

# Praxis der Programmierung

**Institut für Informatik und Computational Science  
Universität Potsdam**

**Henning Bordihn**

*Einzelne Folien gehen auf A. Terzibaschian zurück.*

# Organisatorisches

## Vorlesung und Übung

- Vorlesung gibt Überblick und erklärt Konzepte (*prüfungsrelevant!*)
- Übungen vertiefen ausgewählte Elemente des Vorlesungsstoffs und liefern Ergänzungen
- **drei** Übungsgruppen: Montag 12:15 – 13:45 *oder*  
Dienstag 14:15 – 15:45 *oder*  
Mittwoch 16:15 – 17:45
- Bitte für **genau** eine Gruppe anmelden!!!

## Ziele des Kurses

- Programmieren “im Kleinen”
  - mit einer imperativen Sprache
  - mit einer objektorientierten Sprache
- Verständnis zugrundeliegender Konzepte und Techniken
- Implementierung einiger typischer Algorithmen und Datenstrukturen

## Inhalte des Kurses (1)

- **Voraussetzung:** Grundlagen der Programmierung (Python)
- imperative Programmierung mit C
  - Code, Compiler, Linker
  - einfache Typen, Variablen, Ausdrücke, Ein- und Ausgabe
  - Kontrollstrukturen, Funktionen, Parameter
  - Pointer (Verweisdatentypen), Arrays
  - Strukturen und Typdefinition

## Inhalte des Kurses (2)

- objektorientierte Programmierung mit C++ und Java
  - Klassen, Objekte, Datenelemente, Methoden, Konstruktoren
  - Vererbung und Polymorphie
  - abstrakte Klassen, Interfaces, Templates, Generics
  - Definition und Nutzung von Bibliotheken
- Realisierung typischer algorithmischer Konzepte
  - Rekursion
  - schnelles Sortieren
  - verkettete Listen

## Leistungserfassung

- zwei 45-minütige **Tests** in den Übungen: 18.–20.5. und 22.–24.6. (Programmieraufgaben)  
    ~> Das Bestehen beider Tests ist Zulassungsvoraussetzung zur Klausur.  
    Die Leistung in den Tests geht aber nicht in die Note ein.
- Zwei 60-minütige **Klausuren** am 5.6. (in der Vorlesung) und am 30.7., 10 Uhr im H04
- **Hausaufgaben:** Unregelmäßig werden Hausaufgaben angeboten. Hausaufgaben sind eine freiwillige Leistung. Man kann sich aber Bonuspunkte für die Klausur erarbeiten.

## Falls Sie nur EiP brauchen ...

- Der erste Teil zu C-Programmierung endet mit der ersten Klausur am 5.6.
- Die Klausurnote ist dann Ihre EiP-Note (3 Leistungspunkte).
- Damit steigen Sie aus dem Kurs aus.
- *Für die anderen:*  
Danach geht es um OO-Programmierung. Die zweite Klausur hat nur diese Inhalte zum Gegenstand. Die Gesamtnote (für 6 LP PdP) ist der Mittelwert der beiden Klausurnoten.

**Zugang zu den Materialien:** Link auf

<http://www.cs.uni-potsdam.de/bordihn/teaching.php>



# Die Programmiersprache C

# Programmiersprachen

- Programm

Folge von Anweisungen, die auf einem Computer ausgeführt werden können

- realisieren Algorithmen
- muss vom Prozessor verarbeitet werden
- Binärfolgen

z.B. Folgen von 32-Bit-Sequenzen

1000 1010 0001 0100 1000 1010 1100 1101

- für Menschen kaum lesbar

**Lösung:** Programmiersprachen



# Abstraktionsebenen von Programmiersprachen

## Maschinensprachen

Binärcodes  
abgestimmt auf die  
Architektur eines  
Prozessors (Register)

## Assembler

an Prozessorbefehle  
angelehnt;  
arbeitet mit virtuellen  
Registern

## Hochsprachen

Befehlssatz an  
menschliche Denkweise  
angepasst



Abstraktionsgrad nimmt zu

Übersetzung durch Compiler oder Interpreter

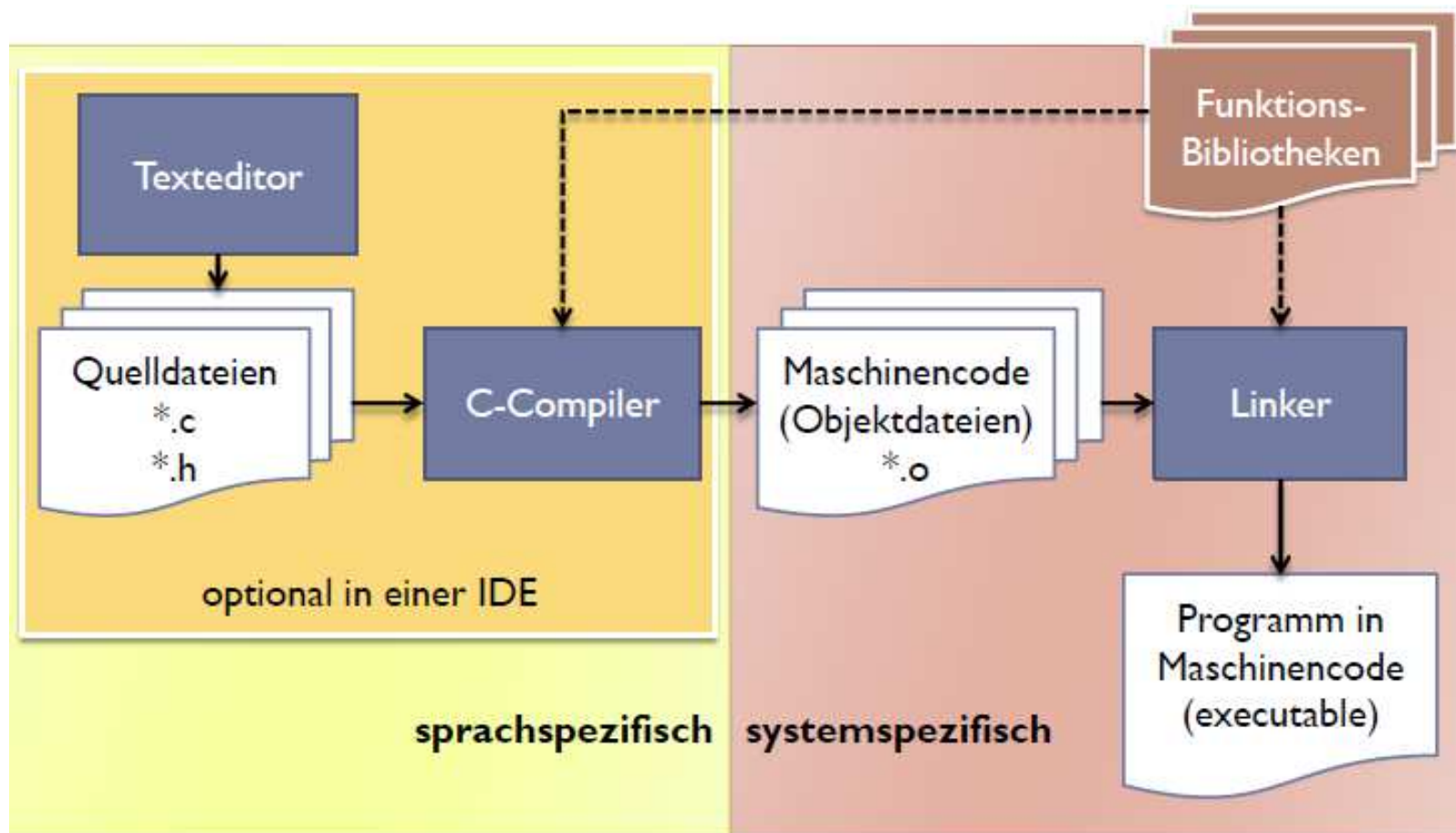
# Die Programmiersprache C

- höhere Programmiersprache (mit einigen Assembler-ähnlichen Konstrukten)
  - **imperative Sprache**: definiert Rechenwege (*Wie* wird gerechnet?)
  - klare, relativ einfache Syntax (wenige, assoziative Schlüsselwörter)
  - Compilersprache (Übersetzung **vor** der Ausführung)
- 1970/71 aus dem Vorläufer B entwickelt (Kernighan & Richie)
- universell, weit verbreitet
- UNIX ist in C geschrieben (Kern und die meisten Systemkommandos)
- viele moderne Sprachen eng an C angelehnt (z.B. C++, Java, C#)

## C-Compiler

- Linux/Unix gcc
- iOS gcc, clang
- Windows VC++, Cygwin
- übersetzen C-Quellcode in Maschinencode  
Quellcode (C-Code) ist **portabel** (unabhängig vom OS)
- Quellcode in (oft mehreren) (Text-)Dateien  
*typischerweise mit Endungen .c und .h*
- ~> Programm-Entwicklung mit Texteditor + Compiler
- ~> Integrierte Entwicklungsumgebungen (IDE)  
(z.B. Eclipse CDT, Visual Studio, ...)

## Entwicklung mit C



# Aufbau eines C-Programms

## Programm (Konzept)

- ein Text (Code), der einen *Algorithmus* formuliert, so dass er auf einer Rechenanlage ausgeführt werden kann
- Ein **Algorithmus** ist eine Folge von Anweisungen, die Eingabedaten in Ausgabedaten überführt (intuitiver Algorithmenbegriff).

**Dabei muss bei jeder Eingabe eindeutig sein:**

- Welche Anweisung wird zuerst ausgeführt?
- Welche Anweisung folgt auf eine gerade ausgeführte Anweisung?
- In welchen Situationen ist der Algorithmus beendet?



## Umsetzung des Programm-Konzepts in C

1. Ein C-Programm berechnet eine **Funktion**.

*algorithmisch:* Eingabedaten  $\longrightarrow$  Ausgabedaten

*mathematisch:* Argumente  $\longrightarrow$  Funktionswert

*in C:* (aktuelle) Parameter  $\longrightarrow$  Rückgabewert

2. Berechnung von Funktionen durch Abarbeitung einer Folge von **Anweisungen**.

$\rightsquigarrow$  C ist eine *imperative Programmiersprache*

# Funktionen und C-Programme

- Funktionen können weitere Funktionen aufrufen, z.B.:  
 $f(x) = \sin(\ln x)$     vordefinierte Funktionen in richtiger Folge aufrufen  
 $g(x) = \sqrt{f(x)}$     selbstdefinierte und Standardfunktion aufrufen
- Aufruf  $g$  mit Argument  $x$      $\rightarrow$      $g$ : Aufruf  $f$  mit Argument  $x$   
 $\rightsquigarrow$   $f$ : Rückgabewert  $f(x)$   
 $\rightarrow$      $g$ : Aufruf  $\sqrt{\phantom{x}}$  mit  $f(x)$   
 $\rightsquigarrow$   $\sqrt{\phantom{x}}$ : Rückgabewert  $\sqrt{f(x)}$   
 $\rightsquigarrow$   $g$ : Rückgabewert  $g(x) = \sqrt{f(x)}$
- Den Rückgabewert einer aufgerufenen Funktion erhält die aufrufende Funktion.

## Struktur von C-Programmen

- C-Programm: Definition einer oder mehrerer Funktionen
  - vom Programm realisierte Funktion: `main()`
    - ↪ wird stets zuerst aufgerufen
  - ggf. weitere, aufzurufende Funktionen
- Häufig zu benutzende Funktionen (**Standardfunktionen**) sind in *Bibliotheksdateien* vordefiniert.
  - ↪ können eingebunden und dann aufgerufen werden
- Besonderheit: Den Rückgabewert von `main()` erhält das Programm, das das C-Programm aufruft
  - ↪ kann als Exit-Status interpretiert werden
  - ↪ `main()` gibt ganzzahligen Wert zurück

## Ein erstes Programm

```
/* hello.c
 *
 * Ausgabe einer Zeichenkette auf stdout
 */

#include <stdio.h>          // Bibliotheksdatei einbinden

int main() {
    printf("Hello world!\n");
    return 0;               // Rueckgabewert 0 (alles o.k.)
}
```

## Erläuterungen zum ersten Programm

- Zeichen hinter `//` und zwischen `/*` und `*/` sind *Kommentar*
- `int main()`:
  - `()` zeigen (stets) an, dass es sich um eine Funktion handelt
  - `int` zeigt an, dass der Rückgabewert ganzzahlig ist
- `printf()`:
  - Aufruf einer Funktion zur formatierten Ausgabe auf `stdout`
  - ist Standardfunktion, die in der Bibliotheksdatei `stdio.h` deklariert ist
  - Parameter von `printf()` zwischen `()`:  
Anführungszeichen  $\rightsquigarrow$  Zeichenkette; `\n`  $\rightsquigarrow$  Zeilenvorschub (**n**ewline)
- Kommandos und Funktionsaufrufe müssen mit `;` abgeschlossen werden

## Vom Quellcode zum ausführbaren Code

- Aufruf des Compilers:

```
gcc [-Wall] beispiel.c [-o beispiel]
```

1. **Präprozessor** bereitet den Quellcode zur Übersetzung vor
  - kopiert Bibliotheksdateien (für den Übersetzungslauf) in den Quellcode,
  - erstellt „Aliasnamen“ im Quellcode u.ä.
2. **Compiler** übersetzt in *Objektcode*: Befehlsfolgen für den Prozessor
3. **Linker** verbindet mehrere Objektcode-Dateien zu einer Datei

- Option `-Wall`: alle Warnungen ausgeben (*empfohlen!*)
- Option `-o`: Name der Ausgabedatei festlegen (default: `a.out`)

## Präprozessor-Anweisungen

- beginnen mit #
- enden *nicht* mit Semikolon
- Beispiel: `#include datei`
  - bindet *datei* für die Arbeit des Compilers in den Quellcode ein
  - Funktionen, die in *datei* deklariert sind, werden verfügbar
  - *datei* in Anführungszeichen: *datei* aus aktuellem Verzeichnis
  - *datei* in spitzen Klammern: *datei* aus Verzeichnis mit C-Bibliotheken (z.B. `/usr/include`)

# Variablen und Datentypen in C



# Variablen

- dienen zum Speichern von Werten (Parameter, (Zwischen-)Ergebnisse etc.)
- Werte werden im Arbeitsspeicher abgelegt  
(an eindeutiger, zum Variablennamen gehörender Speicherstelle)
- Werte werden über den Variablennamen aufgefunden
- Werte können verändert werden
- haben einen eindeutigen, unveränderlichen Datentyp  
     $\rightsquigarrow$  C ist *typisiert*

## Variablennamen in C

- Zeichenketten aus ASCII-Buchstaben, Ziffern und “underline” `_`, die mit einem Buchstaben oder `_` beginnen
- Groß- und Kleinschreibung wird unterschieden!!!
- maximale Länge (systemabhängig) zwischen 63 und 255 Zeichen

## Lebenszyklus von Variablen

### Variablen

- müssen **deklariert** werden, z.B. `int x; oder float f1, f2;`
    - *Datentyp Variablenname;*
    - Reservierung genügend vieler Speicherzellen im Arbeitsspeicher (abhängig vom *Datentyp*)
  - werden durch die erste Wertzuweisung **initialisiert**, z.B. `x = 3;`  
gleichzeitige Definition und Initialisierung: `int x = 3;`
  - **Anweisungen** ändern Werte der Variablen,  
z.B. Überschreiben durch Wertzuweisung, z.B. `x = y - x;`
- ⇒ `=` ist **Zuweisungsoperator**, nicht symmetrisch:  
`3 = x` ist *keine* gültige Anweisung

## Datentypen

- Datentyp einer Variablen bestimmt
  - Darstellung (Repräsentation) der Werte im Arbeitsspeicher
    - \* Anzahl der Speicherzellen (Bytes)  $\rightsquigarrow$  Wertebereich/Genauigkeit
    - \* Bedeutung der einzelnen Bits
  - erlaubte Operationen und deren Wirkung
- **einfache/elementare Datentypen:**
  - Ganzzahltypen `char`, `int`, `short`, `long`, `long long` und deren `unsigned` Typen (z.B. `unsigned int`)
  - Gleitpunkttypen `float`, `double`, `long double`
- **abgeleitete Datentypen:** setzen sich aus anderen Datentypen zusammen
- sind als *Standardtypen* vordefiniert (z.B. alle elementaren Typen) oder *selbst definierte Typen*

## Elementare Ganzzahltypen

Datentyp	Bytes (z.B.)	Wertebereich (dezimal)
[signed] char	1	-128 ... +127
unsigned char	1	0 ... 255 (erweiterter ASCII-Satz)
[signed] short [int]	2	-32.768 ... +32.767
unsigned short [int]	2	0 ... +65.535
[signed] int	4	-2.147.483.648 ... +2.147.483.647
unsigned int	4	0 ... +4.294.967.295
[signed] long [int]	4	-2.147.483.648 ... +2.147.483.647
unsigned long [int]	4	0 ... +4.294.967.295
[signed] long long	8	$-2^{63} \dots +2^{63} - 1$
unsigned long long	8	$0 \dots +2^{64} - 1$

Größe vom Compiler abhängig, aber stets:

$$|\text{char}| < 2 \leq |\text{short}| \leq |\text{int}| \leq 4 \leq |\text{long}| \leq |\text{long long}|$$

## signed *versus* unsigned

char	0	0	0	0	1	1	0	1	=	13
unsigned char	0	0	0	0	1	1	0	1	=	13
char	1	0	0	0	1	1	0	1	=	-13
unsigned char	1	0	0	0	1	1	0	1	=	141

 optionales Vorzeichenbit

- Achtung: Darstellung in Maschine im 2er Komplement

## Elementare Gleitpunktypen

Datentyp	Bytes (z.B.)	Wertebereich (dezimal)
float	4	$-3,4 \cdot 10^{38} \dots +3,4 \cdot 10^{38}$
double	8	$-1,7 \cdot 10^{308} \dots +1,7 \cdot 10^{308}$
long double	10	$-1,1 \cdot 10^{4932} \dots +1,1 \cdot 10^{4932}$

- Nutzen der Exponentialschreibweise zum "Sparen von Bits", z.B.:

$$\begin{array}{lll}
 \triangleright 0,0000356 & = 3.56 & * 10^{(-5)} \\
 \triangleright 356\ 000\ 000 & = 3.56 & * 10^{(8)} \\
 \triangleright 3,1416 & = \underbrace{3.1416}_{\text{Mantisse}} & * \underbrace{10^0}_{\text{Exponent}}
 \end{array}$$

- interne Darstellung:  $Mantisse * 2^{Exponent}$  (nach IEEE 754)
- von float zu long double wächst die Genauigkeit (Dezimalstellen)

## Literale

- bezeichnen eine Konstante, die durch ihren Wert dargestellt wird
- **ganzzahlige Literale:**
  - `int`:    dezimal            `[1-9][0-9]*`    oder    `0`                            z.B.: 26  
              oktal             `0[0-7]*`                            z.B.: 032  
              hexadezimal    `0x[0-9a-f]+`    oder    `0X[0-9A-F]+`    z.B.: 0x1a  
  
               $\rightsquigarrow$  stets positiv; ggf. Minus-Operator anwenden
  - Suffix `u` oder `U`  $\rightsquigarrow$  unsigned
  - Suffix `l` oder `L`  $\rightsquigarrow$  long



## Gleitkomma-Literale

- `double`:    Dezimalbruch                    mit Dezimalpunkt            z.B.: 300.0  
                 Exponentialdarstellungen    *Mantisse**eExponent*        z.B.: 3e2  
                 (zur Basis 10)                *Mantisse**EExponent*        z.B.: .3E3
- Suffix `f` oder `F`  $\rightsquigarrow$  `float`
- Suffix `l` oder `L`  $\rightsquigarrow$  `long double`
- Beispiele erlaubter Werte im Quellcode:  
    3.1416e-4            // Exponentialschreibweise (e=10)  
    3.14159265e+300    // ein sehr großer Gleitkommawert  
    17.42                // auch ohne Exponent  
    -35                  // auch ganze Zahlen möglich

## Character-Literale

- **Character-Literale:**

- Zeichen in einfachen Hochkommata, z.B. '0'
- Ersatzdarstellungen nicht druckbarer Zeichen in einfachen Hochkommata, z.B. '\n', '\t', aber auch '\\'
- Oktal- oder Hexadezimaldarstellung des Zeichens:  
'\Oktalziffern' ('\060') bzw. '\xHexadezimalziffern' ('\x30')

# Arithmetik in C

## Operatoren (1)

- arithmetische Operatoren: +, -, \*, /, %  
(% (Modulo) nur für den Ganzzahltyp)
- Vergleichsoperatoren: ==, !=, <, >, <=, >=
- bei geschachtelten Operatoren:
  - vordefinierte *Prioritäten* (z.B. \* vor +)  
Bsp.: 3+5\*2 ergibt 13
  - Auswertung bei gleicher Priorität von links nach rechts (*Linksassoziativität*)  
Bsp.: 5-3-2 ergibt 0
  - explizite Reihenfolge mit Klammern ausdrücken!

## Operatoren (2)

- Ausdrücke haben einen Rückgabewert  $\rightsquigarrow$  können Teil eines Ausdrucks sein
- Inkrement und Dekrement in Präfix- und Suffixnotation:
  - **Präfixnotation:** `++A` bzw. `--A`  
 $\rightsquigarrow$  Rückgabewert ist Inkrement bzw. Dekrement von `A`  
*Nebeneffekt:* Wert von `A` ist in- bzw. dekrementiert
  - **Postfixnotation:** `A++` bzw. `A--`  
 $\rightsquigarrow$  Rückgabewert ist der Wert von `A`  
*Nebeneffekt:* Wert von `A` ist in- bzw. dekrementiert

zum Vergleich:

`A + B` gibt die Summe der Werte von `A` und `B` zurück; keine Nebeneffekte

## Operatoren (3)

- Zuweisungsoperatoren `+=`, `-=`, `*=`, `/=`, `%=` geben den Wert der Operation zurück und weisen als Nebeneffekt diesen Wert dem linken Ausdruck zu, z.B.:

`x += 8` realisiert `x = x + 8`

- bitweise Operatoren, sonstige Operatoren, Assoziativitäten und Prioritäten  
s. Literatur, z.B.:

M. Dausmann, U. Bröckl, J. Goll: C als erste Programmiersprache.  
Teubner Verlag/GWV Fachverlage, Wiesbaden, 2008.

- Funktionsaufrufe können eingebunden werden (z.B. `3+sin(1.2)`)
- vordefinierte mathematische Funktionen und Konstanten  
`#include <math.h>`

## Beispiele vordefinierter Funktionen (math.h)

Definition	
<code>double cos(double x)</code>	Kosinus von x
<code>double sin(double x)</code>	Sinus von x
<code>double acos(double x)</code>	Arkustangens von x
<code>double asin(double x)</code>	Arkussinus von x
<code>double exp(double x)</code>	$e^x$
<code>double log(double x)</code>	$\ln(x)$
<code>double log10(double x)</code>	$\log_{10}(x)$
<code>double ceil(double x)</code>	Aufrunden zur nächsten Ganzzahl
<code>double floor(double x)</code>	Abrunden zur nächsten Ganzzahl
<code>double pow(double y, double x)</code>	$y^x$
<code>double fabs(double x)</code>	Betrag von x
<code>double sqrt(double x)</code>	Wurzel von x

## Typumwandlung

- bei Kombination verschiedener Typen in einem Ausdruck oder Zuweisung eines Ausdrucks an eine Variable eines anderen Typs
- **explizite Typumwandlung** durch den Programmierer:  
(*Zieltyp*) *Ausdruck*  

```
int x = 3;  
float pi = 3.14;  
x = x * (int) pi;    // x ist 9
```
- **implizite Typumwandlung** durch den Compiler:  
wenn immer es nötig und möglich ist, mit möglichst geringem Genauigkeitsverlust



## Implizite Typumwandlung bei einfachen Typen

- Wenn Operatoren verschiedene Typen verknüpfen:
  - kleinere Ganzzahltypen werden immer nach `int` umgewandelt.  
(`unsigned short` in `unsigned int`, falls `short` und `int` äquivalent sind)
  - Umwandlung aller Operanden in den höchsten Typ des Ausdrucks gemäß

`int` → `unsigned int` → `long` → `unsigned long` → `long long`  
→ `unsigned long long` → `float` → `double` → `long double`

- Bei Wertzuweisungen:
  - Umwandlung des rechten Ausdrucks in Typ der linken Variablen

## Implizite Typumwandlung (Beispiele)

Datentyp von $x$	Datentyp von $y$	Datentyp von $x*y$ und $y*x$
double	float	double
double	int	double
float	int	float
long	int	long

etc.

## Verhalten der Werte bei Typumwandlungen

- $\text{gro\ss} \rightarrow \text{kleinerer Ganzzahltyp} \rightsquigarrow \text{Abschneiden der oberen Bits}$
- $\text{Ganzzahltyp} \rightarrow \text{Gleitpunkttyp} \rightsquigarrow \text{(meist) n\acchster darstellbarer Wert}$
- $\text{Gleitpunkttyp} \rightarrow \text{Ganzzahltyp} \rightsquigarrow \text{Abschneiden der Nachkommastellen}$
- $\text{Gleitpunkttyp} \rightarrow \text{Typ mit zu kleinem Wertebereich} \rightsquigarrow \text{unbestimmt}$

# Ein- und Ausgabe von C-Programmen

## Formatierte Ausgabe mit printf()

- variable Anzahl von Parametern (Argumenten)
- erstes Argument wird ausgegeben (s. `printf("Hello world!\n")`)
- Argumente durch *Komma* voneinander getrennt
- erstes Argument kann auf Werte der weiteren Argumente zugreifen, z.B.:

```
int x = 42;  
printf("%d\t%d\n", 1, x); ~> 1      42
```

- `%d` ist ein Formatelement:  
der nächste, noch nicht verwendete Parameter wird an Stelle des `%d`  
als dezimale ganze Zahl ausgegeben
- `\t` ~> Tabulatorschritt

## Formatelemente von `printf()`

<code>%d</code>	dezimale ganze Zahl
<code>%md</code>	dezimale ganze Zahl, mindestens <i>m</i> Zeichen breit
<code>%f</code>	Gleitpunktzahl
<code>%mf</code>	Gleitpunktzahl, mindestens <i>m</i> Zeichen breit
<code>%.nf</code>	Gleitpunktzahl, <i>n</i> Nachkommastellen
<code>%m.nf</code>	Gleitpunktzahl, mind. <i>m</i> Zeichen inkl. <i>n</i> Nachkommastellen
<code>%o</code>	oktale ganze Zahl
<code>%x</code>	hexadezimale ganze Zahl
<code>%c</code>	einzelnes Zeichen (Datentyp <code>char</code> )

(mehr auf der Manpage `man printf`; wie in Python)

## Benutzereingaben mit scanf

- formatiertes Einlesen eines Wertes mit scanf
- Formatelemente wie bei printf
- Beispiele: Einlesen einer ganzen Zahl und Speichern auf `int n`:  
`scanf("%d", &n);`

Einlesen einer Gleitkommazahl und Speichern auf `double x`:  
`scanf("%f", &x)`

Hinweis: Das Zeichen `&` ist nötig, da die Speicheradresse der Variablen angegeben werden muss (und kein Zugriff auf den Wert der Variablen erfolgt; s. Vorlesung zu "Pointern")

# Kontrollstrukturen



# Sequenzen

1. **Block:**  $\{$  *Anweisung\_1*  
          *Anweisung\_2*  
          :  
          *Anweisung\_n*  
           $\}$

- Block fasst eine Folge von Anweisungen zusammen  $\rightsquigarrow$  „*Sequenz*“
  - Block kann überall auftreten, wo Anweisungen stehen dürfen  
 $\rightsquigarrow$  Blöcke können geschachtelt werden
  - kein Semikolon nach einem Block
- $\rightsquigarrow$  Anweisungen der `main()`-Funktion bilden einen Block

## Iterationen (1)

### 2. **while**-Schleife: `while (Ausdruck)` *Anweisung*

- Wiederholung der *Anweisung* solange, bis der *Ausdruck* den Wert 0 hat
  - **arithmetischer** *Ausdruck*
  - **boolescher** *Ausdruck* mit Werten  $\neq 0$  für true oder 0 für false
    - $\rightsquigarrow$  Vergleichsoperatoren: `==`, `!=`, `<=`, `>=`, `<`, `>`
    - und logische Operatoren `&&`, `||`, `!`
- kann auch gar nicht ausgeführt werden  $\rightsquigarrow$  „*abweisende Schleife*“

### 3. **do while**-Schleife: `do {` *Anweisung* `} while (Ausdruck)`

- wird mindestens einmal durchlaufen  $\rightsquigarrow$  „*annehmende Schleife*“

## Iterationen (2)

### 4. **for**-Schleife: `for (init; test; update)` *Anweisung*

<i>init</i> :	Anweisung	$\rightsquigarrow$	Initialisierung des Schleifenzählers
<i>test</i> :	Ausdruck	$\rightsquigarrow$	„while-Bedingung“
<i>update</i> :	Anweisung	$\rightsquigarrow$	Überschreiben des Schleifenzählers

Beispiel: `for (i = 0; i < 10; i++)`

- Schleifenzähler muss als Ganzzahltyp definiert sein
- *init*, *test* oder *update* können leer sein, z.B. `for(;;)`  
 $\rightsquigarrow$  ggf. Anweisung(en) nutzen
- dynamische, abweisende Schleife

## Selektionen (1)

5. einfache Selektion `if` (*Ausdruck*)  
    *Anweisung\_1*  
    else  
        *Anweisung\_2*

- Der else-Zweig ist optional.
- Bei Schachtelung von `if`-Anweisungen bezieht sich ein `else`-Zweig immer auf die letzte `if`-Anweisung ohne `else`.

```
if (x >= 0)
    if (x > 0)
        printf("groesser als Null\n");
    else;
else
    printf("kleiner als Null\n");
```

## Selektionen (2)

### 6. Mehrfachselektion – else if

- Auswahl unter mehreren Alternativen
- if-Anweisung als Anweisung im else-Zweig

```
if (Ausdruck_1)  
    Anweisung_1  
else if (Ausdruck_2)  
    Anweisung_2  
    :  
else if (Ausdruck_n)  
    Anweisung_n  
else                                     /* optional ...  
    Anweisung_else                               ... optional /*
```

## Selektionen (3)

### 7. Wert-gesteuerte, direkte Verzweigung mit switch (nur für Variablen mit ganzzahligem Datentyp)

```
int a = ...;
switch(a) {
    case 10:
        Anweisung für den Fall a ist 10
        break;
    case 20:
        Anweisung für den Fall a ist 20
        break;
    default:
        Anweisung für alle anderen Fälle
        break;
}
```

## Sprunganweisungen

1. `break;`  $\rightsquigarrow$  Abbruch der Schleife  
 $\rightsquigarrow$  zur ersten Anweisung nach der Schleife
2. `continue;`  $\rightsquigarrow$  Abbruch des Schleifendurchlaufs  
 $\rightsquigarrow$  **for:** zum *Update* der Schleife  
 $\rightsquigarrow$  **while:** zur *Bedingung* der Schleife  
 $\rightsquigarrow$  **do-while:** zur ersten Anweisung der Schleife
3. `goto Marke;` Sprung in Anweisung hinter *Marke*:  
nur zum Abfangen von Laufzeitfehlern einsetzen

## Blöcke und Variablen

- In Blöcken deklarierte Variablen sind nur innerhalb dieses Blockes sichtbar.
  - ↪ auch in enthaltenen Blöcken
  - ↪ aber **nicht** in umfassenden Blöcken
- In Blöcken deklarierte Variablen *verdecken* gleichnamige Variablen von umfassenden Blöcken.
- In Blöcken deklarierte Variablen werden beim Verlassen des Blockes wieder ungültig.
  - ↪ überdeckte Variablen werden wieder sichtbar