programming Language
- ▶ STEVE OUALLINE: Practical C Programming
- ▶ ROBERT SEDGEWICK Algorithms in C
- ▶ Regionales Rechenzentrum der Uni Hannover: Die Programmiersprache C – Ein Nachschlagewerk  
[www.rrzn.uni-hannover.de/buecher.html](http://www.rrzn.uni-hannover.de/buecher.html))
- ▶ [www.phim.unibe.ch/comp\\_doc/c\\_manual/C/cref.html](http://www.phim.unibe.ch/comp_doc/c_manual/C/cref.html)
- ▶ [www.strath.ac.uk/IT/Docs/Ccourse/](http://www.strath.ac.uk/IT/Docs/Ccourse/)
- ▶ [www.cppreference.com/](http://www.cppreference.com/)
- ▶ [www.cs.cf.ac.uk/Dave/C/CE.html](http://www.cs.cf.ac.uk/Dave/C/CE.html)
- ▶ [www-ccs.ucsd.edu/c/](http://www-ccs.ucsd.edu/c/)
- ▶ [cm.bell-labs.com/cm/cs/who/dmr/cman.pdf](http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf)
- ▶ [math.brown.edu/~jhs/ReferenceCards/CRefCard.v2.2.pdf](http://math.brown.edu/~jhs/ReferenceCards/CRefCard.v2.2.pdf)
- ▶ [www.lysator.liu.se/c/ten-commandments.html](http://www.lysator.liu.se/c/ten-commandments.html)

# C-Compiler

Produkt	Betriebs-system	URL
gcc <sup>1</sup>	div.	<a href="http://www.gnu.org">www.gnu.org</a>
lcc	Windows	<a href="http://www.cs.virginia.edu/~lcc-win32">www.cs.virginia.edu/~lcc-win32</a>
Cygwin	Windows	<a href="http://www.cygwin.com">www.cygwin.com</a>
MINGW32 <sup>1</sup>	Windows	<a href="http://www.mingw.org">www.mingw.org</a>
TinyCC <sup>1</sup>	Linux	<a href="http://fabrice.bellard.free.fr">fabrice.bellard.free.fr</a>
Pacific C	DOS	<a href="http://www.hitech.com.au/products/pacific.html">www.hitech.com.au/products/pacific.html</a>
Dev-C++	Windows	<a href="http://www.bloodshed.net/devcpp.html">www.bloodshed.net/devcpp.html</a>
Borland C++ <sup>2</sup>	Windows	<a href="http://www.borland.com">www.borland.com</a>
Visual C++ <sup>2</sup>	Windows	<a href="http://www.microsoft.com">www.microsoft.com</a>

---

<sup>1</sup>GNU/GPL

<sup>2</sup>kommerziell

# Betriebssystem – Linux

- ▶ Anmeldung über Name und Passwort (wird vom Dozenten bekanntgegeben);
- ▶ Arbeit beenden:
  - ▶ Anwendungen schließen;
  - ▶ Abmelden;
  - ▶ Rechner herunterfahren;
  - ▶ Rechner ausschalten.

# Betriebssystem – Linux – Oberfläche

Arbeiten mit der Shell (Kommandointerpreter):

- ▶ Verzeichnis anlegen/löschen: `mkdir name / rmdir name;`
- ▶ in Verzeichnis wechseln: `cd name;`
- ▶ Dateien listen (ausführlich): `ls (ls -l)`
- ▶ Datei kopieren/umbenennen/löschen: `cp quelle ziel / mv quelle ziel / rm datei;`
- ▶ Datei editieren: `emacs datei;`
- ▶ Datei compilieren:
  - ▶ Programm erstellen (compilieren und linken): `cc -o prog prog.c;`
  - ▶ nur compilieren: `cc -c prog.c`
  - ▶ linken: `cc -o prog prog.o`
- ▶ Programmaufruf im aktuellen Verzeichnis: `./prog`
- ▶ Datenübergabe an ein Programm bar:
  - ▶ Ausgabe des ersten Programms foo als Eingabe: `foo | bar`
  - ▶ Einlesen von einer Datei data: `bar < data`
  - ▶ Ausgabe in eine Datei result: `bar > result`

# Editor – Emacs

- ▶ Aufruf: `emacs datei &`
- ▶ Editor für mehrere Buffer (Speicherkopien von Dateien);
- ▶ Hauptfunktionen (neu, öffnen, schließen, beenden) über Menü;
- ▶ Kommandozeile (unten): Dateinamen, Erweiterungsfunktionen;
- ▶ Zahlreiche Shortcuts (Strg: C, Alt: M, Shift: S, Escape: ESC, Eingabe: RET, Tabulator TAB):
  - ▶ einrücken: TAB;
  - ▶ Suchen: C-s (vorwärts), C-r (rückwärts);
  - ▶ Suchen/ersetzen: M-%
  - ▶ Rückgängig: C-\_
  - ▶ Positionen: C-a (Zeilenanfang), C-e (Zeilenende), M-< (Dateianfang), M-> (Dateiende);
  - ▶ Anweisung abbrechen: C-g
- ▶ Hilfen:
  - ▶ Tutorial, Manual, FAQ;
  - ▶ Apropos: C-h a

# Hello World

- ▶ Standards includieren
- ▶ Main-Prozedur

## Listing 1: hello.c

```
1  #include <stdio.h>
2
3  main()
4  {
5      printf("hello , \nworld\n");
6  }
```

# Fahrenheit – Celsius

- ▶ Typen
- ▶ Schleifen

## Listing 2: fahrrels.c

```
1  #include <stdio.h>
2
3  /* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300 */
4
5  main()
6  {
7      int fahr, celsius;
8      int lower, upper, step;
9      lower = 0;          /* lower limit of temperature scale */
10     upper = 300;        /* upper limit */
11     step = 20;          /* step size */
12     fahr = lower;
13     while (fahr <= upper) {
14         celsius = 5 * (fahr - 32) / 9;
15         printf("%d\t%d\n", fahr, celsius);
16         fahr = fahr + step;
17     }
18 }
```

# Fließpunktrechnung

## Listing 3: fahrfloat.c

```
1  #include <stdio.h>
2
3  /* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300;
4     floating-point version */
5
6  main()
7  {
8     float fahr, celsius;
9     float lower, upper, step;
10    lower = 0;        /* lower limit of temperature scale */
11    upper = 300;      /* upper limit */
12    step = 20;        /* step size */
13    fahr = lower;
14    while (fahr <= upper) {
15        celsius = (5.0/9.0) * (fahr - 32.0);
16        printf("%3.0f_ %6.1f\n", fahr, celsius);
17        fahr = fahr + step;
18    }
19 }
```

# Schleifensteuerung

## Listing 4: fahrfor.c

```
1  #include <stdio.h>
2
3  /* print Fahrenheit-Celsius table */
4
5  main()
6  {
7      int fahr;
8      for (fahr = 0; fahr <= 300; fahr = fahr + 20) {
9          printf("%3d_%.1f\n", fahr, (5.0/9.0)*(fahr - 32));
10     }
11 }
```

# Konstantendefinition

## Listing 5: fahrconst.c

```
1  #include <stdio.h>
2
3  #define LOWER 0 /* lower limit of table */
4  #define UPPER 300 /* upper limit */
5  #define STEP 20 /* step size */
6
7  /* print Fahrenheit-Celsius table */
8
9  main()
10
11 {
12     int fahr;
13     for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP) {
14         printf("%3d_%.1f\n", fahr, (5.0/9.0)*(fahr - 32));
15     }
16 }
```

# Übungen

- ▶ Erweitern Sie das Programm Fahrenheit/Celsius-Umrechnung um die Ausgabe einer Überschrift.
- ▶ Schreiben Sie ein Programm zur Konvertierung von Celsius in Fahrenheit.
- ▶ Lassen Sie die Tabelle in absteigender Folge ausgeben.
- ▶ Bestimmen Sie den größten und kleinsten Wert einer Ganzzahlvariablen.
- ▶ Bestimmen Sie den betragskleinsten Wert für eine Floatvariable.
- ▶ Bestimmen Sie die Auflösung des Typs `float`.

# Dateien kopieren

- ▶ Zeichen ein- und ausgeben
- ▶ was ist ein Zeichen?
- ▶ Dateiende
- ▶ logischer Ausdruck

## Listing 6: filecopy.c

```
1  #include <stdio.h>
2
3  /* copy input to output */
4
5  main ()
6  {
7      int c;
8      c = getchar ();
9      while (c != EOF) {
10         putchar(c);
11         c = getchar ();
12     }
13 }
```

# Dateien einfacher kopieren

## Listing 7: fcopy.c

```
1  #include <stdio.h>
2
3  /* copy input to output;
4     no buffer variable version */
5
6  main ()
7
8  {
9     int c;
10    while ((c = getchar ()) != EOF) {
11        putchar(c);
12    }
13 }
```

# Übungen

- ▶ Bestimmen Sie den Wert des Ausdrucks `getchar() != EOF`.
- ▶ Bestimmen Sie den Wert von `EOF`.
- ▶ Bestimmen Sie die numerischen Werte der Zeichen '0', '9', 'A', 'B', 'z' und '+'.  
▶ Bestimmen Sie die numerischen Werte der Symbole *Zeilenumbruch*, *Wagenrücklauf*, *Tabulator* und *Leerzeichen*.

# Zeichen zählen

## Listing 8: ccount.c

```
1  #include <stdio.h>
2
3  /* count characters in input;
4   integer version */
5  main()
6  {
7      long nc;
8      nc = 0;
9      while (getchar() != EOF) {
10         ++nc;
11     }
12     printf("%ld\n", nc);
13 }
```

# Viele Zeichen zählen

## Listing 9: ccntflt.c

```
1  #include <stdio.h>
2
3  /* count characters in input; float version */
4  main()
5  {
6      double nc;
7      for (nc = 0; getchar() != EOF; ++nc)
8          ;
9      printf("%.0f\n", nc);
10 }
```

# Zeilen zählen

## Listing 10: lcount.c

```
1  #include <stdio.h>
2
3  /* count lines in input */
4  main()
5  {
6      int c, nl;
7      nl = 0;
8      while ((c = getchar()) != EOF) {
9          if (c == '\n') {
10             ++ nl;
11         }
12     }
13     printf("%d\n", nl);
14 }
```

# Übungen

- ▶ Schreiben Sie ein Programm, das Leerzeichen, Tabulatoren und Zeilenenden zählt.
- ▶ Schreiben Sie ein Programm, das die Eingabedaten ausgibt, dabei alle Folgen von mehrfachen Leerzeichen durch ein Leerzeichen ersetzt.
- ▶ Schreiben Sie ein Programm, das Tabulatoren von der Eingabe als `\t` in der Ausgabe sichtbar macht.
- ▶ Geben Sie eine ASCII-Tabelle aus. Sie soll Spalten für den dezimalen, den hexadezimalen und den Zeichenwert des ASCII-Symbols besitzen. Versehen Sie die Tabelle mit einem sinnvollen Kopf. Beachten Sie, dass die ersten 32 Zeichen nicht darstellbar sind.
- ▶ Ordnen Sie die ASCII-Tabelle der obigen Aufgabe so, dass Sie sie in mehrere Abschnitte gliedern, die nebeneinander ausgegeben werden.
- ▶ Ermitteln Sie die größte Zahl (zum Beispiel Zeichen im Eingabestrom), die in einer `double`-Variablen einzeln zählbar sind.

# Wörter zählen – Fallunterscheidung if-then-else

## Listing 11: wcount.c

```
1  #include <stdio.h>
2  #define IN 1 /* inside a word */
3  #define OUT 0 /* outside a word */
4  main() /* count lines, words, and characters in input */
5  {
6      int c, nl, nw, nc, state;
7      state = OUT;
8      nl = nw = nc = 0;
9      while ((c = getchar()) != EOF) {
10         ++nc; /* count characters */
11         if (c == '\n')
12             ++nl; /* count lines */
13         if (c == '_' || c == '\n' || c == '\t') {
14             state = OUT;
15         } else if (state == OUT) {
16             state = IN;
17             ++nw; /* count words */
18         }
19     }
20     printf("%d_%d_%d\n", nl, nw, nc);
21 }
```

# Übungen

- ▶ Arbeitet das Programm zum Zählen von Zeichen, Wörtern und Zeilen immer korrekt? Wie lässt sich dies prüfen?
- ▶ Schreiben Sie ein Programm, das jedes Wort der Eingabe in einer neuen Zeile ausgibt.

# Distinkte Zeichen zählen – Felder (Arrays)

Listing 12: dcount.c

```
1  #include <stdio.h>
2
3  /* count digits */
4  main()
5  {
6      int c, i;
7      int ndigit[10];
8
9      for (i = 0; i < 10; ++i) {
10         ndigit[i] = 0;
11     }
12     while ((c = getchar()) != EOF) {
13         if (c >= '0' && c <= '9') {
14             ++ ndigit[c-'0'];
15         }
16     }
17     printf("digits _=");
18     for (i = 0; i < 10; ++i) {
19         printf("_%d", ndigit[i]);
20     }
21 }
```

# Übungen

- ▶ Schreiben Sie Programme, die Histogramme für folgende Informationen über die Eingabe ausgeben (Histogramme können durch horizontale Linien dargestellt werden):
  - ▶ Wortlänge;
  - ▶ Zeilenlänge;
  - ▶ Zeichenhäufigkeit.

# Funktionen – Potenz

## Listing 13: power.c

```
1  #include <stdio.h>
2  int power(int m, int n);
3
4  main()
5  {
6      int i;
7      for (i = 0; i < 10; ++i) {
8          printf("%d_%d_%d\n", i, power(2, i), power(-3, i));
9      }
10 }
11
12 /* power: raise base to n-th power; n >= 0 */
13 int power(int base, int n)
14 {
15     int i, p;
16     for (p = i = 1; i <= n; ++i) {
17         p = p * base;
18     }
19     return p;
20 }
```

# Übungen

Die flexible Eingabe kann in C-Programmen mit der (eigentlich sehr komplexen) Funktion `scanf()` erfolgen. Ein Beispieldialog zur Eingabe eines Wertes in die Floatvariable `x` wäre etwa folgender:

```
printf("Geben Sie einen Wert fuer x ein: ");  
scanf("%e", &x);
```

Anschließend befindet sich in der Variablen der eingegebene Wert.

- ▶ Schreiben Sie eine Funktion `kugel`, die zu einem gegebenen Radius das Volumen der Kugel ermittelt.
- ▶ Schreiben Sie das Programm zur Temperaturkonvertierung unter Verwendung einer Funktion um.
- ▶ Schreiben Sie ein Programm, das nach den Koeffizienten einer quadratischen Gleichung fragt und alle Lösungen ausgibt. Behandeln Sie alle denkbaren Kombinationen an Eingaben und Lösungen.
- ▶ Schreiben Sie ein Programm, mit dem festgestellt werden kann, ob Variable als Wert oder als Referenz übergeben werden.

# Längste Zeile ermitteln – Zeichenketten (Strings)

- ▶ Includes
- ▶ Defines
- ▶ Prototypes

## Listing 14: longline.c

```
1 #include <stdio.h>
2 #define MAXLINE 1000 /* maximum input line length */
3
4 int getline(char line[], int maxline);
5 void copy(char to[], char from[]);
6
7 /* print the longest input line */
8 main()
9 {
```

## ... Längste Zeile ermitteln – Zeichenketten (Strings)

### ► Main

#### Listing 15: longline.c

```
1  main()
2  {
3      int len;                /* current line length */
4      int max = 0;           /* maximum length seen so far */
5      char line[MAXLINE];    /* current input line */
6      char longest[MAXLINE]; /* longest line saved here */
7
8      while ((len = getline(line, MAXLINE)) > 0) {
9          if (len > max) {
10             max = len;
11             copy(longest, line);
12         }
13     }
14     if (max > 0) { /* there was a line */
15         printf("%s", longest);
16     }
17     return 0;
18 }
```

## ... Längste Zeile ermitteln – Zeichenketten (Strings)

### ► Function – Zeile lesen

Listing 16: longline.c

```
1  /* getline: read a line into s, return length */
2  int getline(char s[], int lim)
3  {
4      int c, i;
5      for (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i) {
6          s[i] = c;
7      }
8      if (c == '\n') {
9          s[i++] = c;
10     }
11     s[i] = '\0';
12     return i;
13 }
```

## ... Längste Zeile ermitteln – Zeichenketten (Strings)

### ► Function – kopieren

Listing 17: longline.c

```
1  /* copy: copy 'from' into 'to'; assume 'to' is big enough */
2  void copy(char to[], char from[])
3  {
4      int i = 0;
5      while ((to[i] = from[i]) != '\0') {
6          ++i;
7      }
8  }
```

# Übungen

- ▶ Schreiben Sie ein Programm, das alle Zeilen ausgibt, die länger als 30 Zeichen sind.
- ▶ Schreiben Sie ein Programm, das alle Zeilen ausgibt, deren drittes Zeichen ein 'a' oder 'A' ist.
- ▶ Schreiben Sie ein Programm, das Tabulatoren und Leerzeichen am Zeilenende löscht und leere Zeilen unterdrückt.
- ▶ Schreiben Sie eine Funktion `reverse(s)`, das den String `s` umkehrt. Schreiben Sie ein Programm, das diese Funktion nutzt, um Eingabezeilen umzukehren.

# Globale Variable

## Listing 18: lineglobal.c

```
1  #include <stdio.h>
2  #define MAXLINE 1000  /* maximum input line size */
3
4  int max;              /* maximum length seen so far */
5  char line[MAXLINE];  /* current input line */
6  char longest[MAXLINE]; /* longest line saved here */
7  int getline(void);
8
9  void copy(void);
10
11 /* print longest input line; global variable version */
12 main()
13 {
```

## ... Globale Variable

## Listing 19: lineglobal.c

```
1  /* print longest input line; global variable version */
2  main()
3  {
4      int len;
5      extern int max;
6      extern char longest[];
7      max = 0;
8      while ((len = getline()) > 0) {
9          if (len > max) {
10             max = len;
11             copy();
12         }
13     }
14     if (max > 0) { /* there was a line */
15         printf("%s", longest);
16     }
17     return 0;
18 }
```

## ... Globale Variable

## Listing 20: lineglobal.c

```
1  /* getline: specialized version */
2  int  getline(void)
3  {
4      int  c, i;
5      extern char line[];
6      for (i = 0; i < MAXLINE - 1 && (c=getchar()) != EOF && c != '\n';
7          line[i] = c;
8      )
9      if (c == '\n') {
10         line[i] = c;
11         ++ i;
12     }
13     line[i] = '\0';
14     return i;
15 }
```

## ... Globale Variable

Listing 21: lineglobal.c

```
1  /* copy: specialized version */
2  void copy(void)
3  {
4      int i;
5      extern char line [], longest [];
6      i = 0;
7      while ((longest[i] = line[i]) != '\0')
8          ++i;
9  }
```

# Übungen

- ▶ Schreiben Sie ein Programm, das Tabulatoren im Eingabestrom durch passende Leerzeichen ersetzt. Nehmen Sie eine geeignete Tabulatorweite an.
- ▶ Schreiben Sie ein Programm, das Folgen von Leerzeichen geeignet durch Tabulatoren ersetzt.
- ▶ Schreiben Sie ein Programm, das lange Zeilen geeignet (also möglichst an Wortgrenzen) umbricht.
- ▶ Schreiben Sie ein Programm, das Kommentare aus C-Programmen entfernt. (Kommentare werden nicht geschachtelt).

# Typen

- ▶ Basistypen:
  - ▶ char;
  - ▶ int;
  - ▶ float;
  - ▶ double;
- ▶ Qualifier:
  - ▶ short/long;
  - ▶ signed/unsigned;
  - ▶ const.
- ▶ Die Größe in Bytes eines Typs oder einer Variable kann mit dem Operator `sizeof` ermittelt werden, z.B.: `sizeof ( int )`;
- ▶ Schreiben Sie ein Programm, mit dem man die Größe der Typen und ihrer Varianten ermitteln kann, ohne den Operator `sizeof` zu verwenden.

# Typen

- ▶ Basistypen:
  - ▶ char;
  - ▶ int;
  - ▶ float;
  - ▶ double;
- ▶ Qualifier:
  - ▶ short/long;
  - ▶ signed/unsigned;
  - ▶ const.
- ▶ Die Größe in Bytes eines Typs oder einer Variable kann mit dem Operator `sizeof` ermittelt werden, z.B.: `sizeof ( int );`
- ▶ Schreiben Sie ein Programm, mit dem man die Größe der Typen und ihrer Varianten ermitteln kann, ohne den Operator `sizeof` zu verwenden.

# Typen

- ▶ Basistypen:
  - ▶ char;
  - ▶ int;
  - ▶ float;
  - ▶ double;
- ▶ Qualifier:
  - ▶ short/long;
  - ▶ signed/unsigned;
  - ▶ const.
- ▶ Die Größe in Bytes eines Typs oder einer Variable kann mit dem Operator `sizeof` ermittelt werden, z.B.: `sizeof ( int );`
- ▶ Schreiben Sie ein Programm, mit dem man die Größe der Typen und ihrer Varianten ermitteln kann, ohne den Operator `sizeof` zu verwenden.

# Typen

- ▶ Basistypen:
  - ▶ char;
  - ▶ int;
  - ▶ float;
  - ▶ double;
- ▶ Qualifier:
  - ▶ short/long;
  - ▶ signed/unsigned;
  - ▶ const.
- ▶ Die Größe in Bytes eines Typs oder einer Variable kann mit dem Operator `sizeof` ermittelt werden, z.B.: `sizeof ( int );`
- ▶ Schreiben Sie ein Programm, mit dem man die Größe der Typen und ihrer Varianten ermitteln kann, ohne den Operator `sizeof` zu verwenden.

# Numerische Konstante

- ▶ Ganzzahlkonstante: z.B. 1234;
- ▶ lange Ganzzahlkonstante mit Terminal *L*, *l*: z.B. 123456789L;
- ▶ Float-Konstante: Terminal *F*, *f*, Dezimalpunkt und/oder Exponent;
- ▶ Vorzeichenlose Konstante mit Terminal *U*, *u*;
- ▶ Hexadezimale Konstante mit führendem 0x;
- ▶ Oktale Konstante mit führender 0 (Null);

# Numerische Konstante

- ▶ Ganzzahlkonstante: z.B. 1234;
- ▶ lange Ganzzahlkonstante mit Terminal *L*, *l*: z.B. 123456789L;
- ▶ Float-Konstante: Terminal *F*, *f*, Dezimalpunkt und/oder Exponent;
- ▶ Vorzeichenlose Konstante mit Terminal *U*, *u*;
- ▶ Hexadezimale Konstante mit führendem 0x;
- ▶ Oktale Konstante mit führender 0 (Null);

# Zeichen- und String-Konstante

- ▶ Zeichenkonstante in Apostroph, z.B.: 'x', benannt z.B.: '\n' oder nummeriert z.B.: '\ooo' (oktal) oder '\xhh' (hexadezimal);
- ▶ String Konstante in Anführungszeichen, z.B.: "hello" (enthält folgende '\0').
- ▶ Verkettete String-Konstante getrennt durch Zwischenraum, z.B.: "hello " "world"

# Zeichen- und String-Konstante

- ▶ Zeichenkonstante in Apostroph, z.B.: 'x', benannt z.B.: '\n' oder nummeriert z.B.: '\ooo' (oktal) oder '\xhh' (hexadezimal);
- ▶ String Konstante in Anführungszeichen, z.B.: "hello" (enthält folgende '\0').
- ▶ Verkettete String-Konstante getrennt durch Zwischenraum, z.B.: "hello \_" "world"

# Aufzählungs-Konstante

- ▶ `enum boolean { NO, YES }; (0, 1);`
- ▶ `enum escapes { LF = '\n', CR = '\r', HT = '\t'};;`
- ▶ `enum months { JAN = 1, FEB, MAR, APR }; (1, 2, 3, 4);`
- ▶ Beispiel:

```
1 enum months m;  
2 ...  
3 if ( m == FEB ) {  
4     ndays = 29;  
5 }
```

# Aufzählungs-Konstante

- ▶ `enum boolean { NO, YES }; (0, 1);`
- ▶ `enum escapes { LF = '\n', CR = '\r', HT = '\t'};;`
- ▶ `enum months { JAN = 1, FEB, MAR, APR }; (1, 2, 3, 4);`
- ▶ Beispiel:

```
1 enum months m;  
2 ...  
3 if ( m == FEB ) {  
4     ndays = 29;  
5 }
```

# Arithmetische Operatoren

- ▶ Summe, Differenz (+ und -);
- ▶ Produkt und Quotient (\* und /);
- ▶ Modulo (Rest bei Teilen, nur für Integer-Typen):  $x \% y$
- ▶ Übung: Schreiben Sie ein Programm, das für eine gegebene Ganz-Zahl (z.B. eine Kontonummer mit den Ziffern  $n_1 n_2 n_3 \dots n_k$ ) eine Prüfziffer  $z$  nach folgendem Verfahren berechnet:

$$z = (1n_1 + 2n_2 \dots + kn_k) \text{ mod } 10$$

- ▶ Ist die Prüfziffer der vorherigen Aufgabe gleich, wenn nach *jeder* arithmetischen Operation eine Modulo-Operation erfolgt?

# Arithmetische Operatoren

- ▶ Summe, Differenz (+ und -);
- ▶ Produkt und Quotient (\* und /);
- ▶ Modulo (Rest bei Teilen, nur für Integer-Typen):  $x \% y$
- ▶ Übung: Schreiben Sie ein Programm, das für eine gegebene Ganz-Zahl (z.B. eine Kontonummer mit den Ziffern  $n_1 n_2 n_3 \dots n_k$ ) eine Prüfziffer  $z$  nach folgendem Verfahren berechnet:

$$z = (1n_1 + 2n_2 \dots + kn_k) \text{ mod } 10$$

- ▶ Ist die Prüfziffer der vorherigen Aufgabe gleich, wenn nach *jeder* arithmetischen Operation eine Modulo-Operation erfolgt?

# Arithmetische Operatoren

- ▶ Summe, Differenz (+ und -);
- ▶ Produkt und Quotient (\* und /);
- ▶ Modulo (Rest bei Teilen, nur für Integer-Typen):  $x \% y$
- ▶ Übung: Schreiben Sie ein Programm, das für eine gegebene Ganz-Zahl (z.B. eine Kontonummer mit den Ziffern  $n_1 n_2 n_3 \dots n_k$ ) eine Prüfziffer  $z$  nach folgendem Verfahren berechnet:

$$z = (1n_1 + 2n_2 \dots + kn_k) \text{ mod } 10$$

- ▶ Ist die Prüfziffer der vorherigen Aufgabe gleich, wenn nach *jeder* arithmetischen Operation eine Modulo-Operation erfolgt?

# Relationale und logische Operatoren

- ▶ Arithmetisch: `<` `<=` `==` `!=` `>=` `>`
- ▶ Logisch: `&&` `||`
- ▶ Logische Negation: `!`

# Typ-Konversion

- ▶ Binäre Operatoren implizieren Konversion in:
  - ▶ long double;
  - ▶ double;
  - ▶ float;
  - ▶ int (ggf. long).
- ▶ Casting: explizite Typkonversion, z.B. ( float ) n.
- ▶ Schreiben Sie eine Funktion `htoi(s)`, die den String `s`, der hexadezimale Ziffern und optional führendes `0x` enthält, in eine Ganzzahl konvertiert.

# Typ-Konversion

- ▶ Binäre Operatoren implizieren Konversion in:
  - ▶ long double;
  - ▶ double;
  - ▶ float;
  - ▶ int (ggf. long).
- ▶ Casting: explizite Typkonversion, z.B. ( float ) n.
- ▶ Schreiben Sie eine Funktion `htoi(s)`, die den String `s`, der hexadezimale Ziffern und optional führendes `0x` enthält, in eine Ganzzahl konvertiert.

# Typ-Konversion

- ▶ Binäre Operatoren implizieren Konversion in:
  - ▶ long double;
  - ▶ double;
  - ▶ float;
  - ▶ int (ggf. long).
- ▶ Casting: explizite Typkonversion, z.B. ( float ) n.
- ▶ Schreiben Sie eine Funktion `htoi(s)`, die den String `s`, der hexadezimale Ziffern und optional führendes `0x` enthält, in eine Ganzzahl konvertiert.

# Inkrement und Dekrement

- ▶ Inkrement `++` und Dekrement `--`
- ▶ Präinkrement und Postinkrement: wenn `n = 5`, was liefern `x = n++;` und `x = ++n;`?
- ▶ Vorsicht bei logischen Operationen: es muss nicht jeder Teilausdruck ausgewertet werden – problematisch bei Nebeneffekten:  
( `x++ > 3 && y -- < 6` );

# Inkrement und Dekrement

- ▶ Inkrement `++` und Dekrement `--`
- ▶ Präinkrement und Postinkrement: wenn `n = 5`,  
was liefern `x = n++;` und `x = ++n;`?
- ▶ Vorsicht bei logischen Operationen: es muss nicht jeder Teilausdruck ausgewertet werden – problematisch bei Nebeneffekten:  
( `x++ > 3 && y -- < 6` );

# Inkrement und Dekrement – Beispiel

- ▶ Ein trickreiches Beispiel:

Listing 22: `deblank.c`

```
1  /* deblank: loeschen von Leerzeichen aus String */
2  void deblank(char s[])
3  {
4      int i, j;
5      for (i = j = 0; s[i] != '\0'; i++) {
6          if (s[i] != ' ') {
7              s[j++] = s[i];
8          }
9      }
10     s[j] = '\0';
11 }
```

- ▶ Schreiben Sie eine Funktion `squeeze(s1, s2)`, die alle Zeichen in `s2` aus dem String `s1` entfernt.

# Inkrement und Dekrement – Beispiel

- ▶ Ein trickreiches Beispiel:

Listing 23: `deblank.c`

```
1  /* deblank: loeschen von Leerzeichen aus String */
2  void deblank(char s[])
3  {
4      int i, j;
5      for (i = j = 0; s[i] != '\0'; i++) {
6          if (s[i] != ' ') {
7              s[j++] = s[i];
8          }
9      }
10     s[j] = '\0';
11 }
```

- ▶ Schreiben Sie eine Funktion `squeeze(s1, s2)`, die alle Zeichen in `s2` aus dem String `s1` entfernt.

# Bitweise Operatoren

- ▶ Bitweise UND, ODER sowie EXKLUSIV ODER:  $&$  |  $\wedge$
- ▶ Links/rechts verschieben (multiplizieren/dividieren mit 2)  $\ll$   $\gg$
- ▶ Bitweise negieren (Einerkomplement):  $\sim$
- ▶ Beispiel:

a =	0	1	0	1	
b =	0	0	1	1	
a&b =	0	0	0	1	maskieren
a b =	0	1	1	1	überblenden
a^b =	0	1	1	0	reversibel
~a =	1	0	1	0	
a>>2 =	0	0	0	1	
a<<1 =	1	0	1	0	

# Bitweise Operatoren

- ▶ Bitweise UND, ODER sowie EXKLUSIV ODER:  $&$   $|$   $\wedge$
- ▶ Links/rechts verschieben (multiplizieren/dividieren mit 2)  $\ll$   $\gg$
- ▶ Bitweise negieren (Einerkomplement):  $\sim$
- ▶ Beispiel:

$a =$	0	1	0	1	
$b =$	0	0	1	1	
$a \& b =$	0	0	0	1	maskieren
$a   b =$	0	1	1	1	überblenden
$a \wedge b =$	0	1	1	0	reversibel
$\sim a =$	1	0	1	0	
$a \gg 2 =$	0	0	0	1	
$a \ll 1 =$	1	0	1	0	

# Bitweise Operatoren – Beispiel

Listing 24: `getbits.c`

```

1  /* getbits: get n bits from position p */
2  /*
3  /*           p=5                               p=0
4  /*           |                                   |
5  /*  | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
6  /*           +----- n=3 -----+
7  /*
8  /*
9  /*   getbits(10110101b, 5, 3) = 110b
10 /*
11 unsigned getbits(unsigned x, int p, int n){
12     return (x >> (p+1-n)) & ~(~0 << n);
13 }

```

# Bitweise Operatoren – Übungen

Schreiben Sie Funktionen:

- ▶ `setbits(x, y, a, b)`, welche die Bits von Position `a` bis `b` aus `x` nach `y` kopieren, die anderen aber unverändert lassen.
- ▶ `rotate(x, n)`, welche die Bits in `x` um `n` Stellen nach rechts rotiert.

# Bitweise Operatoren – Übungen

Schreiben Sie Funktionen:

- ▶ `setbits(x, y, a, b)`, welche die Bits von Position `a` bis `b` aus `x` nach `y` kopieren, die anderen aber unverändert lassen.
- ▶ `rotate(x, n)`, welche die Bits in `x` um `n` Stellen nach rechts rotiert.

# Zuweisungsoperatoren

- ▶  $x = y + z$
- ▶  $i = i + 2$  ist bedeutungsgleich zu  $i += 2$
- ▶ Kombinationsoperatoren:  $+ - * / \% \ll \gg \& \wedge |$
- ▶ Beispiel:

Listing 25: bitcount.c

```
1  /* bitcount: count 1 bits in x */
2  int bitcount(unsigned x)
3  {
4      int b;
5      for (b = 0; x != 0; x >>= 1) {
6          if (x & 01) {
7              b++;
8          }
9      }
10     return b;
11 }
```

# Konditionalausdrücke

## ► Der Ausdruck

```
1  if (a > b)
2      x = a;
3  else
4      x = b;
```

kann vereinfacht werden durch:

```
x = (a > b) ? a : b;
```

## ► Der Konditionalausdruck

- ist ein Ausdruck;
- ist der einzige ternäre Ausdruck in C.

## ► Beispiel:

```
1  #define max(x, y) (x > y) ? x : y
2  ...
3  z = max(x, y);
```

- Übung: Schreiben Sie eine Funktion `lower(s)`, die alle Buchstaben im String `s` in Kleinbuchstaben umsetzt.

# Konditionalausdrücke

## ► Der Ausdruck

```
1  if (a > b)
2      x = a;
3  else
4      x = b;
```

kann vereinfacht werden durch:

$x = (a > b) ? a : b;$

## ► Der Konditionalausdruck

- ist ein Ausdruck;
- ist der einzige ternäre Ausdruck in C.

## ► Beispiel:

```
1  #define max(x, y) (x > y) ? x : y
2  ...
3  z = max(x, y);
```

- Übung: Schreiben Sie eine Funktion `lower(s)`, die alle Buchstaben im String `s` in Kleinbuchstaben umsetzt.

# Konditionalausdrücke

## ► Der Ausdruck

```
1  if (a > b)
2      x = a;
3  else
4      x = b;
```

kann vereinfacht werden durch:

$x = (a > b) ? a : b;$

## ► Der Konditionalausdruck

- ist ein Ausdruck;
- ist der einzige ternäre Ausdruck in C.

## ► Beispiel:

```
1  #define max(x, y) (x > y) ? x : y
2  ...
3  z = max(x, y);
```

- Übung: Schreiben Sie eine Funktion `lower(s)`, die alle Buchstaben im String `s` in Kleinbuchstaben umsetzt.

# Konditionalausdrücke

## ► Der Ausdruck

```
1  if (a > b)
2      x = a;
3  else
4      x = b;
```

kann vereinfacht werden durch:

```
x = (a > b) ? a : b;
```

## ► Der Konditionalausdruck

- ist ein Ausdruck;
- ist der einzige ternäre Ausdruck in C.

## ► Beispiel:

```
1  #define max(x, y) (x > y) ? x : y
2  ...
3  z = max(x, y);
```

- Übung: Schreiben Sie eine Funktion `lower(s)`, die alle Buchstaben im String `s` in Kleinbuchstaben umsetzt.

# Vorrang und Assoziativität

Vorrang	Operatoren	Assoziativität
hoch	( ) [] -> .	links
	! ~ ++ -- + - * (type) sizeof	rechts
	* / %	links
	+ -	links
	<< >>	links
	< <= > >=	links
	== !=	links
	&	links
	^	links
		links
	&&	links
		links
	?:	rechts
	= += -= *= /= %= &= ^=  = <<= >>=	rechts
niedrig	,	links

Unäre & + - \* haben höheren Vorrang als die binären Formen.

# Sequenz – Statements und Blöcke

## ▶ Einfach-Statements:

- ▶ Deklarationen;
- ▶ Anweisungen;
- ▶ Abschluss per Semikolon;
- ▶ Sequenzoperator Komma.

## ▶ Blocks:

- ▶ Zusammenfassung beliebig vieler Statements zu einem formalen Statement;
- ▶ Kodierung durch geschweifte Klammern:

```
1 Block := { Statement ; Statement ; ... }
```

- ▶ Hinter der schließenden geschweiften Klammer folgt kein Semikolon.

# Sequenz – Statements und Blöcke

- ▶ Einfach-Statements:
  - ▶ Deklarationen;
  - ▶ Anweisungen;
  - ▶ Abschluss per Semikolon;
  - ▶ Sequenzoperator Komma.
- ▶ Blocks:
  - ▶ Zusammenfassung beliebig vieler Statements zu einem formalen Statement;
  - ▶ Kodierung durch geschweifte Klammern:
    - 1 `Block := { Statement ; Statement ; ... }`
  - ▶ Hinter der schließenden geschweiften Klammer folgt kein Semikolon.

# Alternation – If-Then-Else

## ▶ Allgemeine Form:

```
1  if( expression )
2      statement
3  else if( expression )
4      statement
5  else if( expression )
6      statement
7  else if( expression )
8      statement
9  else
10     statement
```

- ▶ Die Zweige `else` und `else if` können entfallen.
- ▶ Unterschied zwischen `expression`, `expression = 0` und `expression == 0`;
- ▶ Empfehlung: Statements grundsätzlich in Blocks schreiben.

# If-Then-Else, Beispiel

## Listing 26: binsearch.c

```
1  /* binsearch:  suche x im sortierten Feld
2  v[0] <= v[1] <= ... <= v[n-1] */
3  int binsearch(int x, int v[], int n)
4  {
5      int low, high, mid;
6      low = 0;
7      high = n - 1;
8      while (low <= high) {
9          mid = (low + high) / 2;
10         if (x < v[mid]) {
11             high = mid - 1;
12         } else if (x > v[mid]) {
13             low = mid + 1;
14         } else { /* gefunden */
15             return mid;
16         }
17     }
18     return -1; /* no match */
19 }
```

# Alternation – Switch

## ▶ Allgemeine Form:

```
1  switch( expression )
2      case const-expr:
3          statements
4      case const-expr:
5          statements
6      default:
7          statements
```

## ▶ Man beachte:

- ▶ Getestet wird gegen konstante Ausdrücke (die also schon zum Zeitpunkt der Compilation festliegen);
- ▶ Die Fälle werden inklusiv und nicht exklusiv behandelt; sonst ist ein `break`-Statement zu verwenden.

# Alternation – Switch

## ▶ Allgemeine Form:

```
1  switch( expression )
2      case const-expr:
3          statements
4      case const-expr:
5          statements
6      default:
7          statements
```

## ▶ Man beachte:

- ▶ Getestet wird gegen konstante Ausdrücke (die also schon zum Zeitpunkt der Compilation festliegen);
- ▶ Die Fälle werden inklusiv und nicht exklusiv behandelt; sonst ist ein `break`-Statement zu verwenden.

# Switch, Beispiel

```
1  ...
2  switch (c) {
3      case '0': case '1': case '2': case '3': case '4':
4      case '5': case '6': case '7': case '8': case '9':
5          ndigit[c-'0']++;
6          break;
7      case '_':
8      case '\n':
9      case '\t':
10         nwhite++;
11         break;
12     default:
13         nother++;
14         break;
15 }
16 ...
```

# Iteration – abweisende Schleifen: while und for

**while:** Endlosschleife:

```
1     while ( expression )
2         statement
```

**for:** initialisierte Endlosschleife mit optionalem Index:

```
1     for ( init-statement; expression; loop-statement )
2         statement
```

ist äquivalent zu

```
1     init-statement;
2     while ( expression ) {
3         statements;
4         loop-statement
5     }
```

# Iteration – abweisende Schleifen: while und for

**while:** Endlosschleife:

```
1     while ( expression )
2         statement
```

**for:** initialisierte Endlosschleife mit optionalem Index:

```
1     for ( init-statement; expression; loop-statement )
2         statement
```

ist äquivalent zu

```
1     init-statement;
2     while ( expression ) {
3         statements;
4         loop-statement
5     }
```

# Abweisende Schleifen – Übung

Schreiben Sie eine Funktion `expand{s1, s2}`, die eine Kurzbeschreibung der Form “a-z“ in `s1` in den String “abc...xyz“ in `s2` entwickelt.

# Iteration – nicht abweisende Schleifen

```
do
    statement
while ( expression );
```

Beispiel:

## Listing 27: itoa.c

```
1  /* itoa: Zahl n zeichenweise in String s ausgeben */
2  void itoa(int n, char s[])    {
3      int i = 0, sign;
4      if ((sign = n) < 0) { /* Vorzeichen merken */
5          n = -n;          /* n positiv machen */
6      }
7      do { /* Ziffern in umgekehrter Reihenfolge erstellen */
8          s[i++] = n % 10 + '0'; /* naechste Ziffer */
9      } while ((n /= 10) > 0); /* Ziffer abtrennen */
10     if (sign < 0) {
11         s[i++] = '-';
12     }
13     s[i] = '\0';
14     reverse(s);
15 }
```

# Kontrollstrukturen – nicht abweisende Schleifen – Übung

Schreiben Sie eine Funktion `itob(n, s, b)`, die eine Zahl `n` in eine Ziffernfolge zur Basis `b` in den String `s` schreibt.

# Kontrollstrukturen – Break und Continue

**break:** Abbruch der innersten Ebene einer Kontrollstruktur;

Beispiel

```
1  /* trim: remove trailing blanks, tabs, newlines */
2  int trim(char s[]) {
3      int n;
4      for (n = strlen(s)-1; n >= 0; n--) {
5          if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
6              break;
7      }
8      s[n+1] = '\0';
9      return n;
10 }
```

**continue:** Außerplanmäßiges Überspringen des Schleifenrestes;

Beispiel

```
1  for (i = 0; i < n; i++) {
2      if (a[i] < 0) /* skip negative elements */
3          continue;
4      ... /* do positive elements */
5  }
```

# Kontrollstrukturen – Goto und Labels

## ▶ Beispiel:

```
1  for ( ... ) {  
2      for ( ... ) {  
3          ...  
4              if ( disaster ) goto error;  
5          }  
6      }  
7  ...  
8  error:          /* Fehlerbehandlung */
```

- ▶ Sprung aus tief verschachtelten Kontrollstrukturen;
- ▶ Fehlerbehandlung;
- ▶ Goto immer vermeidbar;

# Programmstruktur – verteilte Quelltexte

- ▶ Header-Files:
  - ▶ gemeinsame Deklarationen, z.B. `#define LINELENGTH 1000`
  - ▶ Prototypen, z.B. `double power(double basis, double exponent)`
  - ▶ globale Variable
  - ▶ einbinden der Headerfiles mit `#include "file .h"`
- ▶ Sichtbarkeit von Variablen und Funktionen:
  - ▶ begrenzt auf die lokale Funktion: `static`
  - ▶ Bezug auf Namen in anderen Files, z.B. `extern power()`

# Programmstruktur – verteilte Quelltexte

- ▶ Header-Files:
  - ▶ gemeinsame Deklarationen, z.B. `#define LINELENGTH 1000`
  - ▶ Prototypen, z.B. `double power(double basis, double exponent)`
  - ▶ globale Variable
  - ▶ einbinden der Headerfiles mit `#include "file .h"`
- ▶ Sichtbarkeit von Variablen und Funktionen:
  - ▶ begrenzt auf die lokale Funktion: `static`
  - ▶ Bezug auf Namen in anderen Files, z.B. `extern power()`

# Funktionen – Details

- ▶ Rückgabewerte
  - ▶ standardmäßig `int`
  - ▶ können ignoriert werden.
- ▶ Compilersyntaxcheck möglich aber nicht erforderlich
  - ▶ implizite Integerdeklaration ohne Parameterprüfung: `foo()`
  - ▶ explizite Deklaration (keine Parameter): `int bar(void)`
- ▶ Initialisierung
  - ▶ automatische Variable werden jeweils im Block neu generiert und ggf. initialisiert;
  - ▶ statische Variable werden nur beim ersten Erscheinen initialisiert, aktuelle Werte bleiben erhalten.

# Funktionen – Details

- ▶ Rückgabewerte
  - ▶ standardmäßig `int`
  - ▶ können ignoriert werden.
- ▶ Compilersyntaxcheck möglich aber nicht erforderlich
  - ▶ implizite Integerdeklaration ohne Parameterprüfung: `foo()`
  - ▶ explizite Deklaration (keine Parameter): `int bar(void)`
- ▶ Initialisierung
  - ▶ automatische Variable werden jeweils im Block neu generiert und ggf. initialisiert;
  - ▶ statische Variable werden nur beim ersten Erscheinen initialisiert, aktuelle Werte bleiben erhalten.

# Funktionen – Details

- ▶ Rückgabewerte
  - ▶ standardmäßig `int`
  - ▶ können ignoriert werden.
- ▶ Compilersyntaxcheck möglich aber nicht erforderlich
  - ▶ implizite Integerdeklaration ohne Parameterprüfung: `foo()`
  - ▶ explizite Deklaration (keine Parameter): `int bar(void)`
- ▶ Initialisierung
  - ▶ automatische Variable werden jeweils im Block neu generiert und ggf. initialisiert;
  - ▶ statische Variable werden nur beim ersten Erscheinen initialisiert, aktuelle Werte bleiben erhalten.

# Funktionen – Rekursion

## ► Fakultät

```
1 int fact(int n) {
2     if (n <= 1) return 1;
3     return n * fact(n-1);
4 }
```

## ► Türme von Hanoi

```
1 void shift(int n, int a, int b, int c) {
2     /* verschiebe n Scheiben von Saeule a nach Saeule b */
3     /* nimm Saeule c zu Hilfe */
4     if (n<=1) {
5         printf ("%d nach %d\n", a, b);
6     } else {
7         shift (n-1, a, c, b);
8         shift (1, a, b, c);
9         shift (n-1, c, b, a);
10    }
11 }
```

# Funktionen – Rekursion

## ► Fakultät

```
1 int fact(int n) {
2     if (n <= 1) return 1;
3     return n * fact(n-1);
4 }
```

## ► Türme von Hanoi

```
1 void shift(int n, int a, int b, int c) {
2     /* verschiebe n Scheiben von Saeule a nach Saeule b */
3     /* nimm Saeule c zu Hilfe */
4     if (n<=1) {
5         printf ("%d nach %d\n", a, b);
6     } else {
7         shift (n-1, a, c, b);
8         shift (1, a, b, c);
9         shift (n-1, c, b, a);
10    }
11 }
```

# Funktionen – Rekursion – Übungen

- ▶ Schreiben Sie eine Funktion `printd(int n)`, welche das Argument als Dezimalzahl ausgibt.
- ▶ Schreiben Sie eine Funktion `perm(int a [], n)`, welche die n Zahlen im Array a permutiert, also alle möglichen Vertauschungen ermittelt, und jeweils ausgibt.

# Präprozessor

## ▶ Einbinden von Quelltexten

- ▶ aus den Systemverzeichnissen: `#include <stdio.h>`
- ▶ aus dem lokalen Verzeichnis: `#include "file .h"`

## ▶ Makros

- ▶ lexikalische Ersetzung für Token;
- ▶ Definition von Konstanten, z.B. `#define PI 3.14159`
- ▶ formale Parameter, z.B. `#define max(a, b) (a>b)?a:b`
- ▶ man beachte:
  - ▶ Klammerung, z.B. Problem mit `#define square(a) a * a`  
Was liefert Aufruf von `square(x+1)`?
  - ▶ Nebeneffekte, z.B. doppeltes Inkrement bei `square(i++)`  
um wieviel wird `i` wirklich erhöht?
- ▶ Konkatenation, z.B. `#define paste(name, ext) name ## ext -`  
Aufruf `paste(brief, kopf)` liefert das Token `briefkopf`
- ▶ Makros in Strings:  
`#define out(var) printf("#var " _="%d\n", var)`

# Präprozessor

- ▶ Einbinden von Quelltexten
  - ▶ aus den Systemverzeichnissen: `#include <stdio.h>`
  - ▶ aus dem lokalen Verzeichnis: `#include "file .h"`
- ▶ Makros
  - ▶ lexikalische Ersetzung für Token;
  - ▶ Definition von Konstanten, z.B. `#define PI 3.14159`
  - ▶ formale Parameter, z.B. `#define max(a, b) (a>b)?a:b`
  - ▶ man beachte:
    - ▶ Klammerung, z.B. Problem mit `#define square(a) a * a`  
Was liefert Aufruf von `square(x+1)`?
    - ▶ Nebeneffekte, z.B. doppeltes Inkrement bei `square(i++)`  
um wieviel wird `i` wirklich erhöht?
  - ▶ Konkatenation, z.B. `#define paste(name, ext) name ## ext` –  
Aufruf `paste(brief, kopf)` liefert das Token `briefkopf`
  - ▶ Makros in Strings:  
`#define out(var) printf(#var " \u0322=%d\n", var)`

# Präprozessor

- ▶ Einbinden von Quelltexten
  - ▶ aus den Systemverzeichnissen: `#include <stdio.h>`
  - ▶ aus dem lokalen Verzeichnis: `#include "file .h"`
- ▶ Makros
  - ▶ lexikalische Ersetzung für Token;
  - ▶ Definition von Konstanten, z.B. `#define PI 3.14159`
  - ▶ formale Parameter, z.B. `#define max(a, b) (a>b)?a:b`
  - ▶ man beachte:
    - ▶ Klammerung, z.B. Problem mit `#define square(a) a * a`  
Was liefert Aufruf von `square(x+1)`?
    - ▶ Nebeneffekte, z.B. doppeltes Inkrement bei `square(i++)`  
um wieviel wird `i` wirklich erhöht?
  - ▶ Konkatenation, z.B. `#define paste(name, ext) name ## ext` –  
Aufruf `paste(brief, kopf)` liefert das Token `briefkopf`
  - ▶ Makros in Strings:  
`#define out(var) printf(#var " \u0322=%d\n", var)`

# Präprozessor – bedingte Kompilation

## ▶ allgemeine Fallunterscheidung

```
1 #if SYSTEM == SYSV
2     #define HDR "sysv.h"
3 #elif SYSTEM == MSDOS
4     #define HDR "msdos.h"
5 #else
6     #define HDR "default.h"
7 #endif
8 #include HDR
```

## ▶ Test auf bereits erfolgte Definition

```
1 #ifndef HDR
2     #define HDR
3     /* contents of hdr.h go here */
4 #endif
```

## ▶ komplexere Makros

```
1 #define swap(x, y) do { (x)^(y); (y)^(x); (x)^(y); } while(0)
```

# Präprozessor – bedingte Kompilation

## ▶ allgemeine Fallunterscheidung

```
1 #if SYSTEM == SYSV
2     #define HDR "sysv.h"
3 #elif SYSTEM == MSDOS
4     #define HDR "msdos.h"
5 #else
6     #define HDR "default.h"
7 #endif
8 #include HDR
```

## ▶ Test auf bereits erfolgte Definition

```
1 #ifndef HDR
2 #define HDR
3 /* contents of hdr.h go here */
4 #endif
```

## ▶ komplexere Makros

```
1 #define swap(x, y) do { (x)^(y); (y)^(x); (x)^(y); } while(0)
```

# Adressen und Zeiger

- ▶ Variable haben Adressen, diese können in Zeigern abgespeichert werden.
- ▶ Adressoperator: &
- ▶ Dereferenzierungsoperator: \*
- ▶ Übung

```
1 int x = 1, y = 2, z[] = { 5, 6, 7};
2 int *ip, *iq;
3
4 ip = &x; /* was ist der Inhalt von ip? */
5 y = ip; /* was ist der Inhalt von y? */
6 y = *ip; /* was ist der Inhalt von y? */
7 *ip = 0; /* welche Variable ist nun Null */
8 ip = &z[1]; /* was ist der Inhalt von ip? */
9 *ip = *ip + 10; /* welche Variable wird erhoehrt? */
10 *ip += 3; /* welche Variable wird erhoehrt? */
11 ++*ip; /* welche Variable wird erhoehrt? */
12 (*ip)++; /* welche Variable wird erhoehrt? */
13
14 iq = ip;
15 ++ *iq; /* welche Variable wird erhoehrt? */
```

# Adressen und Zeiger

- ▶ Variable haben Adressen, diese können in Zeigern abgespeichert werden.
- ▶ Adressoperator: &
- ▶ Dereferenzierungsoperator: \*
- ▶ Übung

```
1  int x = 1, y = 2, z[] = { 5, 6, 7};
2  int *ip, *iq;
3
4  ip = &x; /* was ist der Inhalt von ip? */
5  y = ip; /* was ist der Inhalt von y? */
6  y = *ip; /* was ist der Inhalt von y? */
7  *ip = 0; /* welche Variable ist nun Null */
8  ip = &z[1]; /* was ist der Inhalt von ip? */
9  *ip = *ip + 10; /* welche Variable wird erhoeht? */
10 *ip += 3; /* welche Variable wird erhoeht? */
11 ++*ip; /* welche Variable wird erhoeht? */
12 (*ip)++; /* welche Variable wird erhoeht? */
13
14 iq = ip;
15 ++ *iq; /* welche Variable wird erhoeht? */
```

# Zeiger und Funktionsargumente

- ▶ Funktion `swap(a, b)` zum Vertauschen der Inhalte der Variablen;
- ▶ Warum scheitert eine Funktion `void swap(int a, int b)`
- ▶ Übung: Schreiben Sie die Funktion `swap()`. Testen Sie sie mit einfachen Variablen und mit Feldelementen.

# Zeiger und Funktionsargumente

- ▶ Funktion `swap(a, b)` zum Vertauschen der Inhalte der Variablen;
- ▶ Warum scheitert eine Funktion `void swap(int a, int b)`
- ▶ Übung: Schreiben Sie die Funktion `swap()`. Testen Sie sie mit einfachen Variablen und mit Feldelementen.

# Zeiger und Felder

## ► Deklaration:

```
1 int a[] = {0, 0, 0, 0, 0};
2 int *pa;
```

## ► Was bewirken die folgenden Zeilen?

```
1 pa = &a[0];
2 pa = a;
3 *(pa+1) = 5; /* was ist pa+1 ? */
4 *(pa++) += 5; /* Unterschied zu (*pa)++ ? */
```

## ► Übergabe von Feldern an Funktionen erfolgt als Übergabe der Zeiger (call-by-reference).

```
1 /* strlen: return length of string s */
2 int strlen(char *s) {
3     int n;
4     for (n = 0; *s != '\0', s++)
5         n++;
6     return n;
7 }
```

# Zeiger und Felder

## ► Deklaration:

```
1 int a[] = {0, 0, 0, 0, 0};
2 int *pa;
```

## ► Was bewirken die folgenden Zeilen?

```
1 pa = &a[0];
2 pa = a;
3 *(pa+1) = 5; /* was ist pa+1 ? */
4 *(pa++) += 5; /* Unterschied zu (*pa)++ ? */
```

## ► Übergabe von Feldern an Funktionen erfolgt als Übergabe der Zeiger (call-by-reference).

```
1 /* strlen: return length of string s */
2 int strlen(char *s) {
3     int n;
4     for (n = 0; *s != '\0', s++)
5         n++;
6     return n;
7 }
```

# Zeiger und Felder

## ► Deklaration:

```
1 int a[] = {0, 0, 0, 0, 0};
2 int *pa;
```

## ► Was bewirken die folgenden Zeilen?

```
1 pa = &a[0];
2 pa = a;
3 *(pa+1) = 5; /* was ist pa+1 ? */
4 *(pa++) += 5; /* Unterschied zu (*pa)++ ? */
```

## ► Übergabe von Feldern an Funktionen erfolgt als Übergabe der Zeiger (call-by-reference).

```
1 /* strlen: return length of string s */
2 int strlen(char *s) {
3     int n;
4     for (n = 0; *s != '\0', s++)
5         n++;
6     return n;
7 }
```

# Zeiger und Felder – Übungen

Funktionieren die folgenden Aufrufe?

```
1 char a[] = "hi_foix";  
2 char *pa;  
3 pa = a;  
4 strlen("hello_world");  
5 strlen(a);  
6 strlen(a[3]);  
7 strlen(pa);
```

# Felder von Zeigern, Zeiger auf Zeiger

- ▶ Zeiger können in Feldern abgelegt werden:

```
1 #define MAXLINES 100
2 int *lines[MAXLINES];
```

- ▶ Initialisierung von Zeigerfeldern

```
1 char *wday[] = {
2     "Montag", "Dienstag", "Mittwoch", "Donnerstag",
3     "Freitag", "Samstag", "Sonntag" };
4 }
```

- ▶ Übung: Schreiben Sie eine Funktion zum Vertauschen von zwei Feldern. Testen Sie das Programm.

# Felder von Zeigern, Zeiger auf Zeiger

- ▶ Zeiger können in Feldern abgelegt werden:

```
1 #define MAXLINES 100
2 int *lines[MAXLINES];
```

- ▶ Initialisierung von Zeigerfeldern

```
1 char *wday[] = {
2     "Montag", "Dienstag", "Mittwoch", "Donnerstag",
3     "Freitag", "Samstag", "Sonntag" };
4 }
```

- ▶ Übung: Schreiben Sie eine Funktion zum Vertauschen von zwei Feldern. Testen Sie das Programm.

# Felder von Zeigern, Zeiger auf Zeiger

- ▶ Zeiger können in Feldern abgelegt werden:

```
1 #define MAXLINES 100
2 int *lines[MAXLINES];
```

- ▶ Initialisierung von Zeigerfeldern

```
1 char *wday[] = {
2     "Montag", "Dienstag", "Mittwoch", "Donnerstag",
3     "Freitag", "Samstag", "Sonntag" };
4 }
```

- ▶ Übung: Schreiben Sie eine Funktion zum Vertauschen von zwei Feldern. Testen Sie das Programm.

# Mehrdimensionale Felder

## ▶ Beispiel:

```

1  static char daytab[2][13] = {
2      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
3      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
4  };
5
6  /* day_of_year: set day of year from month & day */
7  int day_of_year(int year, int month, int day) {
8      int i, leap;
9      leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
10     for (i = 1; i < month; i++)
11         day += daytab[leap][i];
12     return day;
13 }
```

- ▶ Aufruf per `array[i][j]`, nicht per `array[i, j]`!
- ▶ Bei Funktionsdeklaration muss die Array-Struktur angegeben werden:

```

1  f(int daytab[2][13]) { ... }
2  f(int daytab[][13]) { ... } /* hoechste Dimension kann entfallen
```

Die Struktur muss zur Compile-Zeit feststehen.

# Mehrdimensionale Felder

## ▶ Beispiel:

```

1  static char daytab[2][13] = {
2      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
3      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
4  };
5
6  /* day_of_year: set day of year from month & day */
7  int day_of_year(int year, int month, int day) {
8      int i, leap;
9      leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
10     for (i = 1; i < month; i++)
11         day += daytab[leap][i];
12     return day;
13 }
```

## ▶ Aufruf per `array[i][j]`, nicht per `array[i, j]`!

## ▶ Bei Funktionsdeklaration muss die Array-Struktur angegeben werden:

```

1  f(int daytab[2][13]) { ... }
2  f(int daytab[][13]) { ... } /* hoechste Dimension kann entfallen
```

Die Struktur muss zur Compile-Zeit feststehen.

# Mehrdimensionale Felder

## ▶ Beispiel:

```

1  static char daytab[2][13] = {
2      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
3      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
4  };
5
6  /* day_of_year: set day of year from month & day */
7  int day_of_year(int year, int month, int day) {
8      int i, leap;
9      leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
10     for (i = 1; i < month; i++)
11         day += daytab[leap][i];
12     return day;
13 }
```

- ▶ Aufruf per `array[i][j]`, nicht per `array[i, j]`!
- ▶ Bei Funktionsdeklaration muss die Array-Struktur angegeben werden:

```

1  f(int daytab[2][13]) { ... }
2  f(int daytab[][13]) { ... } /* hoechste Dimension kann entfallen
```

Die Struktur muss zur Compile-Zeit feststehen.

# Kommandozeilenargumente

- ▶ Übergabe von Anzahl und Werten der Kommandozeile an `main()`

```
1 #include <stdio.h>
2
3 /* echo command-line arguments */
4 int main(int argc, char *argv[])
5 {
6     int i;
7     for (i = 1; i < argc; i++)
8         printf("%4d: %s\n", i, argv[i]);
9     return 0;
10 }
```

- ▶ Rückgabe eines Erfolgscodes
- ▶ Übung: Schreiben Sie ein Programm, das als `tail -n` aufgerufen die letzten `n` Zeilen des Eingabestroms liefert.

# Kommandozeilenargumente

- ▶ Übergabe von Anzahl und Werten der Kommandozeile an `main()`

```
1▶ #include <stdio.h>
2
3 /* echo command-line arguments */
4 int main(int argc, char *argv[])
5 {
6     int i;
7     for (i = 1; i < argc; i++)
8         printf("%4d: %s\n", i, argv[i]);
9     return 0;
10 }
```

- ▶ Rückgabe eines Erfolgscode
- ▶ Übung: Schreiben Sie ein Programm, das als `tail -n` aufgerufen die letzten `n` Zeilen des Eingabestroms liefert.

# Kommandozeilenargumente

- ▶ Übergabe von Anzahl und Werten der Kommandozeile an `main()`

```
1 ▶ #include <stdio.h>
2
3 /* echo command-line arguments */
4 int main(int argc, char *argv[])
5 {
6     int i;
7     for (i = 1; i < argc; i++)
8         printf("%4d: %s\n", i, argv[i]);
9     return 0;
10 }
```

- ▶ Rückgabe eines Erfolgscodes
- ▶ Übung: Schreiben Sie ein Programm, das als `tail -n` aufgerufen die letzten `n` Zeilen des Eingabestroms liefert.

# Kommandozeilenargumente

- ▶ Übergabe von Anzahl und Werten der Kommandozeile an `main()`

```
1 ▶ #include <stdio.h>
2
3 /* echo command-line arguments */
4 int main(int argc, char *argv[])
5 {
6     int i;
7     for (i = 1; i < argc; i++)
8         printf("%4d: %s\n", i, argv[i]);
9     return 0;
10 }
```

- ▶ Rückgabe eines Erfolgscodes
- ▶ Übung: Schreiben Sie ein Programm, das als `tail -n` aufgerufen die letzten `n` Zeilen des Eingabestroms liefert.

# Zeiger auf Funktionen

## Listing 28: newton.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double sqeps; /* Wurzel Maschinengenauigkeit */
5
6  double f(double);
7  double df(double);
8  double newton (double x, double f(double), double df(double));
9
10 int main(void)
11 {
12     double x0 = 0.0, eps = 1.0;
13
14     while ( 1 + (eps /= 2.0) > 1);
15     sqeps = sqrt(eps);
16
17     x0 = newton(x0, f, df); /* Nullstelle nach Newton */
18
19     printf("x0 = %g\n", x0);
20     return 0;
21 }
```

## ... Zeiger auf Funktionen

## Listing 29: newton.c

```
1  /* Nullstelle einer Funktion f(x) mit ihrer Ableitung df(x) */
2  double newton (double x, double f(double), double df(double))
3  {
4      double dx;
5      do {
6          x -= dx = f(x)/df(x);
7      } while (fabs(dx/x) > sqeps);
8      return x;
9  }
10
11 double f(double x)
12 {
13     return ((x - 3) * x - 5);
14 }
15
16 double df(double x)
17 {
18     return (2*x - 3);
19 }
```

# Strukturen – Deklaration

- ▶ Kombination zusammengehörender Daten, z.B. die Koordinaten eines Punktes:

```
1 struct point {  
2     int x;  
3     int y;  
4 };
```

- ▶ Instanzen:

- ▶ kombiniert mit Strukturdeklaration:

```
1 struct point {  
2     int x;  
3     int y;  
4 } lu, ro;
```

- ▶ unter Bezug auf erfolgte Deklaration:

```
1 struct point pt;
```

- ▶ Zugriff auf Strukturelemente: *struktur.element*

```
1 a = pt.x;
```

# Strukturen – Deklaration

- ▶ Kombination zusammengehörender Daten, z.B. die Koordinaten eines Punktes:

```
1 struct point {  
2     int x;  
3     int y;  
4 };
```

- ▶ Instanzen:

- ▶ kombiniert mit Strukturdeklaration:

```
1 struct point {  
2     int x;  
3     int y;  
4 } lu, ro;
```

- ▶ unter Bezug auf erfolgte Deklaration:

```
1 struct point pt;
```

- ▶ Zugriff auf Strukturelemente: *struktur.element*

```
1 a = pt.x;
```

# Strukturen – Deklaration

- ▶ Kombination zusammengehörender Daten, z.B. die Koordinaten eines Punktes:

```
1 struct point {  
2     int x;  
3     int y;  
4 };
```

- ▶ Instanzen:

- ▶ kombiniert mit Strukturdeklaration:

```
1 struct point {  
2     int x;  
3     int y;  
4 } lu, ro;
```

- ▶ unter Bezug auf erfolgte Deklaration:

```
1 struct point pt;
```

- ▶ Zugriff auf Strukturelemente: *struktur.element*

```
1 a = pt.x;
```

# Strukturen – Deklaration

- ▶ Kombination zusammengehörender Daten, z.B. die Koordinaten eines Punktes:

```
1 struct point {  
2     int x;  
3     int y;  
4 };
```

- ▶ Instanzen:

- ▶ kombiniert mit Strukturdeklaration:

```
1 struct point {  
2     int x;  
3     int y;  
4 } lu, ro;
```

- ▶ unter Bezug auf erfolgte Deklaration:

```
1 struct point pt;
```

- ▶ Zugriff auf Strukturelemente: *struktur.element*

```
1 a = pt.x;
```

# Geschachtelte Strukturen

```
1 ▶ struct {  
2     struct point pt1;  
3     struct point pt2;  
4 } viereck;  
5 ...  
6 a = viereck.pt1.x;
```

## ▶ Übungen:

- ▶ Schreiben Sie eine Sequenz, die die Fläche des Rechtecks `viereck` bestimmt.
- ▶ Schreiben Sie eine Sequenz, die die Länge eines Vektors vom Koordinatenursprung bis zum Punkt `pt` berechnet. (Hinweis: nutzen Sie die Funktion `double sqrt(double x)`, Prototyp in `math.h`; alternativ entwickeln Sie eine eigene Wurzelfunktion.)

# Geschachtelte Strukturen

```
1 ▶ struct {  
2     struct point pt1;  
3     struct point pt2;  
4 } viereck;  
5 ...  
6 a = viereck.pt1.x;
```

## ▶ Übungen:

- ▶ Schreiben Sie eine Sequenz, die die Fläche des Rechtecks `viereck` bestimmt.
- ▶ Schreiben Sie eine Sequenz, die die Länge eines Vektors vom Koordinatenursprung bis zum Punkt `pt` berechnet. (Hinweis: nutzen Sie die Funktion `double sqrt(double x)`, Prototyp in `math.h`; alternativ entwickeln Sie eine eigene Wurzelfunktion.)

# Strukturen und Funktionen

- ▶ Strukturen werden wie Einfachvariable als Wert übergeben, nicht als Referenz.

- ▶ Beispiel:

```
1  /* diffpoints: Abstandsvektor */
2  struct point diffpoints(struct point p1, struct point p2)
3  {
4      p1.x -= p2.x;
5      p1.y -= p2.y;
6      return p1;
7  }
```

- ▶ Übergabe als Referenz (Bindung beachten):

```
1  struct point punkt, *pp;
2  pp = &kpunkt;
3  ...
4  printf("P_=_(%d,%d)\n", (*pp).x, (*pp).y);
5  /* oder */
6  printf("P_=_(%d,%d)\n", pp->x, pp->y);
```

# Strukturen und Funktionen

- ▶ Strukturen werden wie Einfachvariable als Wert übergeben, nicht als Referenz.

- ▶ Beispiel:

```
1  /* diffpoints: Abstandsvektor */
2  struct point diffpoints(struct point p1, struct point p2)
3  {
4      p1.x -= p2.x;
5      p1.y -= p2.y;
6      return p1;
7  }
```

- ▶ Übergabe als Referenz (Bindung beachten):

```
1  struct point punkt, *pp;
2  pp = &kpunkt;
3  ...
4  printf("P_=_(%d,%d)\n", (*pp).x, (*pp).y);
5  /* oder */
6  printf("P_=_(%d,%d)\n", pp->x, pp->y);
```

# Strukturen – Übung

▶ gegeben

```
1 struct {  
2     int len;  
3     int val;  
4 } s = {7, 13}, *p = &s;
```

Was bewirken oder bedeuten:

- ▶  $++p \rightarrow len$
- ▶  $(++p) \rightarrow len$
- ▶  $(p++) \rightarrow len$

# Felder von Strukturen

- ▶ Beispiel (Struktur zum Zählen von C-Schlüsselwörtern):

```
1 struct key {
2     char *word;
3     int count;
4 } keytab [] = {
5     "break", 0,
6     "case", 0,
7     "char", 0,
8     /* ... */
9     "unsigned", 0,
10    "while", 0
11 };
```

- ▶ Bestimmen der Anzahl der Einträge:

```
1 #define NKEYS (sizeof keytab / sizeof(struct key))
```

# Felder von Strukturen

- ▶ Beispiel (Struktur zum Zählen von C-Schlüsselwörtern):

```
1 struct key {
2     char *word;
3     int count;
4 } keytab [] = {
5     "break", 0,
6     "case", 0,
7     "char", 0,
8     /* ... */
9     "unsigned", 0,
10    "while", 0
11 };
```

- ▶ Bestimmen der Anzahl der Einträge:

```
1 #define NKEYS (sizeof keytab / sizeof(struct key))
```

# Strukturen und Zeiger

## ► Listen (Beispiel)

```
1 struct nlist {           /* table entry: */
2     struct nlist *next; /* next entry in chain */
3     int val;             /* value */
4 } a, b, c, *p;
```

## ► Strukturen mit Zeigern auf abhängige Elemente (Beispiel):

```
1 struct tnode {           /* the tree node: */
2     char *word;           /* points to the text */
3     int count;           /* number of occurrences */
4     struct tnode *left;  /* left child */
5     struct tnode *right; /* right child */
6 };
```

# Strukturen und Zeiger

## ► Listen (Beispiel)

```
1 struct nlist {           /* table entry: */
2     struct nlist *next; /* next entry in chain */
3     int val;             /* value */
4 } a, b, c, *p;
```

## ► Strukturen mit Zeigern auf abhängige Elemente (Beispiel):

```
1 struct tnode {          /* the tree node: */
2     char *word;         /* points to the text */
3     int count;         /* number of occurrences */
4     struct tnode *left; /* left child */
5     struct tnode *right; /* right child */
6 };
```

# Unions

- ▶ Formal ähnlich zu Struktur.
- ▶ Speichert dieselbe Information in unterschiedlichen Typen.
- ▶ Beispiel:

```
1 union number {  
2     int ival;  
3     float fval;  
4     char s[sizeof(int)];  
5 } num;
```

- ▶ Endianess
- ▶ Übungen:
  - ▶ Wie wird der Wert `0x41424344` auf verschiedenen Architekturen (z.B. Intel/PowerPC) intern gespeichert?
  - ▶ Wie wird intern eine Fließpunktzahl dargestellt?

# Unions

- ▶ Formal ähnlich zu Struktur.
- ▶ Speichert dieselbe Information in unterschiedlichen Typen.
- ▶ Beispiel:

```
1 union number {  
2     int ival;  
3     float fval;  
4     char s[sizeof(int)];  
5 } num;
```

- ▶ Endianess
- ▶ Übungen:

- ▶ Wie wird der Wert `0x41424344` auf verschiedenen Architekturen (z.B. Intel/PowerPC) intern gespeichert?
- ▶ Wie wird intern eine Fließpunktzahl dargestellt?

# Unions

- ▶ Formal ähnlich zu Struktur.
- ▶ Speichert dieselbe Information in unterschiedlichen Typen.
- ▶ Beispiel:

```
1 union number {  
2     int ival;  
3     float fval;  
4     char s[sizeof(int)];  
5 } num;
```

- ▶ Endianess
- ▶ Übungen:
  - ▶ Wie wird der Wert `0x41424344` auf verschiedenen Architekturen (z.B. Intel/PowerPC) intern gespeichert?
  - ▶ Wie wird intern eine Fließpunktzahl dargestellt?

# Typdefinitionen

- ▶ Neue Typen aus Einfachtypen deklarieren:
  - ▶ `typedef int time_t` führt einen neuen Typ `time_t` ein, der de facto identisch ist zu `int`.
  - ▶ Komplexe Typen können ebenfalls deklariert werden:

```
1  typedef struct tnode {           /* the tree node: */
2      char *word;                 /* points to the text */
3      int count;                  /* number of occurrences */
4      struct tnode *left;        /* left child */
5      struct tnode *right;       /* right child */
6  } Treenode;
7
8  Treenode n; /* Variable n ist vom Typ der Struktur tnode */
```

# Typdefinitionen

- ▶ Neue Typen aus Einfachtypen deklarieren:
  - ▶ `typedef int time_t` führt einen neuen Typ `time_t` ein, der de facto identisch ist zu `int`.
  - ▶ Komplexe Typen können ebenfalls deklariert werden:

```
1  typedef struct tnode {           /* the tree node: */
2      char *word;                 /* points to the text */
3      int count;                  /* number of occurrences */
4      struct tnode *left;        /* left child */
5      struct tnode *right;       /* right child */
6  } Treenode;
7
8  Treenode n; /* Variable n ist vom Typ der Struktur tnode */
```

# I/O – Standard-ein/ausgabe

- ▶ Zeichenweise: `getchar()` und `putchar()`
- ▶ Zeilenweise: `char *gets(char *s)` und `char *puts(char *s)` (keine Prüfung auf Überlauf!)
- ▶ Formatierte Ein/Ausgabe: `scanf()` und `printf()`:
  - ▶ Formatstring mit Formatbeschreibungen (`%-b.n`) für Argumente

Zeichen	Beschreibung
d, i	Ganzzahl, dezimal
o	Ganzzahl, oktal
x	Ganzzahl, hexadezimal
u	Ganzzahl, vorzeichenlos dezimal
c	Zeichen
s	Zeichenkette
f	Fließpunktzahl
e	Fließpunktzahl, doppelt genau
g	Fließpunktzahl, doppelt genau, flexibles Format
%	Prozentzeichen

# I/O – Standard-ein/ausgabe

- ▶ Zeichenweise: `getchar()` und `putchar()`
- ▶ Zeilenweise: `char *gets(char *s)` und `char *puts(char *s)` (keine Prüfung auf Überlauf!)
- ▶ Formatierte Ein/Ausgabe: `scanf()` und `printf()`:
  - ▶ Formatstring mit Formatbeschreibungen (`%-b.n`) für Argumente

Zeichen	Beschreibung
d, i	Ganzzahl, dezimal
o	Ganzzahl, oktal
x	Ganzzahl, hexadezimal
u	Ganzzahl, vorzeichenlos dezimal
c	Zeichen
s	Zeichenkette
f	Fließpunktzahl
e	Fließpunktzahl, doppelt genau
g	Fließpunktzahl, doppelt genau, flexibles Format
%	Prozentzeichen

# I/O – Standard-ein/ausgabe

- ▶ Zeichenweise: `getchar()` und `putchar()`
- ▶ Zeilenweise: `char *gets(char *s)` und `char *puts(char *s)` (keine Prüfung auf Überlauf!)
- ▶ Formatierte Ein/Ausgabe: `scanf()` und `printf()`:
  - ▶ Formatstring mit Formatbeschreibungen (`%-b.n`) für Argumente

Zeichen	Beschreibung
d, i	Ganzzahl, dezimal
o	Ganzzahl, oktal
x	Ganzzahl, hexadezimal
u	Ganzzahl, vorzeichenlos dezimal
c	Zeichen
s	Zeichenkette
f	Fließpunktzahl
e	Fließpunktzahl, doppelt genau
g	Fließpunktzahl, doppelt genau, flexibles Format
%	Prozentzeichen

# I/O – Standard-ein/ausgabe

- ▶ Zeichenweise: `getchar()` und `putchar()`
- ▶ Zeilenweise: `char *gets(char *s)` und `char *puts(char *s)` (keine Prüfung auf Überlauf!)
- ▶ Formatierte Ein/Ausgabe: `scanf()` und `printf()`:
  - ▶ Formatstring mit Formatbeschreibungen (`%-b.n`) für Argumente

Zeichen	Beschreibung
d, i	Ganzzahl, dezimal
o	Ganzzahl, oktal
x	Ganzzahl, hexadezimal
u	Ganzzahl, vorzeichenlos dezimal
c	Zeichen
s	Zeichenkette
f	Fließpunktzahl
e	Fließpunktzahl, doppelt genau
g	Fließpunktzahl, doppelt genau, flexibles Format
%	Prozentzeichen

## ... I/O – Standard-ein/ausgabe

- ▶ `scanf()` und `printf()` akzeptieren variable Parameterlisten
- ▶ `scanf()` benötigt Zeiger als Parameter
- ▶ Formatierungsfeatures auch auf Strings anwendbar: `sscanf()` und `sprintf()`

## ... I/O – Standard-ein/ausgabe

- ▶ `scanf()` und `printf()` akzeptieren variable Parameterlisten
- ▶ `scanf()` benötigt Zeiger als Parameter
- ▶ Formatierungsfeatures auch auf Strings anwendbar: `sscanf()` und `sprintf()`

## ... I/O – Standard-ein/ausgabe

- ▶ `scanf()` und `printf()` akzeptieren variable Parameterlisten
- ▶ `scanf()` benötigt Zeiger als Parameter
- ▶ Formatierungsfeatures auch auf Strings anwendbar: `sscanf()` und `sprintf()`

# I/O – Dateizugriff

- ▶ Zugriff auf Dateien per Name und Handle vom Type FILE

- ▶ Öffnen der Datei

```
1 FILE *fp ;  
2 fp = fopen("datei.name", "w") /* maybe NULL – do not follow! */
```

Modus:

- ▶ Lesen: "r"
- ▶ Schreiben: "w"
- ▶ Anfügen: "a"
- ▶ optional Unterscheidung zwischen Text und Binärdatei: "rb"
- ▶ Zugriff:
  - ▶ Lesen: `fgetc(fp)`, `fgets(line, size, fp)`, `fscanf()`
  - ▶ Schreiben: `fputc(fp)`, `fputs(line, size, fp)`, `fprintf()`
- ▶ Schließen: `fclose(fp)`
- ▶ Positionieren: `fseek(fp, offset, vonwo)`
- ▶ Status: `stat(name, stat)` liefert Informationen über Datei (keine ANSI-C-Funktion!)

# I/O – Dateizugriff

- ▶ Zugriff auf Dateien per Name und Handle vom Type FILE
- ▶ Öffnen der Datei

```
1 FILE *fp;  
2 fp = fopen("datei.name", "w") /* maybe NULL – do not follow! */
```

## Modus:

- ▶ Lesen: "r"
- ▶ Schreiben: "w"
- ▶ Anfügen: "a"
- ▶ optional Unterscheidung zwischen Text und Binärdatei: "rb"
- ▶ Zugriff:
  - ▶ Lesen: `fgetc(fp)`, `fgets(line, size, fp)`, `fscanf()`
  - ▶ Schreiben: `fputc(fp)`, `fputs(line, size, fp)`, `fprintf()`
- ▶ Schließen: `fclose(fp)`
- ▶ Positionieren: `fseek(fp, offset, vonwo)`
- ▶ Status: `stat(name, stat)` liefert Informationen über Datei (keine ANSI-C-Funktion!)

# I/O – Dateizugriff

- ▶ Zugriff auf Dateien per Name und Handle vom Type FILE
- ▶ Öffnen der Datei

```
1 FILE *fp ;  
2 fp = fopen("datei.name", "w") /* maybe NULL – do not follow! */
```

Modus:

- ▶ Lesen: "r"
- ▶ Schreiben: "w"
- ▶ Anfügen: "a"
- ▶ optional Unterscheidung zwischen Text und Binärdatei: "rb"
- ▶ Zugriff:
  - ▶ Lesen: `fgetc(fp)`, `fgets(line, size, fp)`, `fscanf()`
  - ▶ Schreiben: `fputc(fp)`, `fputs(line, size, fp)`, `fprintf()`
- ▶ Schließen: `fclose(fp)`
- ▶ Positionieren: `fseek(fp, offset, vonwo)`
- ▶ Status: `stat(name, stat)` liefert Informationen über Datei (keine ANSI-C-Funktion!)

# I/O – Dateizugriff

- ▶ Zugriff auf Dateien per Name und Handle vom Type FILE
- ▶ Öffnen der Datei

```
1 FILE *fp ;  
2 fp = fopen("datei.name", "w") /* maybe NULL – do not follow! */
```

Modus:

- ▶ Lesen: "r"
- ▶ Schreiben: "w"
- ▶ Anfügen: "a"
- ▶ optional Unterscheidung zwischen Text und Binärdatei: "rb"
- ▶ Zugriff:
  - ▶ Lesen: `fgetc(fp)`, `fgets(line, size, fp)`, `fscanf()`
  - ▶ Schreiben: `fputc(fp)`, `fputs(line, size, fp)`, `fprintf()`
- ▶ Schließen: `fclose(fp)`
- ▶ Positionieren: `fseek(fp, offset, vonwo)`
- ▶ Status: `stat(name, stat)` liefert Informationen über Datei (keine ANSI-C-Funktion!)

# I/O – Dateizugriff

- ▶ Zugriff auf Dateien per Name und Handle vom Type FILE
- ▶ Öffnen der Datei

```
1 FILE *fp ;  
2 fp = fopen("datei.name", "w") /* maybe NULL – do not follow! */
```

Modus:

- ▶ Lesen: "r"
- ▶ Schreiben: "w"
- ▶ Anfügen: "a"
- ▶ optional Unterscheidung zwischen Text und Binärdatei: "rb"
- ▶ Zugriff:
  - ▶ Lesen: `fgetc(fp)`, `fgets(line, size, fp)`, `fscanf()`
  - ▶ Schreiben: `fputc(fp)`, `fputs(line, size, fp)`, `fprintf()`
- ▶ Schließen: `fclose(fp)`
- ▶ Positionieren: `fseek(fp, offset, vonwo)`
- ▶ Status: `stat(name, stat)` liefert Informationen über Datei (keine ANSI-C-Funktion!)

# I/O – Dateizugriff

- ▶ Zugriff auf Dateien per Name und Handle vom Type FILE
- ▶ Öffnen der Datei

```
1 FILE *fp;  
2 fp = fopen("datei.name", "w") /* maybe NULL – do not follow! */
```

Modus:

- ▶ Lesen: "r"
- ▶ Schreiben: "w"
- ▶ Anfügen: "a"
- ▶ optional Unterscheidung zwischen Text und Binärdatei: "rb"
- ▶ Zugriff:
  - ▶ Lesen: `fgetc(fp)`, `fgets(line, size, fp)`, `fscanf()`
  - ▶ Schreiben: `fputc(fp)`, `fputs(line, size, fp)`, `fprintf()`
- ▶ Schließen: `fclose(fp)`
- ▶ Positionieren: `fseek(fp, offset, vonwo)`
- ▶ Status: `stat(name, stat)` liefert Informationen über Datei (keine ANSI-C-Funktion!)

# I/O – Standarddatenströme

- ▶ Vordefinierte Datei-Handles:
  - ▶ Standardeingabe `stdin`
  - ▶ Standardausgabe `stdout`
  - ▶ Standardfehlerausgabe `stderr`
- ▶ Verwendung mit Dateifunktionen ohne vorheriges Öffnen.

# I/O – Verzeichnisse (kein ANSI-C)

- ▶ Verzeichnis erzeugen: `mkdir("name", rechte)`
- ▶ in Verzeichnis wechseln: `chdir("name")`
- ▶ (leeres) Verzeichnis löschen: `rmdir("name")`
- ▶ Verzeichnis lesen:

```
1 #include <sys/types.h>
2 #include <dirent.h>
3 DIR *dp; /* Verzeichnishandle */
4 struct dirent eintrag; /* Verzeichniseintrag */
5 dp = opendir("name");
6 eintrag = readdir(dp); /* sukzessive aufrufen */
```

# Betriebssystem – Speicherverwaltung

- ▶ Speicher beim Betriebssystem beantragen: `p = malloc(size)`

```
1 #include <stdlib.h>
2 int *ip;
3 ip = (int *) malloc(1000); /* zeigt auf Block von 1000 Bytes */
```

- ▶ Speicher wieder freigeben: `free(p)`
- ▶ Speicherblockgröße verändern: `realloc(p, size)`

# Betriebssystem – Speicherverwaltung

- ▶ Speicher beim Betriebssystem beantragen: `p = malloc(size)`

```
1 #include <stdlib.h>
2 int *ip;
3 ip = (int *) malloc(1000); /* zeigt auf Block von 1000 Bytes */
```

- ▶ Speicher wieder freigeben: `free(p)`
- ▶ Speicherblockgröße verändern: `realloc(p, size)`

# Betriebssystem – Zeit

- ▶ Zeit in der Epoche (seit 1.1.1970 0:00 UTC): `time()`
- ▶ Zeit in strukturierter Fassung aufbereitet:

```
1 #include <time.h>
2 #include <stdio.h>
3 #define dlen 100
4
5 main()
6 {
7     time_t epoch;
8     struct tm now;
9     char date[dlen];
10    epoch = time(NULL);
11    now = *localtime(&epoch);
12    strftime(date, dlen, "%Y-%m-%d_%H:%M:%S_(%Z)", &now);
13    printf("%s\n", date);
14 }
```

# ANSI-C – Standardbibliotheken (Header)

`assert.h`: Implementationsabhängige Präprozessordirektiven

`float.h`: Fließpunktverarbeitung

`math.h`: Mathematische Funktionen

`stdarg.h`: Variable Argumentlistem

`stdlib.h`: Nützliche Funktionen

`ctype.h`: Typentests

`limits.h`: Implementationsabhängige Grenzen

`setjmp.h`: Implementationsabhängige Sprünge

`stddef.h`: Vordefinitionen

`string.h`: Zeichenkettenverarbeitung

`errno.h`: Fehlernummern

`locale.h`: Lokalisierung

`signal.h`: Signalbehandlung

`stdio.h`: Ein/Ausgabe

`time.h`: Zeit

# Programmieren – Hilfswerkzeuge

- ▶ Flussdiagramme (z.B. Programmablaufplan nach DIN 66001)
- ▶ Struktogramme (z.B. Nassi-Shneiderman-Diagramm)
- ▶ Software-Tools
  - ▶ Projektverwaltung, z.B. *make*
  - ▶ Debugger, z.B. *gdb*
  - ▶ Profiler, z.B. *gprof*
  - ▶ Syntax-Checker, z.B. *lint*
  - ▶ Cross-Referencer, z.B. *cxref*
  - ▶ IDE (integrated development environment), z.B. *eclipse*
  - ▶ Versionierung, z.B. *subversion*
  - ▶ lexikalische und syntaktische Codegeneratoren (z.B. *lex* und *yacc*)
- ▶ Formale Syntaxbeschreibung (z.B. Backus-Naur-Notation)

# Programmieren – Hilfswerkzeuge

- ▶ Flussdiagramme (z.B. Programmablaufplan nach DIN 66001)
- ▶ Struktogramme (z.B. Nassi-Shneiderman-Diagramm)
- ▶ Software-Tools
  - ▶ Projektverwaltung, z.B. *make*
  - ▶ Debugger, z.B. *gdb*
  - ▶ Profiler, z.B. *gprof*
  - ▶ Syntax-Checker, z.B. *lint*
  - ▶ Cross-Referencer, z.B. *cxref*
  - ▶ IDE (integrated development environment), z.B. *eclipse*
  - ▶ Versionierung, z.B. *subversion*
  - ▶ lexikalische und syntaktische Codegeneratoren (z.B. *lex* und *yacc*)
- ▶ Formale Syntaxbeschreibung (z.B. Backus-Naur-Notation)

# Programmieren – Hilfswerkzeuge

- ▶ Flussdiagramme (z.B. Programmablaufplan nach DIN 66001)
- ▶ Struktogramme (z.B. Nassi-Shneiderman-Diagramm)
- ▶ Software-Tools
  - ▶ Projektverwaltung, z.B. *make*
  - ▶ Debugger, z.B. *gdb*
  - ▶ Profiler, z.B. *gprof*
  - ▶ Syntax-Checker, z.B. *lint*
  - ▶ Cross-Referencer, z.B. *cxref*
  - ▶ IDE (integrated development environment), z.B. *eclipse*
  - ▶ Versionierung, z.B. *subversion*
  - ▶ lexikalische und syntaktische Codegeneratoren (z.B. *lex* und *yacc*)
- ▶ Formale Syntaxbeschreibung (z.B. Backus-Naur-Notation)