Informationen zur Lehrveranstaltung

- Ausführliches **Vorlesungsscript** und **Übungsaufgaben:**
  http://apps.e-technik.fh-schmalkalden.de/krause/cintro bzw. Verzeichnis J:\Krause\C
- Diese Präsentation:
  Bildschirm: http://apps.e-technik.fh-schmalkalden.de/krause/cintro/folien.pdf
  Handout: http://apps.e-technik.fh-schmalkalden.de/krause/cintro/druck.pdf
- **Literatur:**
  - Wolf, J.
    C von A bis Z
    Rheinwerk Verlag (ehemals Galileo Verlag)
    <openbook> http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/
  - Erlenkötter, H.
    C Programmieren von Anfang an
    Grundkurs Computerpraxis
    Rowohlt Taschenbuchverlag
    ISBN 978-3-499-60074-6

Zu erwerbende Kenntnisse

- Sicherer Umgamg mit Computer
  (wird für LV Informatik II vorausgesetzt)
- Entwicklungsumgebung und Entwicklungstools
- Programmiersprache
- Bibliotheken
- Coding Standards, z.B.:
  - Firmeninterne Standards
  - Branchenspezifische Standards, z.B. MISRA-C
  - Allgemeine Standards, z.B. CERT C Coding Standard

Programmiersprachen der ersten bis dritten Generation

1. **Generation: Maschinensprachen**
   Binäre hexadezimale prozessorspezifische Codes und Daten

2. **Generation: Assemblersprachen**
   Mnemonics anstelle der Hex-Codes

3. **Generation**

Prozedural:	PL/1, FORTRAN, MODULA-2, ALGOL, PASCAL, C
Objektorientiert:	PARC, EIFEL, SIMULA, SMALLTALK
Hybrid:	C++
Domänenspezifisch	
Deklarativ:	LISP

Programmiersprachen der vierten und fünften Generation

4. Generation

Scriptsprachen:	Tcl/Tk, Perl,...
Descriptive Sprachen:	IDL, XML, XSLT

5. Generation

Deklarative, logische und funktionale Sprachen:

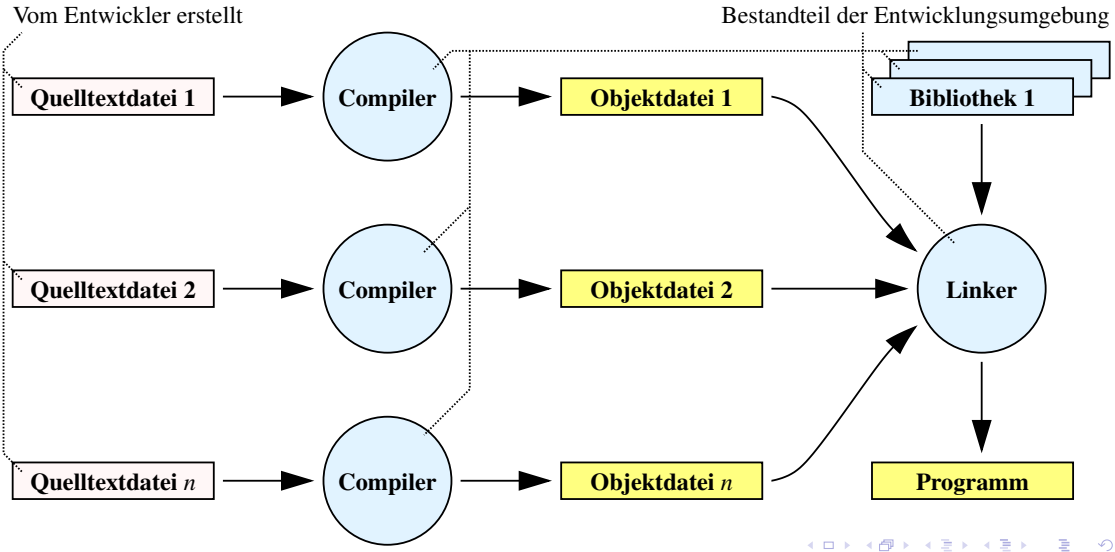
ADA	(strukturierte Sprache)
PROLOG	(logische Sprache)

Probleme nicht explizit gelöst, sondern durch Rand- und Zwangsbedingungen beschrieben

Warum C

- **Verfügbarkeit**
  Mikrocontroller, Dekstop-PCs und Server, mobile Endgeräte, Großrechner
- Verschiedene **Abstraktionsniveaus**
  - Maschinennahe Programmierung möglich
  - Programmierung auf höheren Abstraktionsniveaus durch Nutzung von Bibliotheken

Datenfluss


Vom Entwickler erstellt
Bestandteil der Entwicklungsumgebung
Quelltextdatei 1
Quelltextdatei 2
Quelltextdatei n
Compiler
Compiler
Compiler
Objektdatei 1
Objektdatei 2
Objektdatei n
Bibliothek 1
Linker
Programm

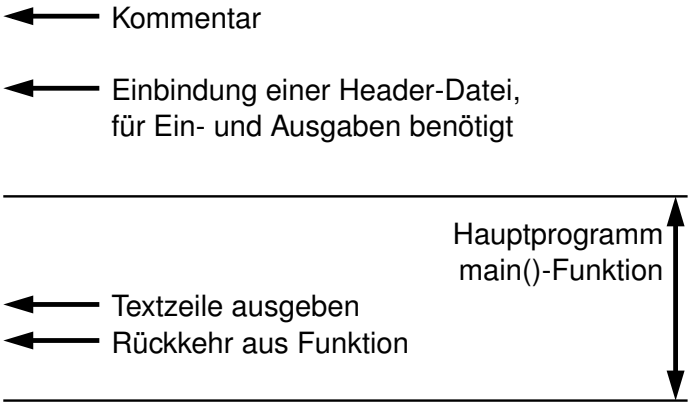
Einführungsbeispiel

```
/* Ein erstes Beispiel. */

#include <stdio.h>



int main(void)
{
    printf("Hallo\n");
    return(0);
}
```



Übungsaufgaben

- Einführungsbeispiel ausprobieren.
- Änderung, so dass es z.B. ausgibt:
  "Dies ist ein Programm von Max Mustermann."
- Was bewirken \t \n \a im Text?

Genereller Programmaufbau

- Ein C-Programm besteht aus einer **Menge von Funktionen**.
- Die Funktionen können auf **mehrere Dateien** verteilt sein.
- Die Funktionen können **nicht verschachtelt** sein.
- Die **main()**-Funktion ist das Hauptprogramm.

Empfehlungen

- **Keine verschachtelten Kommentare**
  Verschachtelte Kommentare sind laut C-Standard unzulässig.
- **Nur ASCII-Zeichen** in C-Quelltexten
  Andere Zeichen (z.B. deutsche Umlaute) sind laut Standards nicht zugelassen.
- **Maximale Zeilenlänge: 79 Zeichen**
  Empfohlen werden eher weniger, sonst evtl. Probleme beim Ausdrucken.
- **Kommentar am Dateianfang**
  Jede Quelltextdatei sollte am Anfang einen Kommentar enthalten:
  - Zweck der Funktionen in der Datei
  - Falls Datei die main()-Funktion enthält:
    Wozu dient das Programm, mit welchen Parametern wird es aufgerufen?

Datentypen

Buchstaben:	char	
Ganze Zahlen mit Vorzeichen:	int	variable Größe
	signed char	1 Byte
	short	2 Bytes
	long	mind. 4 Bytes
Ganze Zahlen ohne Vorzeichen:	unsigned	variable Größe
	unsigned char	1 Byte
	unsigned short	2 Bytes
	unsigned long	mind. 4 Bytes
Gleitkommazahlen:	float	6 Dezimalstellen genau, Verwendung nicht mehr empfohlen
	double	15 Dezimalstellen genau
	long double	mind. 19 Dezimalstellen genau
Zeiger:	*Typ* *	Addresse eines Objektes im Hauptspeicher, mit Information über Typ

Größe von Ganzzahl-Datentypen

Datentyp	16-Bit-CPU	32-Bit-CPU	64-Bit-CPU	
			Windows	Linux
char	8	8	8	8
short	16	16	16	16
int	16	32	32	32
long	32	32	32	64
long long	-	64	64	64
intmax_t	-	groß genug für verlustfreie Konvertierung aller anderen Ganzzahl-Datentypen zu intmax_t		

Die Größen gelten analog für unsigned char, unsigned short, unsigned, unsigned long, unsigned long long und uintmax_t.

Wertebereiche von Ganzzahl-Datentypen

Größe in Bits	Vorz.	Minimalwert	Maximalwert
8	nein	0	$2^8 - 1 = 255$
8	ja	$-2^7 = -128$	$2^7 - 1 = 127$
16	nein	0	$2^{16} - 1 = 65535$
16	ja	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
32	nein	0	$2^{32} - 1 = 4294967295$
32	ja	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
64	nein	0	$2^{64} - 1 = 18446744073709551615$
64	ja	$-2^{63} = -9223372036854775808$	$2^{63} - 1 = 9223372036854775807$

Variablennamen

- **Deklaration**
  Variablen müssen vor Verwendung deklariert werden, dabei wird Datentyp und Variablenname angegeben.
- **Variablennamen**
  und alle anderen Bezeichner (wie z.B. Typnamen, Funktionsnamen, Sprungmarken):
  - **beginnen** mit Buchstaben oder Unterstrich (Unterstrich nicht empfohlen),
  - **darauf folgen** Buchstaben, Zahlen oder Unterstriche beliebig gemischt.
  - **Reservierte Wörter**
    dürfen nicht als Bezeichner benutzt werden:

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

Beispielprogramm für Variablen: Quadratflächenberechnung

```c
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>

int main(void)
{
   double     slaenge;     /* Seitenlaenge */
   double     flaeche;     /* Flaeche */

   printf("Bitte Seitenlaenge angeben: ");
   scanf("%lg", &slaenge);
   flaeche = slaenge * slaenge;
   printf("Flaecheninhalt: %lg\n", flaeche);
   return 0;
}
```

- **Deklaration** vor der Verwendung (z.B. Definition am Blockanfang vor erster Anweisung)
- **Wert setzen** bevor Wert benutzt wird

Empfehlungen (1)

- **Gut unterscheidbare Namen**
  Variablennamen sollten sich an mindstens einer Stelle durch Zeichen unterscheiden, die nicht zu den nachfolgenden je nach Schriftart optisch schlecht unterscheidbaren Kombinationen gehört:
  - 0 (Zahl 0), O (großes o), D (großes d)
  - 1 (Zahl 1), I (großes i), l (kleines L)
  - 2 (Zahl 2), Z (großes z)
  - 5 (Zahl 5), S (großes s)
  - 8 (Zahl 8), B (großes b)
  - n (kleines N), h (kleines H)
  - rn (kleines RN), m (kleines M)

Empfehlungen (2)

- **Keine Überdeckung von Identifiern**
  Vorhandene Variablen und Funktionen nicht überdecken.
- **Nur eine Variablendeklaration pro Zeile**
  damit jede Variable einzeln kommentiert werden kann.
- **Eindeutige Namen, nicht zu lang**
  Der C-Standard sind in Namen für externe Identifier (Variablen- und Funktionsnamen) nur die ersten 31 Zeichen signifikant.
  Für interne Identifier und Makronamen 64 Zeichen.
- **Kleinschreibung**
  am Beginn von Variablennamen, Großschreibung am Anfang nur für Konstanten und Makros.

Empfehlungen (3)

- **Aussagekräftige Variablennamen**
  z.B. ist „seiten_laenge“ oder „seitenLaenge“ informativer als „s“ oder „sl“. Mindestens 2 Zeichen lange Namen (außer in Sonderfällen).
- **Suchbare Namen**
  Beispiel: Text-Suche nach „nt“ findet auch alle „nt“ in printf().
- **Variablen initialisieren**
  d. h. mit Anfangswerten versehen.

Initialisierung von Variablen

Mit

Typ Variablenname = *Wert*;

wird in der Definition ein Anfangswert vorgegeben.

Anfangswert: Konstante oder Verknüpfung von Konstanten durch Operatoren

Zeichen-Literale

Literale sind Wertangaben, die direkt im C-Quelltext stehen.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char  c1 = 'A';          /* Zeichen 'A' direkt angegeben */
6    char  c2 = 0x42;         /* Zeichen 'B' als Hexadezimalwert */
7
8    printf("c1=%c␣c2=%c\n", c1, c2);
9    return 0;
10 }
```

Zeichen-Literale

- direkt in einfachen Anführungszeichen oder
- als Dezimalwert oder
- als Hexadezimalwert mit vorangestelltem „0x“.

String-Literale

Zeichenketten (Strings) sind Folgen von Zeichen, mit Nullbyte abgeschlossen.

```
1  #include <stdio.h>
2
3  const char s1[] = { "Dies␣ist␣ein␣Text." };
4  const char s2[] = {
5    'D', 'i', 'e', 's', '␣', 'a', 'u', 'c', 'h', '.', '\0'
6  };
7
8  int main(void)
9  {
10   printf("s1=\"%s\"␣s2=\"%s\"\n", s1, s2);
11   return 0;
12 }
```

- Entweder String in **doppelte Anführungszeichen** setzen oder
- **Elemente einzeln** angeben, dabei abschließendes **Nullbyte** nicht vergessen!

Backslash-Sequenzen in Zeichen und Strings

\a	Bell (akustisches Signal)
\t	Tabulator
\b	Backspace
\v	Vertikal-Tabulator
\r	Carriage return (Wagenrücklauf)
\n	Newline (Zeilenwechsel)
\\	Backslash
\'	Einfaches Anführungszeichen, wenn dieses als einfaches Zeichen angegeben wird
\"	Doppeltes Anführungszeichen, wenn es in einem String-Literal steht
xxx	Oktalcode (Bereich 000...377 für ein Byte)

int-Literale

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int   beste_note      = 1;
6    int   studenten       = 0x20;        /* Hexadezimal fuer 32 */
7    int   dauer           = 0132;        /* Oktal-Darstellung fuer 90 */
8
9    printf("Es nahmen %d Studenten an der Klausur teil.\n", studenten);
10   printf("Die Klausur dauerte %d Minuten.\n", dauer);
11   printf("Die beste Note war %d.\n", beste_note);
12   return 0;
13 }
```

- Normale Darstellung: Dezimalzahl (keine führende 0).
- Hexadezimal-Darstellung: „0x“ voranstellen.
- Oktal-Darstellung: „0“ voranstellen.

Variationen für andere Ganzzahl-Datentypen

Ein nachgestelltes „U“, „L“, „UL“, „LL“ bzw. „ULL“ kennzeichnet unsigned-, long-, unsigned-long-, long-long- bzw. unsigned-long-long-Werte, siehe Beispiel:

```
1 unsigned               u    = 1U;   /* nachgestelltes U */
2 long                   l    = 1L;   /* nachgestelltes L */
3 unsigned long          ul   = 1UL;  /* nachgestelltes UL */
4 long long              ll   = 1LL;  /* nachgestelltes LL */
5 unsigned long long     ull  = 1ULL; /* nachgestelltes ULL */
```

double-Literale

```c
#include <stdio.h>

int main(void)
{
   double erd_masse = 5.9722e24;          /* Masse der Erde in kg. */

   printf("Die Masse der Erde ist %lg kg.\n", erd_masse);
   return 0;
}
```

- Wissenschafliche Schreibweise (scientific notation):
- 5.9722e24 bedeutet $5,9722 \cdot 10^{24}$.
- Punkt anstelle des Komma als Dezimaltrenner.
- Falls erforderlich: Ganzzahliger Exponent zur Basis 10, mit „e“ abgetrennt.
- Dezimalpunkt oder Exponent erforderlich zur Unterscheidung von Ganzzahlwerten.

Empfehlungen

- **L für long-Konstanten**
  anstelle des theoretisch auch möglichen „l“ (Verwechslungsgefahr mit 1).
- **Keine 0 am Anfang von Dezimalwerten**
  da dies zur Behandlung als Oktalwert führen würde.
- **.0 für ganzzahlige double-Werte**
  double a = 4.0;
  entspricht eher den Lesegewohnheiten als
  double a = 4.;

Zuweisungen

Variable = *Wert*

bzw. allgemein

lvalue = *Wert*

Ein *lvalue* ist eine Variable oder ein anderes Konstrukt (z. B. unter Zuhilfename von Zeigern), dem ein Wert zugewiesen werden kann.

Der Wert ist ein Term (Berechnungsergebnis, engl.: expression). Er ensteht durch sinnvolle Verknüpfung von Konstanten, Operatoren und Funktionsaufrufen.

Arithmetische Operatoren

- `-` unäres Minus (negatives Vorzeichen)
- `*` Multiplikation
- `/` Division
- `%` Modulo-Division (Divisionsrest der ganzzahligen Division)
- `+` Addition
- `-` Subtraktion

Divisionsrest bei Modulo-Division mit % ist negativ, wenn Zähler negativ ist.

Bitweise Operatoren

~	Einerkomplement (alle Bits negieren)
&	Bitweises UND (AND)
|	Bitweises ODER (OR)
^	Bitweises exklusives ODER (XOR)
<<	Bitweise Verschiebung nach links Beispiel 0x0001U << 5 $\rightarrow$ 0x0020U
>>	Bitweise Verschiebung nach rechts Beispiel 0x0040U >> 6 $\rightarrow$ 0x0001U

Bitweise Verschiebung nur für *vorzeichenlose* Zahlen einsetzen!
Bitweise Verschiebung nicht mehr zum Optimieren mathematischer Operationen verwenden!

Verkürzte Zuweisungsoperatoren

Operator	Beispiel		Auswirkung
*=	a *= b;	↔	a = a * b;
/=	a /= b;	↔	a = a / b;
+=	a += b;	↔	a = a + b;
-=	a += b;	↔	a = a - b;
&=	a &= b;	↔	a = a & b;
|=	a |= b;	↔	a = a | b;
<<=	a <<= b;	↔	a = a<< b;
>>=	a >>= b;	↔	a = a>> b;

Vergleichsoperatoren

`<`	$a < b$
`<=`	$a \leq b$
`==`	$a = b$
`!=`	$a \neq b$
`>`	$a > b$
`>=`	$a \geq b$

Logische Operatoren

!	Logische Negation einer Bedingung (!*a* bedeutet: *a* ist nicht erfülllt)
&&	Logisches UND
||	Logisches ODER

Ganzzahlwerte oder Zeiger als Logikwerte

- Ganzzahlige Werte
  - Wert 0 $\rightarrow$ false (falsch)
  - Wert $\neq$ 0 $\rightarrow$ true (wahr)
- Zeiger
  - Wert NULL $\rightarrow$ false (falsch)
  - Wert $\neq$ NULL $\rightarrow$ true (wahr)

```
if (i != 0) {
...
}
if (pointer != NULL) {
...
}
```

```
if (i) {
...
}
if (pointer) {
...
}
```

Unterschied zwischen bitweisen Operatoren und Logikoperatoren

```
short x, y, z;
...
x = 1;          /* bitweise dargestellt: 0000000000000001 */
y = 2;          /* bitweise dargestellt: 0000000000000010 */
z = x & y;      /* z wird 0 */
z = x && y;     /* z wird beliebiger Wert ungleich 0 (repräsentiert Logikwert true) */
z = x | y;      /* z wird 3 */
z = x || y;     /* z wird beliebiger Wert ungleich 0 (repräsentiert Logikwert true) */
```

Inkrement- und Dekrement-Operatoren

Operator	Beispiel			Auswirkung
++	a = ++i;	↔	i = i + 1; a = i;	Inkrementierung vor Zuweisung
	a = i++;	↔	a = i; i = i + 1;	Inkrementierung nach Zuweisung
--	a = --i;	↔	i = i - 1; a = i;	Dekrementierung vor Zuweisung
	a = i--;	↔	a = i; i = i - 1;	Dekrementierung nach Zuweisung

Entscheidungsoperator

Bedingung ? *Wahr-Wert* : *Falsch-Wert*

- Bedingung erfüllt:
  gesamter Ausdruck ergibt den Wahr-Wert
- Bedingung nicht erfüllt:
  gesamter Ausdruck ergibt den Falsch-Wert

Beispiel: Maximum zweier Zahlen

```
int a, b, max;
a = ...;
b = ...;
max = ((a > b) ? (a) : (b));
```

Empfehlung: Klammern um Bedingung, Wahr-Wert, Falsch-Wert und Gesamtkonstrukt.

Vorrang und Assoziativität

() [] .> .	linksassoziativ
! ~ ++ -- * & sizeof	linksassoziativ
* / %	linksassoziativ
+ -	linksassoziativ
<< >>	linksassoziativ
< != >= >	linksassoziativ
== !=	linksassoziativ
&	linksassoziativ
^	linksassoziativ
|	linksassoziativ
&&	linksassoziativ
||	linksassoziativ
?:	linksassoziativ
= *Operator*=	rechtsassoziativ
,	rechtsassoziativ

Linksassoziativ:

Bearbeitung beginnt mit linker Operation.

a = b + c + d;

a = (b + c) + d;

Rechtsassoziativ:

Bearbeitung beginnt mit rechter Operation.

a = b = c = d;

a = (b = (c = d));

Empfehlungen

- **Klammern verwenden,**
  um Priorität der Abarbeitung festzulegen.
  Damit werden Fehler vermieden, der Quelltext wird übersichtlicher.
- **Keinen nicht initialisierten Speicher auslesen.**
  Bevor Variablen verwendet werden, muss Wert zugewiesen worden sein!
- **Klammern im Entscheidungsoperator verwenden,**
  sowohl um Bedingung, Wahr- und Falsch-Wert als auch um gesamtes Operatorkonstrukt.
- **Zeilenumbruch an Operator**, falls erforderlich. Operator in zweiter Zeile und Folgezeile unter erstem Operand. Beispiel:

```
wert = eine_ganz_ganz_lange_variable * eine_andere_lange_variable
       + noch_eine_variable * und_immer_noch_eine_andere_variable;
```

if

```
if (Bedingung) {
  /* Anweisungen, falls Bed. erfuellt */
} else {
  /* Anweisungen, falls nicht erfuellt */
}
```

```
if (Bedingung) {
  /* Anweisungen, falls Bed. erfuellt */
}
```

Je nachdem, ob die Bedingung erfüllt ist, wird entweder der eine oder der andere Programmzweig ausgeführt.
Kürzere Variante rechts: Nur ein Programmzweig, dieser wird nur ausgeführt, falls die Bedingung erfüllt ist.

Einrückungen folgen Programmstruktur

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    double    slaenge;    /* Seitenlaenge */
6    double    flaeche;    /* Flaecheninhalt */
7    printf("Bitte Seitenlaenge fuer Quadrat angeben: ");
8    scanf("%lg", &slaenge);
9    if (0.0 <= slaenge) {
10     flaeche = slaenge * slaenge;
11     printf("Flaecheninhalt: %lg\n", flaeche);
12   } else {
13     printf("FEHLER: Seitenlaenge darf nicht negativ sein!\n");
14   }
15   return 0;
16 }
```

Gegenbeispiel: Unsinnige Einrückungen

```
1                          #include <stdio.h>
2
3       int  main(void)
4   {
5 double                    slaenge;
6     double
7   flaeche;
8           printf("Bitte Seitenlaenge fuer Quadrat angeben: ");
9       scanf("%lg", &slaenge);
10        if (0.0 <= slaenge)
11  {
12          flaeche = slaenge * slaenge;
13            printf("Flaecheninhalt: %lg\n", flaeche); }
14        else {
15      printf("FEHLER: Seitenlaenge darf nicht negativ sein!\n"); }
16  return 0; }
```

for

Hauptsächlich genutzt, wenn Anzahl der Schleifendurchläufe von Beginn an bekannt

```
for (Starteinstellungen; Fortsetzungsbedingung; Änderung) {
  /* Anweisungen, die mehrfach ausgeführt werden sollen */
}
```

```
for (i = 1; i <= 5; i++) {
  printf("Heute verteilen wir Note %d\n", i);
}
```

Starteinstellungen:	Setzen einer Variable auf Anfangswert
Fortsetzungsbedingung:	Bedingung für erneuten Schleifendurchlauf
Änderung:	Erhöhung oder Änderung von Variablen, erfolgt nach Abarbeitung des Schleifeninhaltes

while

```
while (Bedingung) {
  /* Anweisungen, die mehrfach ausgeführt werden sollen */
}
```

```
i = 1;
while (i < 5) {
  printf("Note %d: bestanden.\n", i);
  i++;
}
printf("Note %d: nicht bestanden.\n", i);
```

- **Test** erfolgt jeweils **vor Abarbeitung** des Schleifeninhaltes.
- **Abweisende Schleife:**
  Gar keine Abarbeitung des Schleifeninhaltes, wenn Bedingung gleich in erstem Test nicht erfüllt.

do . . . while

```
do {
  /* Anweisungen, die mehrfach ausgeführt werden sollen */
} while(Bedingung);
```

```
i = 1;
do {
  printf("Note %d: bestanden.\n", i++);
} while (i <= 4);
printf("Note %d: nicht bestanden.\n", i);
```

- **Test** erfolgt jeweils **nach Abarbeitung** des Schleifeninhaltes.
- **Nichtabweisende Schleife:**
  Schleifeninhalt wird mindestens einmal abgearbeitet, da erster Test erst danach.

switch / case

```
switch (int-Ausdruck) {
  case Konstant-Wert1: {
    /* Anweisungsblock1 */
  } break;
  case Konstant-Wert2: {
    /* Anweisungsblock2 */
  } break;
  ...
  default: {
    /* Std.-Anweisungsblock */
  } break;
}
```

```
switch (note) {
  case 1: {
    printf("Sehr gut.\n");
  } break;
  case 2: {
    printf("Gut.\n");
  } break;
  ...
  default: {
    printf("Note gibt es nicht.\n");
  } break;
}
```

- **case**-Anweisungen sind Einsprungpunkte.
- **break** nach jedem Zweig nicht vergessen, sonst Abarbeitung auch des nachfolgenden Codes!

Empfehlungen (1)

- **Geschweifte Klammern für Anweisungsblöcke** von if, else, for, while, do...while, case
  Obwohl auch eine einzelne Anweisung ohne geschweifte Klammern möglich wäre, immer die geschweiften Klammern verwenden.
- **Grundgerüst zuerst komplett schreiben**, dann ausfüllen.
  Also zunächst

  ```
  if () {
  } else {
  }
  ```

  schreiben, damit Klammern und Einrückungstiefe stimmig sind. Dann Bedingung, if- und ggf. else-Zweig ausfüllen.

Empfehlungen (2)

- **Lvalue in Vergleichen rechts**
  Konstrukte, denen etwas zugewiesen werden kann, sollten in Vergleichen möglichst rechts stehen (vereinfacht gesagt: Konstanten links, Variablen rechts). Damit wird versehentliche Zuweisung anstelle eines Vergleiches bei Tippfehler „=“ statt „==“ vermieden.
  Beispiel:

```
if (i == 5) {
...
}
if (i = 5) { /* Zuweisung */
...
}
```

```
if (5 == i) {
...
}
if (5 = i) { /* Compiler-Fehler */
...
}
```

Empfehlungen (3)

- **Keine Zuweisungsoperationen in Bedingungen**
  Damit ist klar, dass alle Compilerwarnungen über Zuweisungen in Bedingungen ernst genommen werden müssen und Tippfehler sind.
- **Schneller Abbruch bei logischem UND und ODER**
  Logische Verknüpfungen werden von links nach rechts abgearbeitet.
  Steht nach Test einer Bedingung schon das Endergebnis fest, werden weitere unnötige Tests vermieden.
- **Klammern bei Verknüpfung logischer Bedingungen**
  vermeiden Fehler und erhöhen die Übersichtlichkeit.

Empfehlungen (4)

- **Leerzeichen nach if, for, while, do, switch...**
  sorgen für bessere Lesbarkeit.
- **Zeilenumbruch an Operator**, falls erforderlich. Einrückung so, dass Operator unter erstem Operanden steht.

```
if ((EINE_GANZ_GANZ_LANGE_KONSTANTE == ganz_lange_variable)
    || (EINE_ANDERE_GANZ_LANGE_KONSTANTE == andere_variable))
{
  ...
}
```

Funktionen - Prototyp / Deklaration

[*Speicherklasse*]
Rückgabetyp
Funktionsname(*Argument(e)*);

Beispiel:

```
double
quadermasse(double laenge, double breite, double hoehe, double dichte);
```

- **Rückgabetyp**
  Datentyp, den das Ergebnis der Funktion hat.
- **Argumente**
  Datentyp des jeweiligen Argumentes und ein Name.

Funktionen - Definition (allgemein)

Rückgabetyp
Funktionsname(*Argument(e)*);
{
  Variablendeklarationen

  Anweisungen
}

Funktionen - Definition (Beispiel)

```
/* Masse eines Quaders berechnen.
   laenge, breite, hoehe: Laenge, Breite und Hoehe jeweils in Zentimetern.
   dichte: Dichte des Quaders in Gramm pro Kubikzentimeter.
   Rueckgabewert: Die berechnete Quadermasse in Gramm.
*/
double
quadermasse(double laenge, double breite, double hoehe, double dichte)
{
  double volumen;
  double masse;

  volumen = laenge * breite * hoehe;
  masse = volumen * dichte;
  return masse;
}
```

Nutzung

```
double m1; /* Masse des ersten Quaders */
double m2; /* Masse des zweiten Quaders */
double anfang; /* x-Koordinate, an der zweiter Quader beginnt */
double ende; /* x-Koordinate, an der zweiter Quader endet */
anfang = ...
ende = ...
m1 = quadermasse(3.0, 4.0, 2.0, 10.0);
m2 = quadermasse((ende - anfang), 5.0, 3.0, 15.0);
```

- **Reihenfolge** der Argumente im Funktionsaufruf passend zur Reihenfolge der Parameter im Prototyp.
- **Argumente im Funktionsaufruf** können durch Berechnungen oder wiederum Funktionsaufrufe zustande kommen.

Vorgänge bei Nutzung einer Funktion

- **Speicherreservierung auf dem Stack** für
  - Argumente der Funktion
  - Lokale Variablen der Funktion
  - Rückgabewert
- **Wertübergabe**
  Für jeden Funktionsparameter wird der Wert bestimmt und in den soeben für die Argumente der Funktion reservierten Speicherbereich eingetragen.
  Funktion besitzt lokale Kopie des jeweiligen Wertes.
- **Initialisierung der lokalen Variablen**
  durch Eintrag der entsprechenden Werte in den oben reservierten Speicherbereich.
- **Einsprung in den Code für den Anweisungsblock der Funktion**
  Der Code arbeitet mit dem oben reservierten Speicher.
- **return-Anweisungen** im Code tragen Funktionsergebnis in vorgesehenen Speicher für Rückgabewert ein und veranlassen Rücksprung zu aufrufendem Code.
- **Abruf/Verarbeitung des Funktionsergebnisses** und
  Freigabe des oben reservierten Speichers.

Parameter und lokale Variable

- **Funktionsargumente werden als Wert übergeben.**
- **Funktion hat eigene Kopie der Parameter**,
  Änderungen der Funktionsparameter haben keine Rückwirkungen auf aufrufenden Code.
- **Variablen in Funktionen sind lokal**
  und somit unabhängig von anderen gleichnamigen Variablen in anderen Funktionen.
  Namensüberdeckungen mit anderen sichtbaren Funktionen oder globalen Variablen sollten vermieden werden.

Empfehlungen (1)

- **Prototypen für alle Funktionen**
  Für alle verwendeten Funktionen muss der Compiler den Prototyp kennen.
- **Argumente prüfen**
  Funktionen sollten testen, ob alle Funktionsparameter gültige Werte aufweisen (z. B. keine negativen Seitenlängen für geometische Objekte...).
- **return-Anweisung sicherstellen**
  Für Funktionen mit einem Rückgabetyp muss in jedem Fall die Rückkehr aus der Funktion über eine return-Anweisung erfolgen, den den Ergebniswert festlegt.
  Möglichst nur eine return-Anweisung am Ende der Funktion.
- **Initialisierung der lokalen Funktion**
  Variable für Rückgabewert auf Kennwert für „Fehler aufgetreten“ initialisieren, alle anderen Variablen auf geeignete Startwerte.

Empfehlungen (2)

- **Funktionsname an linkem Rand**
- **Geschweifte Klammern für Funktionsblock am linken Rand**
- **„Sprechende“ Funktions- und Parameternamen**

```
double
quadermasse(double laenge, double breite, double hoehe, double dichte);
```

ist besser als

```
double
qm(double l, double b, double h, double d);
```

Feld anlegen

Feld: Folge von mehreren Elementen gleichen Typs, zusammengefasst in einer Variable.

Felddefinition mit **Angabe der Größe**:

Typ Name[*Größe*] double temperatur[180];

oder durch Angabe der **Initialisierungswerte**:

Typ Name[] = { *Werte-Liste* }; int notenliste[] = { 1, 5, 3, 2, 1, 2, 2, 3 };

Zugriff auf Feldelemente

- Zugriff über ***Name*[*Index*]**
- **Index beginnt bei 0** und endet bei Feldgröße-1.

```
double temperatur[180];
...
temperatur[23] = 27.5;
...
x = temperatur[48];
```

size_t / sizeof

size_t: Ganzzahliger Datentyp für Größenangaben
hauptsächlich für: Größen von Objekten im Speicher, Feldgrößen, Indizes

sizeof: Operator, liefert Größe von Variablen und Datentypen
als size_t-Wert

```
double temperatur[180];
size_t s1, s2, s3;

s1 = sizeof(temperatur);   /* Feldgroesse in Bytes */
s2 = sizeof(double);       /* Byte-Anzahl eines double-Wertes */
s3 = s1 / s2;              /* Feldgroesse als Anzahl der Elemente */
```

Mehrdimensionale Felder ohne Initialisierung

Datentyp Variablenname[*Anzahl1*][*Anzahl2*]; int tf[2][3];

- *tf* ist zweidimensionales Feld.
- Ausdehnung in der ersten Dimension: 2 (Indizes 0 und 1).
- Jedes Element der ersten Dimension ist ein Feld, hier mit 3 Elementen (Indizes 0...2).

Mehrdimensionale Felder mit Initialisierung

```
int tf[2][3] = {
  { 1, 2, 3 },
  { 4, 5, 6 }
};
```

```
/* Erste Dim: 0...1, zweite Dim: 0...2 */
/*   tf[0][0],   tf[0][1],   tf[0][2]   */
/*   tf[1][0],   tf[1][1],   tf[1][2]   */
```

Möglich, aber *nicht empfohlen*, wäre auch:

```
int tf[2][3] = { 1, 2, 3, 4, 5, 6 };
```

Zeiger und Adressoperator &

*Typ *Name*;

```
int notenliste[] = { 1, 2, 3, 2, 1, 5 };
int *ptr = NULL;
...
ptr = &(notenliste[0]);   /* Adressbildung mit Adressoperator */
ptr = notenliste;   /* Vereinfachte Form für Feldanfang */
```

- ... enthält:
  - **Adresse** eines Objektes im Speicher und
  - Information über **Datentyp** des Objektes.
- ... muss vor Verwendung auf Adresse des jeweiligen Objektes gesetzt werden (Variable oder Feld).
- ... sollten bei Definition auf NULL initialisiert werden.

Adressoperator (vorangestelltes &) ermittelt die Adresse von Variablen (z. B. um sie einem Zeiger zuzuweisen).

Benutzen von Zeigern

*Zeiger
Zeiger[*Index*]

```
int notenliste[] = { 2, 2, 3, 2, 1, 5 };
int *ptr = NULL;

ptr = &(notenliste[0]);

*ptr   = 1; /* Noch Punkte gefunden */
ptr[5] = 4; /* Doch nicht durchgefallen */
printf("Note des ersten Studenten: %d\n", *ptr);
printf("Note des dritten Studenten: %d\n", ptr[2]);
```

Zugriff

- Operator *
  Zeiger-Dereferenzierung oder
- über Angabe des Index,
  wenn Zeiger auf ein Feld zeigt.

Inkrementieren und Dekrementieren von Zeigern (Beispiel)

```
#include <stdio.h>
static int nl[] = { 1, 2, 3, 2, 1, 4 };
static const size_t sz_nl = sizeof(nl) / sizeof(int);   /* Groesse des Feldes */
int main(void)
{
  int *ptr = NULL;
  size_t ind = 0;
  ptr = nl;                                             /* auf Feldanfang setzen */
  for (ind = 0; ind < sz_nl; ind++) {
    printf(
      "Student %u hat Note %d\n",
      (unsigned)(ind+1), *(ptr++)                       /* nach Verwendung weiterruecken */
    );
  }
  return 0;
}
```

Zeigerarithmetik

$Zeiger + Index$ $\&(Zeiger[Index])$

ergibt einen Zeiger auf das *Index*-te Element des Feldes, auf das der *Zeiger* zeigt.

Zeigerarithmetik (Beispiel)

```
#include <stdio.h>
static int nl[] = { 1, 2, 3, 2, 1, 4 };
static const size_t sz_nl = sizeof(nl) / sizeof(int);   /* Groesse des Feldes */
int main(void)
{
  int *ptr = NULL;
  size_t ind = 0;
  ptr = nl;                                             /* auf Feldanfang setzen */
  for (ind = 0; ind < sz_nl; ind++) {
    printf(
      "Student %u hat Note %d\n",
      (unsigned)(ind+1), *(ptr+ind)                     /* Verwendung Zeigerarithmetik */
    );
  }
  return 0;
}
```

Hinweise, Empfehlungen

- **Zeiger initialisieren**
  Zeiger müssen vor Benutzung auf eine gültige Adresse gesetzt werden.
  Initialisierung mit NULL in Definition zeigt, dass Zeiger noch nicht auf eine gültige Adresse zeigt.
- **Zeigerarithmetik nur für Zeiger, die auf dasselbe Feld zeigen**
  Zeigerarithmetik und Vergleiche von Zeigern nur für Zeiger, die sich auf dasselbe Feld beziehen!
- **Geeignete Bibliotheksfunktionen benutzen**
  Nach Möglichkeit nur solche Bibliotheksfunktionen benutzen, die nicht nur einen Zeiger auf ein Feld übergeben bekommen, sondern auch die Feldgröße.
- **Automatische Größenberechnung** für initialisierte Felder
  Mit
  sizeof(*Variable*)/sizeof(*Typ*)
  kann die Größe eines Feldes berechnet werden.

Strings

- Feld von char, mit 0-Byte abgeschlossen
- String "Hallo" wird dargestellt durch 6 Bytes: 'H', 'a', 'l', 'l', 'o', 0x00
- Entweder in les- und schreibbarem Speicher oder nur lesbarem Speicher (const).
- **Literale Strings immer nur lesbar!**

Literale Strings (String-Konstanten)

Zeiger auf anonymes Feld
(String-Literal verwendet, daher const):

```
const char *test_ptr = "Hallo";
```

Feld mit Namen
(String-Literal verwendet, daher const):

```
const char test_str[] = { "Hallo" };
```

Feld mit Namen
(kein String-Literal, daher nicht const):

```
char test_str[] = { 'H', 'a', 'l', 'l', 'o', '\0' };
```

Abschließendes Nullbyte nicht vergessen!

const in Variablen-Deklarationen/Definition

- const unmittelbar vor oder nach Datentyp legt fest, dass die Daten nicht änderbar sind.
- const bezieht sich immer auf „das Nächstliegende“.
- const char *test_ptr = "Hallo";
  char const *test_ptr = "Hallo";
  test_ptr ist ein Zeiger auf Zeichen, die nicht änderbar sind.
- char * const test_ptr = ...;
  Der Zeiger test_ptr ist nicht änderbar, aber die Zeichen, auf die mit test_ptr zugegriffen werden kann, können geändert werden.
- const char * const test_ptr = "Hallo";
  test_ptr ist ein Zeiger auf nicht änderbare Zeichen, auch die Variable test_ptr kann nicht geändert werden.

Felder mit String-Literalen

Z. B. für Ausgabe von Hilfetexten:

```
const char * const hilfe_text[] = {
  "Benutzung",
  "",
  "tolles_kopier_programm Quelldatei Zieldatei",
  "",
  "Quelldatei: Name der Datei, die kopiert werden soll",
  "Zieldatei: Name der Kopie, die erzeugt werden soll",
  NULL
};
```

Weder die eigentlichen Textzeichen, noch die Feldelemente (String-Zeiger) sind änderbar.

Felder mit String-Literalen (Zugriff)

Wird mit Zeigern auf String-Literal-Felder zugegriffen, müssen auch in der Zeiger-Definition die passenden const-Zusätze stehen, z. B.:

```
const char * const *ptr;
...
ptr = hilfe_text;
while (NULL != *ptr) {
  fputs(*(ptr++), stdout);
  fputc('\n', stdout);
}
fputc('\n', stdout);
```

const in Funktionsparametern

- für Funktionsparameter, die nicht geändert werden sollen

```
char *strcpy(char *dest, const char *src);
```

Der Parameter *src* der Funktion *strcpy()* (zum Kopieren von Strings) ist ein Zeiger auf nicht änderbare Zeichen.
Der Parameter *dest* ist ein Zeiger auf den Zielbereich. Auf den Zielbereich wird schreibend zugegriffen – der String soll ja dorthin kopiert werden – deshalb ist der Zielbereich nicht als const markiert.

const im Rückgabewert

```
const
char *
irgendeinefunktion(const char *src);
```

Ein const-Zeiger als Rückgabewert einer Funktion bedeutet, dass dieser Zeiger auf Daten zeigt, die nicht verändert werden dürfen.

String-Funktionen (Auswahl)

```
#define _CRT_SECURE_NO_WARNINGS 1
#include <string.h>
```

Funktion	Zweck
strlen	Länge ermitteln
strcpy	Kopieren
strcat	Anhängen
strcmp	Vergleichen
strchr	Erstes Vorkommen eines Zeichens suchen
strrchr	Letztes Vorkommen eines Zeichens suchen

strlen - Länge ermitteln

size_t
strlen(const char *s);

- liefert Länge des Strings *s*, **ohne das abschließende Nullbyte** mitzuzählen.
- Soll der String *s* in einen Puffer kopiert werden, so muss die Größe des Puffers mindesten 1 Byte größer sein als das Ergebnis von *strlen()*.

strcpy - String kopieren

char *
strcpy(char *destination, const char *source);

- kopiert den Text von *source* nach *destination*.
- Vor Aufruf sicherstellen, dass *destination* groß genug ist, um den String aufzunehmen und das abschließende Nullbyte.

strcat - String anhängen

char *
strcat(char *destination, const char *source);

- hängt den Text von *source* an den bereits im Puffer *destination* stehenden Text an.
- Vor Aufruf sicherstellen, dass der Puffer *destination* groß genug ist, um den bereits enthaltenen String, den String *source* und das abschließende Nullbyte aufzunehmen.

strcmp - Strings vergleichen

```
int
strcmp(const char *s1, const char *s2);
```

- vergleicht zwei Strings *s1* und *s2*.
- Negativer Rückgabewert, falls *s1* lexikographisch kleiner als *s2*.
- 0, falls beide Strings gleich sind.
- Positiver Rückgabewert, falls *s1* lexikographisch größer als *s2*.
- Groß- und Kleinschreibung wird unterschieden.
- Funktionen *stricmp()* (Windows) bzw. *strcasecmp()* (POSIX) für Vergleich ohne Unterscheidung zwischen Groß- und Kleinschreibung.

strchr / strrchr - Zeichen in String suchen

char *
strchr(const char *s, int c);

- sucht das erste Vorkommen des Zeichens c im String.
- Rückgabewert:
  - Gültiger Zeiger, wenn Zeichen gefunden bzw.
  - NULL wenn Zeichen nicht gefunden.
- Variante *strrchr()* sucht nach letztem (am weitesten rechts stehenden) Vorkommen des Zeichens.

Hinweise

- **Literale Strings nicht modifizieren**
  Der Compiler legt diese in nur lesbarem Speicherbereich ab.
- **Für literale Strings keine explizite Feldgröße angeben**
  Sonst Gefahr, dass 0-Byte vergessen wird. Bei Änderungen Gefahr, dass Änderung der Feldgröße vergessen wird.
- **Ausreichende Puffergröße**
  Vor Kopier- bzw. Anhängeoperationen muss getestet werden, ob der Zielpuffer groß genug ist.
- **0-Byte-Terminierung**
  Beim Kopieren und Bearbeiten von Strings muss sichergestellt werden, dass der String mit einem 0-Byte abgeschlossen ist.
- **Erst Typecast zu unsigned char, dann zu größeren Datentypen**
  Ansonsten entstehen fehlerhafte Ergebnisse.

Empfehlungen

- **Ganzzahlwerte gegenüber Strings bevorzugen**
  z. B. in Datumsangaben den Monat bevorzugt als Zahlenwert verwenden, nicht als Text.
- **strncpy() und strncat() vermeiden**
  - String wird abgeschnitten, falls zu lang.
  - Fehlende 0-Byte-Terminierung, wenn String zu lang.

Klassifikation von Zeichen

```
#include <ctype.h>

int
isascii(int c);
```

- prüft, ob das Zeichen c ein 7-Bit-Zeichen aus dem ASCII-Zeichensatz ist und gibt entsprechenden „wahr“- oder „falsch“-Wert zurück.
- char-Zeichen müssen **zunächst nach unsigned char konvertiert** werden, bevor eine weitere – evtl. implizite – Umwandlung zu int erfolgt:

```
char c;
c = ...;
if ( isascii((unsigned char)c) ) {
  /* Zeichen ist aus ASCII-Zeichensatz */
}
```

- Makros der nächsten Folie müssen immer mit *isascii()* UND-verknüpft werden, nur dann wird richtiges Ergebnis geliefert.

Klassifikation von Zeichen

Makro	Test
isalpha	Buchstabe aus Alphabet
islower	Kleinbuchstabe
isupper	Großbuchstabe
isalnum	Alphanumerisches Zeichen (Buchstabe oder Zahl)
isdigit	Dezimalziffer
isxdigit	Hexadezimalziffer
isblank	Leerzeichen oder Tabulator
isspace	Whitespace: Leerzeichen, Form-Feed, Newline, Zeilenrücklauf, Tabulator oder Vertikal-Tabulator
iscntrl	Control-Zeichen
isgraph	Druckbares Zeichen, jedoch kein Leerzeichen
isprint	Druckbares Zeichen, einschl. Leerzeichen
ispunct	Druckbar, jedoch weder Leerzeichen noch alphanumerisch

typedef

typedef *VorhandenerTyp NeuerName*;

```
typedef long KOORDINATE;

KOORDINATE a;

a = 5L;
```

- Kann Übersichtlichkeit erhöhen.
- Meist in Zusammenhang mit struct, union und enum benutzt.

enum

```
enum Name {
  Option [= Zahlenwert], ...
};
```

```
/** Fuellstand eines Trinkbechers
*/
enum TRINKBECHER {
  /** Zu wenig drin, weiter fuellen. */
  BECHER_LEER = 0,

  /** Fuellzustand im akzeptablen Bereich. */
  BECHER_OK ,

  /** Becher voll, nicht weiter fuellen sonst Ueberlauf. */
  BECHER_VOLL
};
```

Hinweise für enum

- **Bildung der Zahlenwerte:**
  - Zahlenwert für erstes Element beginnt mit 0, wenn kein Wert festgelegt wurde.
  - Zahlenwert für andere Elemente ist jeweils Wert des Vorgängers um 1 erhöht, wenn kein Wert festgelegt wurde.
- **Zahlenwerte müssen eindeutig sein, deshalb:**
  - gar keine Zahlenwerte angeben (Werte beginnen mit 0 und erhöhen sich fortlaufend) oder
  - nur Wert für erstes Element angeben (Werte für andere Elemente erhöhen sich fortlaufend) oder
  - Werte für alle Werte angeben und dabei auf Eindeutigkeit achten!

struct

```
struct Name {
  Typ1  Variablenname1;
  Typ2  Variablenname2;
  ...
};
```

```
struct artikelpreis {
  unsigned long  artikelnr;
  unsigned long  preis_euro;
  unsigned short preis_cent;
};

struct artikelpreis ap1;

ap1.artikelnr = 1234UL;
ap1.preis_euro = 2;
ap1.preis_cent = 99;
```

- Alle Komponenten gleichzeitig verfügbar.
- Empfehlung: Komponenten in absteigender Größe anordnen.

Beispiel für Initialisierung eines struct

```
typedef struct {
  double x;                    /* zuerst x-Wert */
  double y;                    /* danach y-Wert */
} PUNKT;

PUNKT a = { 1.0, 2.0 };    /* zuerst x-Wert, danach y-Wert */
```

- Initialisierungswerte in geschweiften Klammern
- Werte in derselben Reihenfolge wie in struct-Definition

Beispiel für Initialisierung eines verschachtelten struct

```
typedef struct {
  double x;
  double y;
} PUNKT;

                                        /* Achsenparalleles Rechteck ist */
typedef struct {                        /* definiert durch 2 Punkte: */
  PUNKT lu;                             /* links unten */
  PUNKT ro;                             /* rechts oben */
} RECHTECK;

RECHTECK re = { { 1.0, 2.0 }, { 3.0, 4.0 } };   /* erst lu, danach ro */
```

- Verschachtelung der geschweiften Klammern entsprechend der Verschachtelung der Typdefinitionen

union

```
union Name {
  Typ1 Variablenname1;
  Typ2 Variablenname2;
  ...
};
```

```
/* ... Definition der Datentypen rechteck, dreieck, kreis ... */
union geometrieobject {
  struct rechteck r;
  struct dreieck d;
  kreis k;
};
```

- Alle Komponenten teilen sich den Speicher.
- Nur eine Komponente wirklich vorhanden (die zuletzt geschriebene)!
- Empfehlung: union wird meist mit einem int-Wert zu einem struct kombiniert, der int-Wert gibt an, welche Komponente des union belegt ist.

Ausgabe mit printf()

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

```
int i;

i = ...;

printf("i hat den Wert: %d\n", i);
```

- printf für „print formatted“.
- format-String kombiniert:
  - konstante Textbausteine und
  - Platzhalter für auszugebende Werte.
- Reihenfolge der Platzhalter muss Reihenfolge der Werte entsprechen!

Platzhalter für printf()

- Prozentzeichen (erforderlich, markiert Beginn eines Platzhalters)
- Flags (optional)
- Breite (optional)
- Präzision (optional, durch Dezimalpunkt von Breite abgetrennt)
- Größenmodifizierer (optional)
- Datentyp (erforderlich)

Platzhalter für printf()

- Flags

-	Text linksbündig in zur Verfügung stehenden Platz setzen
0	Numerische Werte mit führenden Nullen versehen, um Platz voll zu nutzen
+	Vorzeichen auch für nichtnegative Werte ausgeben
Leerzeichen	Leerstelle anstelle eines Vorzeichens für nichtnegative Werte
#	Markierung für Hexadezimal- und Oktalwerte schreiben, für Gleitkommazahlen Dezimalpunkt erzwingen

- Breite

  Dezimalzahl, wieviele Zeichen soll die Ausgabe enthalten
- Präzision
  - für Strings: Anzahl der Zeichen
  - für ganze Zahlen: minimale Ziffernanzahl
  - für Gleitkommazahlen: Signifikante Stellen bzw. Nachkommastellen

Datentypen und Größenmodifizierer im Platzhalter für printf()

Datentypen und ausgewählte Größenmodifzierer:

c	char (einzelnes Zeichen)	
s	String (0-Byte-terminierte Zeichenkette)	
d	int (Ausgabe dezimal)	hd für short, ld für long
i	int (Ausgabe dezimal)	hi für short, li für long
u	unsigned (Ausgabe dezimal)	hu, lu
o	unsigned (Ausbabe oktal)	ho, lo
x	unsigned (Ausgabe hexadezimal, „a“...„f“)	hx, lx
X	unsigned (Ausgabe hexadezimal, „A“...„F“)	hX, lX
e	float (Ausgabe immer mit Exponent, mit „e“ abgetrennt)	le für double
E	float (Ausgabe immer mit Exponent, mit „E“ abgetrennt)	le für double
f	float (Ausgabe ohne Exponenten)	lf für double
g	float (Exponent nur, falls nötig, abgetrennt mit „e“)	lg für double
G	float (Exponent nur, falls nötig, abgetrennt mit „E“)	lg für double

Eingabe mit scanf()

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

```
int i = 0; /* Variable */
int r = 0; /* Ergebnis von scanf() */

r = scanf("%d", &i);
if (0 < r) {
  /* Erfolgreich eingelesen */
}
```

- Format-String enthält Platzhalter für einzulesende Werte.
- Darauf folgen **Adressen** der Variablen, in die die Werte eingelesen werden.
- Reihenfolge der Platzhalter muss den Datentypen der Adressen entsprechen.
- Ergebnis: Anzahl der erfolgreich eingelesenen Werte.
- Nutzung von scanf() nicht empfohlen, besser mit fgets() zeilenweise Text einlesen und dann mit sscanf() Daten extrahieren.

Vorzeichenlose ganze Zahlen

- Bitweise Darstellung im Dualsystem
- Unterschiedliche Bitanzahl, je nach Prozessor und Betriebssystem:
  - unsigned char: 8 Bit
  - unsigned short: 16 Bit
  - unsigned: mindestens 16 Bit
  - unsigned long: mindestens 32 Bit
  - unsigned long long: mindestens 64 Bit
  - size_t:
    groß genug, um gesamten Hauptspeicher zu durchmustern
  - uintmax_t:
    groß genug, um jeden anderen vorzeichenlosen Typ verlustfrei zu uintmax_t konvertieren zu können
- Bei einer Breite von n Bits Wertebereich $0 \ldots 2^n - 1$

Vorzeichenbehaftete ganze Zahlen

- Negative Zahlen im Zweierkomplement dargestellt
  (alle Bits invertieren, anschließend „1“ addieren)
- Unterschiedliche Bitanzahl, je nach Prozessor und Betriebssystem: signed char (8), short (16), int (mind. 16), long (mind. 32), long long (mind. 64), intmax_t (groß genug für verlustfreie Konvertierung aller anderen vorzeichenbehafteten Typen).
- Bei einer Breite on n Bits Wertebereich $-2^{n-1} \ldots 2^{n-1} - 1$

Probleme bei ganzzahligen Berechnungen

- **Division durch 0**
  - führt zu sofortigem Programmabbruch.
  - **Vor jeder Division sicherstellen, dass Divisor ungleich 0!**
- **Überlauf**
  - ist Abschneiden führender Bits, wenn Ergebnis zu groß für gewählten Datentyp.
  - Führt zu falschen Ergebnissen.
- **Vergleich von vorzeichenbehafteter und vorzeichenloser ganzer Zahl**
  - Vorzeichenbehaftete Zahl wird in vorzeichenlose Zahl umgewandelt.
  - Negative vorzeichenbehaftete Zahlen ergeben dabei u. U. sehr große vorzeichenlose Zahlen.
  - Fehlerhaftes Vergleichsergebnis.

Beispiel für Überlauf

```
unsigned short a, b, c;
a = 65535;              /*  1111111111111111_2*/
b = 65535;              /*  1111111111111111_2*/
c = a + b;              /* 1111111111111110_2*/
                        /* Da Datentyp 16 Bit breit ist: */
/* c = 65534 */         /*  1111111111111110_2*/
```

Überlauferkennung (Fehlerhaftes Beispiel)

Ungleichungen für das Auftreten des Überlaufes müssen so umgestellt werden, dass bei den Berechnungen und Tests nicht wiederum ein Überlauf auftreten kann.

Falsch:

```
unsigned short a, b, c;
a = ...;                    /* z.B. 65535 */
b = ...;                    /* z.B. 65535 */
if (a + b <= 65535) {       /* a + b -> 65534 (Ueberlauf bei Berechnung a+b) */
  c = a + b;                /* Bedingung immer erfuellt, Ueberlauf wird nie erkannt! */
} else {
  /* Ueberlauf! */
}
```

Überlauferkennung (Funktionierendes Beispiel)

Umstellung $a + b \leq MAX$ zu $a \leq MAX - b$:

```
unsigned short a, b, c;
a = ...;
b = ...;
if (a <= 65535 - b) {
  c = a + b;
} else {
  /* Ueberlauf! */
}
```

Da $0 \leq b \leq MAX$ gilt auch wieder $0 \leq MAX - b \leq MAX$, somit tritt bei der Berechnung kein Überlauf auf und der Test liefert das richtige Ergebnis.

Fehlererkennung bei Operationen mit vorzeichenlosen ganzen Zahlen

Fehler	Operation	Bedingung	Test in Programm
Überlauf	Addition $a + b$	$a + b > MAX$	$a > MAX - b$
Überlauf	Subtraktion $a - b$	$b > a$	$b > a$
Überlauf	Multiplikation $a \cdot b$	$a \cdot b > MAX$	nur wenn $b > 0$: $a > \frac{MAX}{b}$
Division durch 0	Division $\frac{a}{b}$	$b = 0$	$b = 0$

Für $MAX = 2^n - 1$ sind Konstanten in limits.h und stdint.h definiert:

Datentyp	Konstante für Maximalwert
unsigned char	UCHAR_MAX
unsigned short	USHRT_MAX
unsigned	UINT_MAX
unsigned long	ULONG_MAX
uintmax_t	UINTMAX_MAX
size_t	SIZE_MAX

Fehlererkennung bei Operationen mit vorzeichenbehafteten ganzen Zahlen

Für $MIN = -2^{n-1}$ und $MAX = 2^{n-1} - 1$ sind Konstanten in limits.h und stdint.h definiert:

Datentyp	Konstante für Minimalwert	Konstante für Maximalwert
signed char	SCHAR_MIN	SCHAR_MAX
short	SHRT_MIN	SHRT_MAX
int	INT_MIN	INT_MAX
long	LONG_MIN	LONG_MAX
intmax_t	INTMAX_MIN	INTMAX_MAX

- Unsymmetrie von positivem und negativem Wertebereich
- dadurch deutlich höherer Aufwand für Überlauftests
- Überlauf tritt auch bei Betragsbildung und Division auf
- In Überlauftests muss Subtraktion ...-MIN vermieden werden
- siehe Datei fe-long.c für Beispielquelltext bzw. ausführliches Vorlesungsscript

Vergleich zwischen vorzeichenloser und vorzeichenbehafteter Zahl

Vergleich zwischen vorzeichenloser Zahl u und vorzeichenbehafteter Zahl s in zwei Schritten:

- Wenn $s < 0$, so gilt automatisch $s < u$.
- Andernfalls kann s mit u verglichen werden.

Hinweise, Empfehlungen (1)

- **Division durch 0 vermeiden!** Sowohl bei Division mit / als auch bei Modulo-Division (Divisionsrest) mit % darf Divisor nicht 0 sein.
- **Überlauftest vor ganzzahligen Operationen.**
- **Überlauftest vor Konvertierung/Casten zu kleineren Datentypen.**
- **Negative Divisionsreste verarbeiten.**
  Divisionsreste aus Modulo-Division können negativ sein.
- **Für Größenangaben size_t verwenden.**
  (Speicheranforderungen, Indizes, Längen, Schleifenzähler...).
- **Vergleich in zwei Schritten**
  bei Vergleich von vorzeichenbehafteter und vorzeichenloser Zahl.
- **Immer entweder signed char oder unsigned char**
  für 8-Bit-Ganzzahlen verwenden, nicht einfach char.

Hinweise, Empfehlungen (2)

- **Bitweise Operatoren nur für vorzeichenlose Operanden**
  verwenden, speziell die Bitschiebe-Operatoren.
- **Bitschiebe-Operatoren nicht mehr zur Optimierung**
  von Multiplikation oder Division mit Zweierpotenzen verwenden
  (Compiler optimieren dies normalerweise selbst).
- **Bitschiebe-Operatoren nur mit positiver Verschiebungsweite**
  verwenden, nicht größer als Bitanzahl des Operanden.
- **Ganzzahl-Berechnungen in ausreichender Bitgröße durchführen.**
  Einer der Operanden – bevorzugt der erste – wird hierzu in einen ausreichend großen Datentyp gecastet.

Speicherung von Gleitkommazahlen

- **Vorzeichenfeld** (VZ), 1 Bit
  0 für nichtnegative Zahlen, 1 für negative Zahlen
- **Exponentenfeld** (EF), allgemeine Länge LEF Bits
  Exponent e zur Basis 2 mit hinzugefügter Verschiebung ($BIAS = 2^{LEF-1} - 1$). Hinzufügen des BIAS ermöglicht Darstellung auch negativer Exponenten im vorzeichenlosen Exponentenfeld.
- **Mantissenfeld** (MF), allgemeine Länge LMF Bits
  Nachkommateil der Mantisse. In normalisierter Darstellung steht eine 1 vor dem Komma, in nichtnormalisierter Darstellung (Exponentenfeld hat den Wert 0) steht eine führende 0 vor dem Komma.

Sonderfälle von Gleitkommazahlen

- **INF**
  Unendlich (positiv, negativ)
  Alle Bits des Exponentenfeldes sind auf 1 gesetzt, alle Bits des Mantissenfeldes auf 0.
- **NAN**
  Not a number
  Alle Bits des Exponentenfeldes sind auf 1 gesetzt, das Mantissenfeld ist ungleich 0.
  - **signalisierende NAN**
    Höchstwertiges Mantissenbit ist 0.
    Ausnahmebehandlung (Trap) wird ausgelöst, wenn Zahl in Berechnungen als Operand benutzt wird.
  - **stille NAN**
    Höchstwertiges Mantissenbit ist 1.
    Kein Auslösen eines Traps.

  Restliche Mantissenbits können zusätzliche Informationen enthalten, nicht normiert.

Darstellungsformen

- $EF \neq 0 \ \wedge \ EF \neq 2^{LEF} - 1 \rightarrow$ Normalisierte Darstellung
  Wert $= (-1)^{\text{VZ}} \cdot 1, \text{MF} \cdot 2^{\text{EF}-\text{BIAS}}$
  Normalerweise benutzte Darstellung.
- $EF = 0 \ \rightarrow$ Nichtnormalisierte Darstellung
  Wert $= (-1)^{\text{VZ}} \cdot 0, \text{MF} \cdot 2^{1-\text{BIAS}}$
  Nur für sehr kleine Werte benutzt, aufgrund der geringeren Anzahl signifikanter Bits deutliche ungenauer als normalisierte Darstellung.

Wertebereiche

	float	double	long double
Länge EF	8	11	15
Länge MF	23	52	63
Größte darstellbare Zahl	$\frac{2^{127}\cdot(2^{24}-1)}{2^{23}}$ $\approx 1,7 \cdot 10^{38}$	$\frac{2^{1023}\cdot(2^{53}-1)}{2^{52}}$ $\approx 1,8 \cdot 10^{308}$	$\frac{2^{16383}\cdot(2^{64}-1)}{2^{63}}$ $\approx 1,19 \cdot 10^{4932}$
Kleinste normalisiert darstellbare Zahl	2^{-126} $\approx 1,18 \cdot 10^{-38}$	2^{-1022} $\approx 2,23 \cdot 10^{-308}$	2^{-16382}
Kleinste nicht normalisert darstellbare Zahl	2^{-149} $\approx 1,4 \cdot 10^{-45}$	2^{-1074} $\approx 4,94 \cdot 10^{-324}$	2^{-16445}

Darstellung der long double hier für eine Länge von 10 Byte, es existieren auch Varianten mit 12 oder 16 Byte.

Test auf benutzbare Ergebniss (Einzelwert)

Bedingung	Windows	POSIX
Weder INF noch NAN:	int _finite(double x);	int isfinite(x); (Makro)
Normalisierte Darstellung:	int _fpclass(double x); Ergebniswerte _FPCLASS_ND bzw. _FPCLASS_PD für neg. bzw. pos. nicht normalisierte Werte	int isnormal(double x); (Makro)

Floating Point Exceptions (Exception-Bits)

- Setzen von Bits in einem Statuswort
- Nach größerer Rechnung Abfrage der Exception-Bits anstelle Test jedes Ergebnisses und jedes Zwischenwertes

Ausnahmebedingung	**Windows**	**POSIX**
Division durch 0	_SW_ZERODIVIDE	FE_DIVBYZERO
Unbestimmtes Ergebnis z. B. $\frac{\infty}{\infty}$ oder $\infty - \infty$	_SW_INVALID	FE_INVALID
Überlauf (Ergebnis zu groß)	_SW_OVERFLOW	FE_OVERFLOW
Unterlauf nicht normalisiertes Ergebnis	_SW_UNDERFLOW	FE_UNDERFLOW
Ungenaues Ergebnis Abschneiden von hinteren Mantissenbits	_SW_INEXACT	FE_INEXACT
Denormalisiertes Ergebnis	_SW_DENORMAL	

Floating Point Exceptions (Anmerkungen)

- **Division durch 0**
  Eine Division durch 0 oder einen anderen sehr kleinen Wert ist aufgetreten.
- **Unbestimmtes Ergebnis**
  Mathematisch unbestimmter Ausdruck, Wurzel aus negativer Zahl...
- **Überlauf**
  Das Ergebnis einer Operation war größer als die größte darstellbare Zahl.
- **Unterlauf**
  Bei einer Operation mit zwei normalisierten Operanden war das Ergebnis eine nicht normalisiert darstellbare Zahl (zu klein).
- **Ungenaues Ergebnis**
  Das Ergebnis hatte mehr Nachkommabits, als im Mantissenfeld gespeichert werden können (z. B. unendlicher Bruch).
  Diese Ausnahme tritt relativ häufig auf.

Floating Point Exceptions (Abfrage unter Windows)

- Vor Berechnung:
  (void)_clearfp();
- Berechnung ausführen.
- Nach Berechnung:
  unsigned u;

  ```
  u = _clearfp();
  if (_SW_ZERODIVIDE & u) {
    /* Division durch 0 aufgetreten */
  }
  ...
  ```
- Testen auf _SW_ZERODIVIDE, _SW_INVALID, _SW_OVERFLOW, _SW_UNDERFLOW, optional auch auf _SW_DENORMAL.
- Im Normalfall kein Test auf _SW_INEXACT, da dies recht häufig auftritt.

Floating Point Exceptions (Abfrage unter POSIX)

```
#include <fenv.h>
#include <float.h>
#include <math.h>
...
int exc = 0;
...
feclearexcept(FE_ALL_EXCEPT);
/* Berechnung */
exc = fetestexcept(FE_ALL_EXCEPT & (~(FE_INEXACT)));
feclearexcept(FE_ALL_EXCEPT);
if (exc & FE_DIVBYZERO) {
  /* Division durch 0 */
}
/* ... Test auf weitere Ausnahmebedingungen ... */
```

Mathematik-Bibliothek (Übersicht)

- Konstanten-Definitionen und Funktionen.
- Header-Datei math.h muss mit eingeschlossen werden.
- Unter Windows:
  #define _USE_MATH_DEFINES 1
  #include <math.h>
- Bibliothek muss beim Linken mit verwendet werden.
  - Windows mit Visual Studio: Automatisch
  - Andere Systeme: Option „-lm“ für Linker mit angeben.

Mathematik-Bibliothek / Konstanten

M_E	e (Eulersche Zahl)
M_PI	π
M_PI_2	$\frac{\pi}{2}$
M_PI_4	$\frac{\pi}{4}$
M_SQRT2	$\sqrt{2}$

Mathematik-Bibliothek / Funktionen (Vergleich, Betrag, Runden)

Vergleichen

isgreater(x, y)	$x > y$
isgreaterequal(x, y)	$x \geq y$
isless(x, y)	$x < y$
islessequal(x, y)	$x \leq y$
islessgreater(x, y)	$x \neq y$
isunordered(x, y)	Nicht vergleichbar, mind. 1 NAN

Vergleichsbedingungen sind nicht erfüllt, wenn einer der beteiligten Operanden INF oder NAN ist.

Betragsbildung, Runden

double fabs(x)	$\|x\|$
double floor(x)	Abrunden zur nächst-kleineren ganzen Zahl
double ceil(x)	Aufrunden zur nächst-größeren ganzen Zahl
double rint(x)	Runden zur nächst-gelegenen ganzen Zahl

Mathematik-Bibliothek / Funktionen (Winkelfunktionen, Potenzen, Logarithmen, hyperbolisch)

Alle Argumente vom Typ double, alle Ergebnisse auch.

sin(x) cos(x) tan(x)	Winkel- funktionen, Radiant
asin(x) acos(x) atan(x) atan2(y, x)	Umkehr- funktionen richtiger Quadrant
sqrt(x)	$\sqrt{x}$
exp(x)	e^x
log(x)	$\ln x = \log_e x$
log10(x)	$\log_{10} x$
pow(x, y)	x^y

cosh(x) sinh(x) tanh(x)	Hyperbolische Funktionen
acosh(x) asinh(x) atanh(x)	Umkehr- funktionen

Empfehlungen

- **Gleitkommazahlen nicht als Zähler in Schleifen**
  Für Schleifendurchläufe immer ganzzahlige Werte benutzen!
- **Wertebereiche beachten bei Konvertierung**
  - **Gleitkomma- zu Ganzzahl**
    - Genauigkeitsverlust durch Abschneiden der Nachkommastellen.
    - Wertebereich von Ganzzahlen wesentlich kleiner.
  - **Ganz- zu Gleitkommazahl**
    - Genauigkeitsverlust möglich, Anzahl Mantissenbits eines double kann kleiner sein als Bitbreite der ganzen Zahl.

Gründe für Aufteiltung der Quelltexte auf mehrere Module

- **Übersichtlichkeit**
  Kleinere Dateien in Texteditor einfacher zu bearbeiten als sehr große.
- **Arbeit im Team**
  An mehreren Quelldateien können mehrere Entwickler gleichzeitig arbeiten.
- **Wiederverwendung**
  Kopieren kompletter Dateien in neues Projekt ist einfacher als Suchen und Kopieren von Textstellen in Dateien.

Header-Dateien, prinzipieller Aufbau

#ifndef *DATEINAME*_H_INCLUDED
#define *DATEINAME*_H_INCLUDED 1

Datentyp-Definitionen und Konstanten-Definitionen (falls vorhanden)

#ifdef __cplusplus
extern "C" {
#endif

Funktions-Prototypen

#ifdef __cplusplus
}
#endif

#endif
/* ifndef *DATEINAME*_H_INCLUDED */

Speicherklassen von Variablen (Sichtbarkeit, Lebensdauer)

- **automatic** (ohne explizite Speicherklasse) in Funktion
  Lokale Variable der Funktion, nur in Funktion nutzbar.
  Lebensdauer: Abarbeitung der Funktion.
- **automatic** (ohne explizte Speicherklasse) auf Dateiebene
  Globale Variable, von allen nachfolgenden Funktionen aus nutzbar.
  Mit Angabe „extern“ auch aus anderen Modulen heraus nutzbar.
  Lebensdauer: Programmlaufzeit.
- **static** in Funktion
  Lokale Variable der Funktion, nur in Funktion nutzbar.
  Lebensdauer: Programmlaufzeit.
  Variable behält zwischen verschiedenen Funktionsaufrufen ihren Wert bei.
- **static** auf Dateiebene
  modulinterne Variable, in allen nachfolgenden Funktionen nutzbar.
  Kein Zugriff aus anderen Modulen heraus möglich.
  Lebensdauer: Programmlaufzeit.

Speicherklassen von Variablen (Sichtbarkeit, Lebensdauer)

- **extern**
  Zugriff auf globale Variable, die in anderen Modulen definiert sind.
- **register**
  Anweisung an Compiler, Variablenwert möglichst durchgängig in Prozessor-Register zu behalten.
  Anzahl der Prozessor-Register ist begrenzt!
  Moderne Compiler optimieren Register-Nutzung automatisch.

Empfehlungen (1)

- **Eindeutige Namen für Headerdateien**
  Namen von anderen Headerdateien oder systemweiten Headerdateien nicht überdecken!
- **Kleinstmöglichen Sichtbarkeitsbereich für Variablen**
  Variablen und Funktionen sollten immer den kleinstmöglichen Sichtbarkeitsbereich haben. Modulinterne Variablen und Funktionen erhalten Speicherklasse „static“!
- **Keine Überdeckung von Variablen- und Funktionsnamen**
  In untergeordneten Sichtbarkeitsbereichen keine Variablen anlegen, die gleichnamige Variablen aus äußeren Sichtbarkeitsbereichen überdecken!
- **Funktionsnamen lassen Rückschluss auf Dateinamen zu**
  Am Anfang von Funktionsnamen steht der Dateiname, in dem die Funktion definiert ist oder eine Kennung, anhand derer sich der Dateiname schnell erschließt.

Empfehlungen (2)

- **Header-Name = Modul-Name**
  Die Headerdatei für das Module abcxyz.c heißt abcxyz.h.
- **Schutzkonstante entsprechend Dateinamen**
  z. B. für Datei abcxyz.h:

```
#ifndef ABCXYZ_H_INCLUDED
#define ABCXYZ_H_INCLUDED 1

/* Inhalt der Header-Datei */

#endif
/* ifndef ABCXYZ_H_INCLUDED */
```

Präprozessor

- **Vorverarbeitung** vor eigentlicher Compilierung.
- **Präprozessorzeilen beginnen mit Raute #**

Anweisung	**Zweck**
#include	Dateiinhalt einschließen
#define	Konstanten und Makros definieren
#if, #else, #endif	Bedingte Compilierung
#error	Compiler-Fehlermeldung veranlassen
#pragma	Compiler-abhängige Anweisungen (hier nicht behandelt)

#include

- #include <datei.h>
  Suche nur in Include-Pfad, z.B.:
  #include <stdio.h>
- #include "datei.h"
  Suche zunächst im aktuellen Verzeichnis, dann im Include-Pfad, z.B.:
  #include "fe-geom.h"

Include-Pfad: System- und compilerspezifische Verzeichnisse mit Include-Dateien

Konstante: #define *Bezeichner Ersetzungstext*

- hauptsächlich für Konstanten-Definitionen, z.B.:
  #define ZEHN 10
- Alle Vorkommen von *Bezeichner* werden durch den *Ersetzungstext* ersetzt.
- Beispiel:

```
#define _USE_MATH_DEFINES 1
#include <math.h>
...
A = r * r * M_PI;
```

anstelle von

```
A = r * r * 3.14159265358979323846;
```

Makro #define *Name*(*Argument*,...) *Ersetzungstext*

```
#define QUADRAT(x) x * x

a = QUADRAT(b);
u = QUADRAT(3);
```

Es wird *reine Text-Ersetzung* durchgeführt.

```
a = b * b;
u = 3 * 3;
```

Stolperstein 1: Seiteneffekte

```
#define QUADRAT(x) x * x
a = QUADRAT(i++)
```

Text-Ersetzung von „x“ durch „i++“ führt zu

```
a = i++ * i++;
```

Seiteneffekte wie „++“ und „--“ sind unverträglich mit Makros, insbesondere wenn die Makro-Argumente mehrfach im Ersetzunstext erscheinen!

Stolperstein 2: Fehlende Klammern um Makro-Parameter im Ersetzungstext

- *Falsch:*

  #define QUADRAT(x) x * x

  y = QUADRAT(2+3);

- Liefert Ergebnis:

  y = 2 + 3 * 2 + 3;

- *Besser:*

  #define QUADRAT(x) (x) * (x)

  führt zu

  y = (2 + 3) * (2 + 3);

Stolperstein 3: Fehlende Klammern um gesamten Ersetzungstext

- *Falsch:*

  ```
  #define QUADRAT(x) (x) * (x)
  i = 100 / QUADRAT(2+3);
  ```

- Liefert Ergebnis:

  ```
  i = 100 / (2+3) * (2+3); /* Erst Division 100 durch 5, dann Multiplikation */
  ```

- *Besser:*

  ```
  #define QUADRAT(x) ((x) * (x))
  ```

  führt zu

  ```
  i = 100 / ((2+3) * (2+3));
  ```

Stolperstein 4: Mehrere Anweisungen

- *Falsch:*

  #define SWAP(x,y) tmp = x; x = y; y = tmp;
- if (a < b) SWAP(a,b);
- Liefert Ergebnis:

  if (a < b) tmp = a; a = b; b = tmp; /* Anweisungen 2 und 3 immer ausgeführt */
- *Besser:*

  #define SWAP(x,y) do { tmp = x; x = y; y = tmp; } while(0)

  führt zu:

  if (a < b) do { tmp = a; a = b; b = tmp; } while(0);

Zusammenfassung Makros

- Makros *nach Möglichkeit vermeiden*, besser Funktionen verwenden!
- *Berechnungsterme*
  - Klammern um Parameter im Ersetzungstext und
  - Klammern um gesamten Ersetzungstext!

  #define QUADRAT(x) ((x) * (x))
- *Mehrere Anweisungen*

  do { ... } while(0)

  einschließen!

Bedingte Compilierung - Ist Konstante definiert

#ifdef KONSTANTE

Quelltext hier wird compiliert, wenn Konstante definiert ist.

#else

Quelltext hier wird compiliert, wenn Konstante nicht definiert ist.

#endif

Bedingte Compilierung - Ist Konstante definiert und ungleich 0

#if AUSDRUCK

Quelltext hier wird compiliert, wenn Ausdruck definiert und ungleich 0 ist.

#else

Quelltext hier wird compiliert, wenn Ausdruck nicht definiert oder 0 ist.

#endif

Zeitweilige Deaktivierung von Quelltext-Teilen

```
1  #include <stdio.h
2  #include <stdlib.h>
3  #ifdef _WIN32
4  #include <process.h>
5  #else
6  #include <unistd.h>
7  #endif
8
9  int main(int argc, char *argv)
10 {
11 #if 0
12    printf("Hello world.\n");
13 #endif
14    exit(EXIT_SUCCESS); return EXIT_SUCCESS;
15 }
```

Zum Aktivieren die 0 durch eine 1 ersetzen.

Beispiel für bedingte Compilierung

String-Vergleich ohne Unterscheidung Groß- und Kleinschreibung: Verwendung von *stricmp()* unter Windows, *strcasecmp()* auf anderen Systemen.

```
#ifdef _WIN32
if (0 == stricmp(str1, str2))
#else
if (0 == strcasecmp(str1, str2))
#endif
{
   printf("Die Strings sind gleich.\n");
}
```

Beispiel für Compiler-Fehlermeldung

```
1  #ifdef _WIN32
2
3  if (0 == stricmp(str1, str2))
4
5  #else
6
7  #if HAVE_STRCASECMP
8  if (0 == strcasecmp(str1, str2))
9  #else
10 #error Weder stricmp() noch strcasecmp() verfuegbar!
11 #endif
12
13 #endif
14 {
15    printf("Die␣Strings␣sind␣gleich.\n");
16 }
```

Kommandozeilenargumente

```
1 int main(int argc, char *argv[])
2 {
3     ...
4 }
```

Aufruf:

testprog Dies ist ein Test

argc:	5 (Gesamtzahl enschließlich Programmname)
argv[0]	"testprog"
argv[1]	"Dies"
argv[2]	"ist"
argv[3]	"ein"
argv[4]	"Test"

Rückgabewert

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #ifdef _WIN32
4  #include <process.h>
5  #else
6  #include <unistd.h>
7  #endif
8
9  int main(int argc, char *argv)
10 {
11    printf("Hello world.\n");
12    exit(EXIT_SUCCESS); return EXIT_SUCCESS; /* Fehler: EXIT_FAILURE */
13 }
```

Dateien lesen und schreiben

- **Binärdateien**
  - *Form wie im Hauptspeicher*
  - Vorteil: Kleine Dateigröße
  - Vorteil: Keine Umwandlung erforderlich
  - Nachteil: Keine Bearbeitung in Text-Editor
- **Textdateien**
  - *Umwandlung in Text*
  - Nachteil: Dateigröße
  - Nachteil: Umwandlung erforderlich (Programmier- u. Zeitaufwand)
  - Vorteil: Bearbeitung in Text-Editor möglich
  - Unter Windows: Umsetzung Newline zu Newline + Carriage return

Filedeskriptor, FILE

- **Filedeskriptoren**
  - Systemweit eindeutiger int-Wert für jede offene Datei
  - Betriebssystem verknüpft Resourcen mit Wert
  - Ausschließlich Lesen und Schreiben von Byte-Folgen
  - Meist keine Pufferung, d.h. Datenträgerzugriff für jeden E/A-Vorgang
  - API auch für Netzwerkzugriff, Inter-Prozess-Kommunikation...
- **FILE**
  - Baut auf Filedeskriptoren auf
  - Höheres Abstraktionslevel
  - Text- und Binärdateien
  - Interne Pufferung

Beispiel für Verwendung von FILE

```c
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE          *fipo    =       NULL;    /* Schreiben der Datei */
  double         erg     =       1.0;     /* Berechnungsergebnis */

  fipo = fopen("C:\\Temp\\datei.txt", "w");
  if (NULL != fipo) {
    fprintf(fipo, "Das Ergebnis ist: %lg\n", erg);
    fclose(fipo);
  } else {
    /* !!! FEHLER: Datei konnte nicht geoeffnet werden */
  }
  exit (0); return 0;
}
```

Öffnen und Schließen von Dateien

- **FILE *fopen(const char *path, const char *mode);**
  - öffnet Datei
  - gibt bei Erfolg Zeiger auf FILE zurück, bei Fehler NULL-Zeiger
  - path ist Dateiname
  - mode ist normalerweise eines von:
    - "r" Text-Datei lesen
    - "w" Text-Datei schreiben
    - "rb" Binär-Datei lesen
    - "wb" Binär-Datei schreiben
- **int fclose(FILE *fp);**
  - schließt Datei
  - gibt bei Erfolg 0 zurück, bei Fehler EOF

Funktionen für Text-Dateien

- **int fputc(int c, FILE *stream);**
  Einzelnen Buchstaben schreiben
- **int fputs(const char *s, FILE *stream);**
  String schreiben
- **int fprintf(FILE *stream, const char *format, ...);**
  Formatierte Ausgabe
- **int fgetc(FILE *stream);**
  Einzelnes Zeichen lesen
- **char *fgets(char *s, int size, FILE *stream);**
  Text-Zeile lesen
- **int feof(FILE *stream)**
  Text auf Dateiende (0, wenn Dateiende noch nicht erreicht)

Funktionen für Binär-Dateien

- **size_t fwrite(const void *ptr, size_t elsize, size_t nmemb, FILE *stream);**
  Binärdaten schreiben
- **size_t fread(void *ptr, size_t elsize, size_t nmemb, FILE *stream);**
  Binärdaten lesen

Dynamische Speicherverwaltung

- Zweck: Anforderung von Speicher exakt in der benötigten Größe.
- Funktionen:
  - **void *malloc(size_t size);**
    Anforderung von Speicher, nicht initialisiert
  - **void *calloc(size_t nmemb, size_t elsize);**
    Anforderung und Initialisierung (mit 0-Bytes beschrieben)
  - **void free(void *ptr);**
    Freigabe nach Benutzung
- size, nmemb und elsize
  - nicht 0
  - Kein Überlauf für Produkt nmemb·elsize
- Nach Anforderung Test auf Erfolg (Zeiger ungleich NULL)
- Nach Benutzung freigeben

Hinweise (1)

- **Keine großen Objekte auf dem Stack ablegen!**
  Stackbereich ist begrenzt, besser dynamisch Speicher allozieren!
  Vermeiden:
  - Große Argumente oder Variable in Funktionen,
  - Felder variabler Größe ohne Größenbegrenzung und
  - rekursive Funktionen ohne Begrenzung der Rekursionstiefe.

Hinweise (2)

- **Größen ungleich 0**
  Die Größe für malloc bzw. Elementgröße und -anzahl für calloc dürfen nicht 0 sein!
- **Kein Überlauf**
  Das Produkt aus Elementgröße und -anzahl für calloc darf nicht zu einem Überlauf führen!
- **Richtige Größe**
  Die Größe des anzufordernden Speicherbereiches muss fehlerfrei berechnet werden!
- **Anforderung kann fehlschlagen**
  Test erforderlich, ob wirklich Speicher alloziert werden konnte!
- **Freigabe**
  Nach Benutzung muss der Speicher wieder freigegeben werden!
- **Kein Zugriff auf bereits freigegebenen Speicher**
  Auf Speicher, der bereits freigegeben wurde, darf nicht mehr zugegriffen werden!

Empfehlung: Grundgerüst mit Tests und Freigabe zuerst

```
1  int      *dynint = NULL;
2  ...
3  if (0 != n) {
4    if(((uintmax_t)SIZE_MAX / (uintmax_t)sizeof(int)) >= (uintmax_t)n) {
5      dynint = (int *)calloc( n, sizeof(int) );
6      if (NULL != dynint) {
7        /* ... Verwendung von dynint ... */
8        free(dynint); dynint = NULL; /* Freigabe, Zeiger wieder NULL! */
9      } else {
10       /* !!! FEHLERMELDUNG, SPEICHERANFORDERUNG FEHLGESCHLAGEN !!! */
11     }
12   } else {
13     /* !!! FEHLERMELDUNG, UEBERLAUF IN GROESSENBERECHNUNG !!! */
14   }
15 } else {
16   /* !!! FEHLERMELDUNG, n IST 0 !!! */
17 }
```

Datum und Uhrzeit mit C-Runtime-Library

- **Datentypen und Funktionen**

	Windows	POSIX
Datentyp für Zeit in Sekunden:	__time64_t	time_t
Funktion für aktuelle Zeit:	_time64()	time()
Funktion für Zerlegung in Komponenten:	_localtime64_s()	localtime_r()

- **Ausgewählte Komponenten** in struct tm:

Komponente	Bedeutung
tm_mday	Tag innerhalb des Monats, beginnend mit 1
tm_mon	Nummer des Monats, beginnend mit 0 für Januar
tm_year	Jahr ab 1900 (also 116 für das Jahr 2016)
tm_hour	Stunde
tm_min	Minute
tm_sec	Sekunde

Informationen über Datei mit C-Runtime-Library

- **Datentypen und Funktionen**

	Windows	**POSIX**
Datentyp:	struct __stat64	struct stat
Funktion:	_stat64()	stat()

- **Ausgewählte Komponenten** der Datenstruktur

Name	Datentyp Windows	Datentyp POSIX	Inhalt
st_mode	unsigned short	mode_t	Dateityp und Zugriffsrechte
st_size	_off_t	off_t	Dateigröße
st_ctime	__time64_t	time_t	Erstellungszeitpunkt
st_mtime	__time64_t	time_t	Zeitpunkt der letzten Änderung
st_atime	__time64_t	time_t	Zeitpunkt des letzten Zugriffs

Resourcenfreigabe und Fehlerbehandlung - Strukturierte Programmierung

- Verschachtelte if-Anweisungen für Tests auf erhaltene Resourcen und Fehlerbedingungen
- Vorteil: Zügiges Schreiben des Quelltextes
- Nachteil: Einrückungstiefe steigt mit Anzahl der Tests
- Nachteil: Code für Fehlerbehandlung u. U. weit entfernt von Feststellung des Fehlers

Resourcenfreigabe und Fehlerbehandlung - Einzelne Sprungmarke

- Einzelne Sprungmarke am Ende der Funktion
- Im normalen Code (oberhalb der Sprungmarke): Einzelne Codeabschnitte jeweils für Bearbeitung von Teilproblemen oder Anforderung einer Resource. Bei Fehler sofortige Fehlerausschrift bzw. Fehlerbehandlung, dann Sprung zur Sprungmarke
- Nach Sprungmarke: Prüfung welche Resourcen erhalten wurden und ggf Freigabe der Resourcen, anschließend Rückkehr aus Funktion
- Vorteil: Code zur Fehlerbehandlung steht nahe bei Code für Feststellung
- Vorteil: Die aufeinanderfolgenden Codeabschnitte haben jeweils die Einrückungstiefe 1, somit gut kommentierbar
- Nachteil: Vor Freigabe der Resourcen ist jeweils erneut Test notwendig, ob Resource erhalten wurde (evtl. Performance-Einbuße)
- Nachteil: Alle Resourcen-Variablen müssen auf „Resource nicht erhalten“ initialisiert sein

Resourcenfreigabe und Fehlerbehandlung - Sprungmarkenkette

- Mehrere Sprungmarken am Ende der Funktion
- Im normalen Code (oberhalb der Sprungmarke): Einzelne Codeabschnitte jeweils für Bearbeitung von Teilproblemen oder Anforderung einer Resource. Bei Fehler sofortige Fehlerausschrift, dann Sprung zur Sprungmarke für Freigabe der bisher erhaltenen Resourcen
- Nach Sprungmarken: Freigabe jeweils einer erhaltenen Resource
- Nach allen Sprungmarken und zugehörigem Code: Rückkehr aus Funktion
- Reihenfolge der Freigaben in umgekehrter Reihenfolge der Anforderung
- Vorteil: Code zur Fehlerbehandlung steht nahe bei Code für Feststellung
- Vorteil: Die aufeinanderfolgenden Codeabschnitte haben jeweils die Einrückungstiefe 1, somit gut kommentierbar
- Nachteil: Mehr potenzielle Fehlerquellen beim Schreiben des Codes, da jeweils zur richtigen Sprungmarke gesprungen werden muss