Grundlagen der C-Programmierung

Ein “Crash-Kurs” für Lehrerinnen und Lehrer im Bereich Elektrotechnik

Version 2.0

andre.maier@elektronikschule.de

est Elektronikschule Tettnang

Drei Argumente für C im Unterricht

- C erlaubt sowohl Abstraktion als auch hardwarenahe Programmierung
- Syntaktische und semantische Prinzipien von C finden sich in einer Vielzahl der heute gängigen höheren Programmiersprachen
- C wird auch heute noch auf allen gängigen Plattformen und Betriebssystemen unterstützt

andre.maier@elektronikschule.de

Nachteile von C im Unterricht

- Der Programmierer benötigt i.d.R. informationstechnisches Grundwissen
- Fehlermeldungen des Compilers sind meist nicht so hilfreich wie es bei anderen höheren Programmiersprachen der Fall ist.
- Die Schülerinnen und Schüler können durch die großen Freiheiten in C überfordert werden.

andre.maier@elektronikschule.de

Die Geschichte von C

- 1966: Martin Richards entwickelt am MIT die Basic Combined Programming Language (BCPL) als Nachfolger der Cambridge Programming Language (CPL)
- 1969: Ken Thompson und Dennis Ritichie entwickeln die Programmiersprache B
- 1972-1973: Dennis Ritchie entwickelt in den Bell Labs die Programmiersprache C
- Seitdem wurden zahlreiche Standards, Dialekte und C-ähnliche Sprachen entwickelt

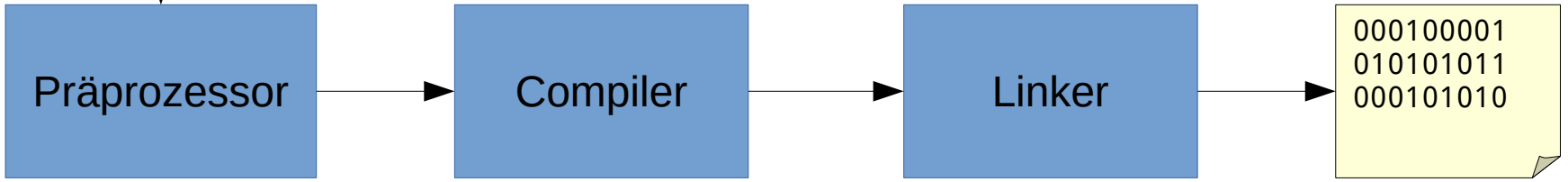
Dennis_Ritchie_(right)_Receiving_Japan_Prize.jpeg: Denise Panyik-Daledерivative work: YMS [CC BY 2.0 (https://creativecommons.org/licenses/by/2.0)]

andre.maier@elektronikschule.de

Der Weg zum ausführbaren Programm

Sourcecode
(Quellcode)

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



- Führt Textersetzungen im Sourcecode durch
- Fügt den Inhalt von Header-Dateien in den Sourcecode ein

- Prüft den Sourcecode auf syntaktische und semantische Fehler
- Übersetzt den Sourcecode in Maschinencode

- Verbindet übersetzte Programm-Module und fertige Programm-bibliotheken
- Nimmt ggf. Anpassung an das Betriebssystem vor

Binary Executable
(Binäre ausführbare Datei)

andre.maier@elektronikschule.de

“Hello World”-Programm auf PC (didaktisch reduziert)

Präprozessor-Anweisung, damit der Compiler die printf-Funktion “kennt”.

Hauptfunktionsblock, der als Einsprungspunkt beim Starten des Programms aufgerufen wird.

Veranlasst eine Bildschirmausgabe des Textes *Hallo Welt!*, gefolgt von einem Zeilenumbruch

```
#include <stdio.h>

int main()
{
    printf("Hallo Welt!\n");
}
```


andre.maier@elektronikschule.de

“Hello World”-Programm auf PC (nicht reduziert)

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hallo Welt!\n");
    return 0;
}
```

Argumente (Daten) übernehmen, die gegebenenfalls beim Programmstart übergeben wurden.

Rückgabe des Wertes 0 an das Betriebssystem, um erfolgreiche Ausführung zu signalisieren.

andre.maier@elektronikschule.de

“Hello World”-Programm auf Arduino

```
void setup()
{
   Serial.begin(9600);
   Serial.println("Hallo Welt!");
}

void loop()
{
}
```

Hauptfunktionsblock, der als Einsprungspunkt beim Starten des Programms ausgeführt wird.

Initialisieren der seriellen Schnittstelle und Einstellen der Baudrate auf 9600 Baud. Danach Ausgabe des Textes *Hallo Welt!*, gefolgt von einem Zeilenumbruch, auf der seriellen Schnittstelle.

Diese Funktion wird nach setup() ausgeführt und ist für Code vorgesehen, der ständig wiederholt ausgeführt werden soll.

andre.maier@elektronikschule.de

Präprozessor

- Der Präprozessor kann u.a.
  - Header-Dateien einbinden
  - Makros ersetzen
  - Sourcecode bedingt übersetzen
- Präprozessoranweisungen starten immer mit dem Zeichen # (also #define, #include, #if, ...)

andre.maier@elektronikschule.de

Variablen

- Variablen müssen vor ihrer ersten Verwendung im Sourcecode **deklariert** werden.

```
int variable_eins;
```

- Achtung: In Standard-C wird eine **Initialisierung** der Variablen vor dem ersten lesenden Zugriff vom Compiler nicht zwingend verlangt. → Fehlerquelle!

andre.maier@elektronikschule.de

Konstanten

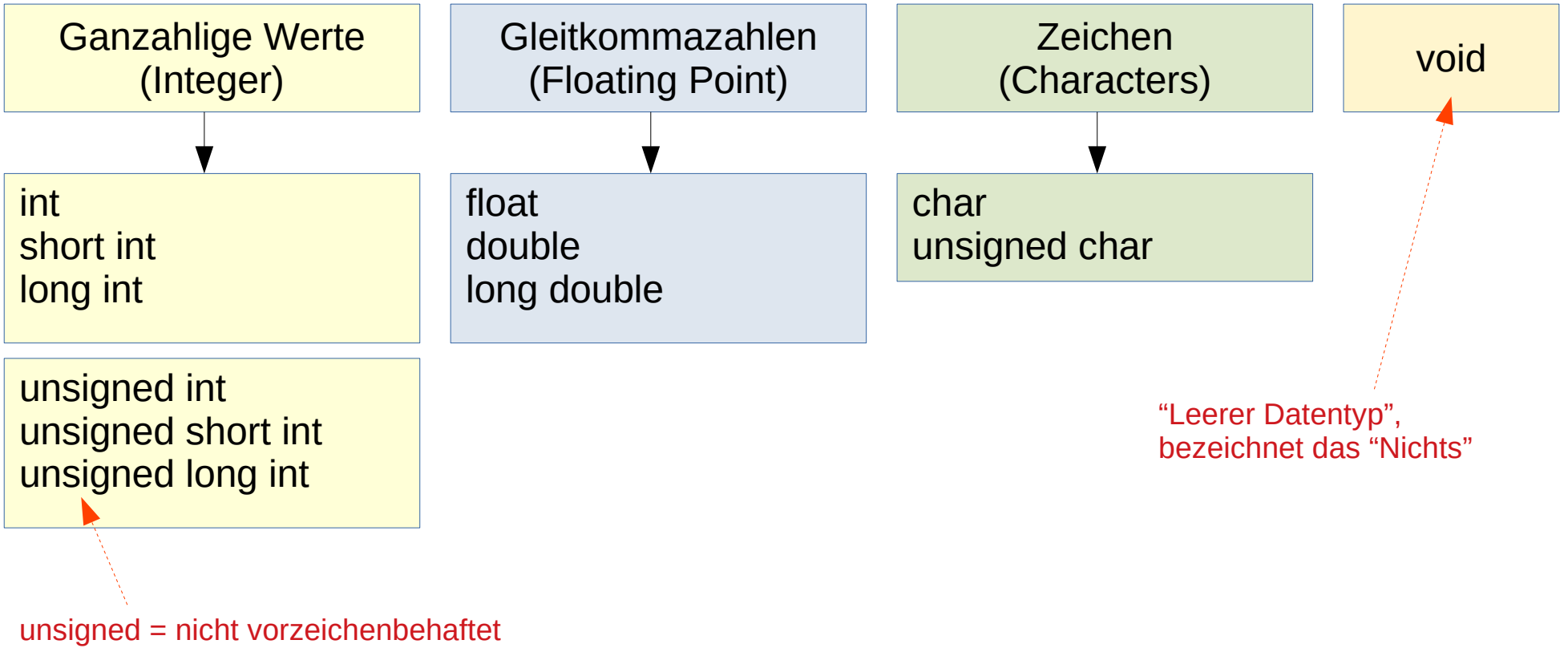
- Symbolische Konstanten
  - werden per Präprozessoranweisung definiert
    z.B. `#define PI 3.14`
  - Vorsicht: Reine Textersetzung!

“Best practice”

- Mit *const* definierte Konstanten
  - Wie eine Variablendeklaration, allerdings mit vorangestelltem Schlüsselwort `const`
    z.B. `const double PI = 3.14159;`

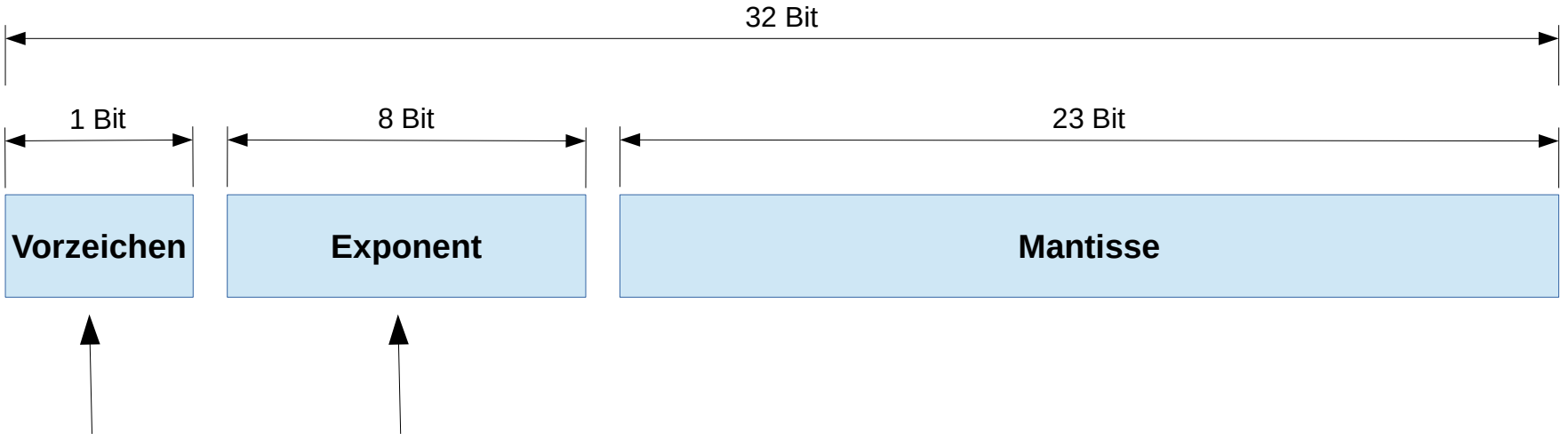
andre.maier@elektronikschule.de

Primitive Datentypen



andre.maier@elektronikschule.de

Gleitkommadarstellung (IEEE754)



0→ +
1→ -

Exponent, bei 8 Bit mit einem Offset von 127 dargestellt.

Mantisse, binär normiert
(Komma wurde so verschoben, dass genau eine 1 vor dem Komma stand.)

andre.maier@elektronikschule.de

Literale

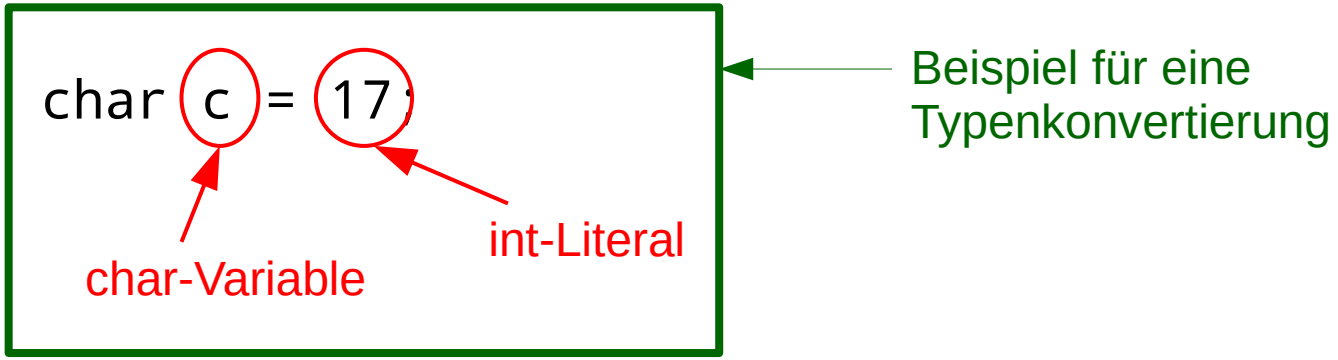
- ... sind direkte Werte (z.B. Zahlenwerte) im Sourcecode.
- Literale sind immer von einem bestimmten Datentyp.

Beispiele:

Literal(e)	Datentyp
42, 0x2A, 052	int
42u	unsigned int
42l	long int
2.5, 3E-5	float
‘A’	char

andre.maier@elektronikschule.de

Typenkonvertierung / Typecast (1)



Frage: Kann hier etwas schiefgehen?

Ja!

andre.maier@elektronikschule.de

Typenkonvertierung / Typecast (2)

```
char c = 4711;
```

Zahlenwert zu hoch für 8-bit
Zahlenbereich von char!

- Mit “Glück” (in einfachen Fällen wie hier) merkt es der Compiler.
- Compiler meldet in C nur ein “Warning”, d.h. das ausführbare Programm wird trotzdem erzeugt.
- Wird das Programm ausgeführt, so wird zur Laufzeit ein falscher Wert in der Variablen c gespeichert.

andre.maier@elektronikschule.de

Typenkonvertierung / Typecast (3)

- Typenkonvertierung kann explizit erzwungen werden.

```
char c = (char)4711;
```

Lies: “Betrachte das/den nachfolgende(n) Literal/Variable/Ausdruck als char”

- Compiler liefert dann keine Warnung mehr und geht davon aus, dass der Programmierer weiß, was er da tut.

andre.maier@elektronikschule.de

Aliasse für Datentypen

```
typedef unsigned char byte;

typedef unsigned long ulong;

…

byte b1, b2;

ulong n;

…
```

- Verringern die Schreibarbeit bei Deklarationen
- Verbessern die Lesbarkeit des Sourcecodes
- Erhöhen die Portabilität und Kompatibilität von Programmen

andre.maier@elektronikschule.de

Binäre Operatoren in C

- Arithmetisch
  + - * / %
- Bitweise
  ~ & | ^ >> <<
- Logisch
  && || ! & |
- Komparativ
  == != < > <= >=
- Zuweisend
  = += -= *= /= %=

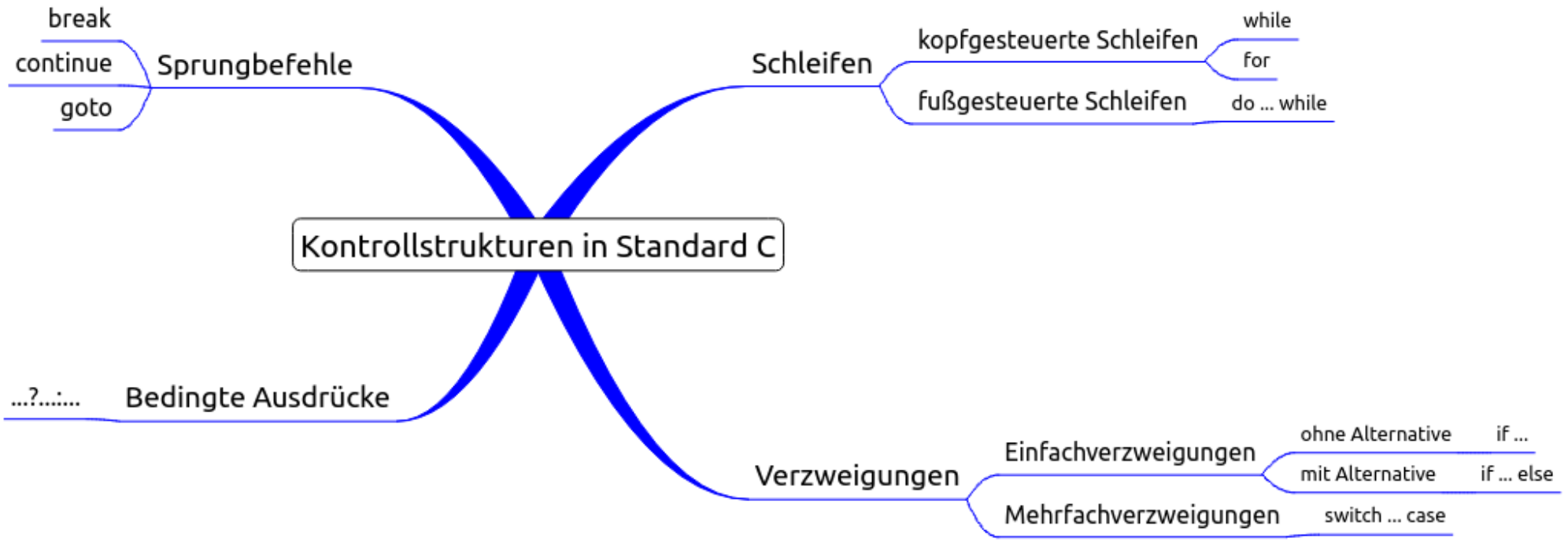
andre.maier@elektronikschule.de

Unäre Operatoren in C

- Bitweise Negation
  ~
- Inkrement
  ++
- Dekrement
  --

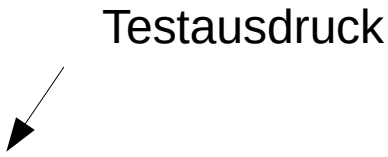
andre.maier@elektronikschule.de

Kontrollstrukturen (Übersicht)



andre.maier@elektronikschule.de

Einfachverzweigungen ohne Alternative




```
if(n<0)
{
   // Code, der nur dann und nur dann ausgeführt
   // wird, wenn der Testausdruck wahr ist.
}
```


andre.maier@elektronikschule.de

Einfachverzweigungen mit Alternative

Testausdruck

```
if(n<0)
{
   // Code, der dann und nur dann ausgeführt
   // wird, wenn der Testausdruck wahr ergibt.
}
else
{
   // Code, der dann und nur dann ausgeführt wird,
   // wenn der Testausdruck falsch ergibt.
}
```

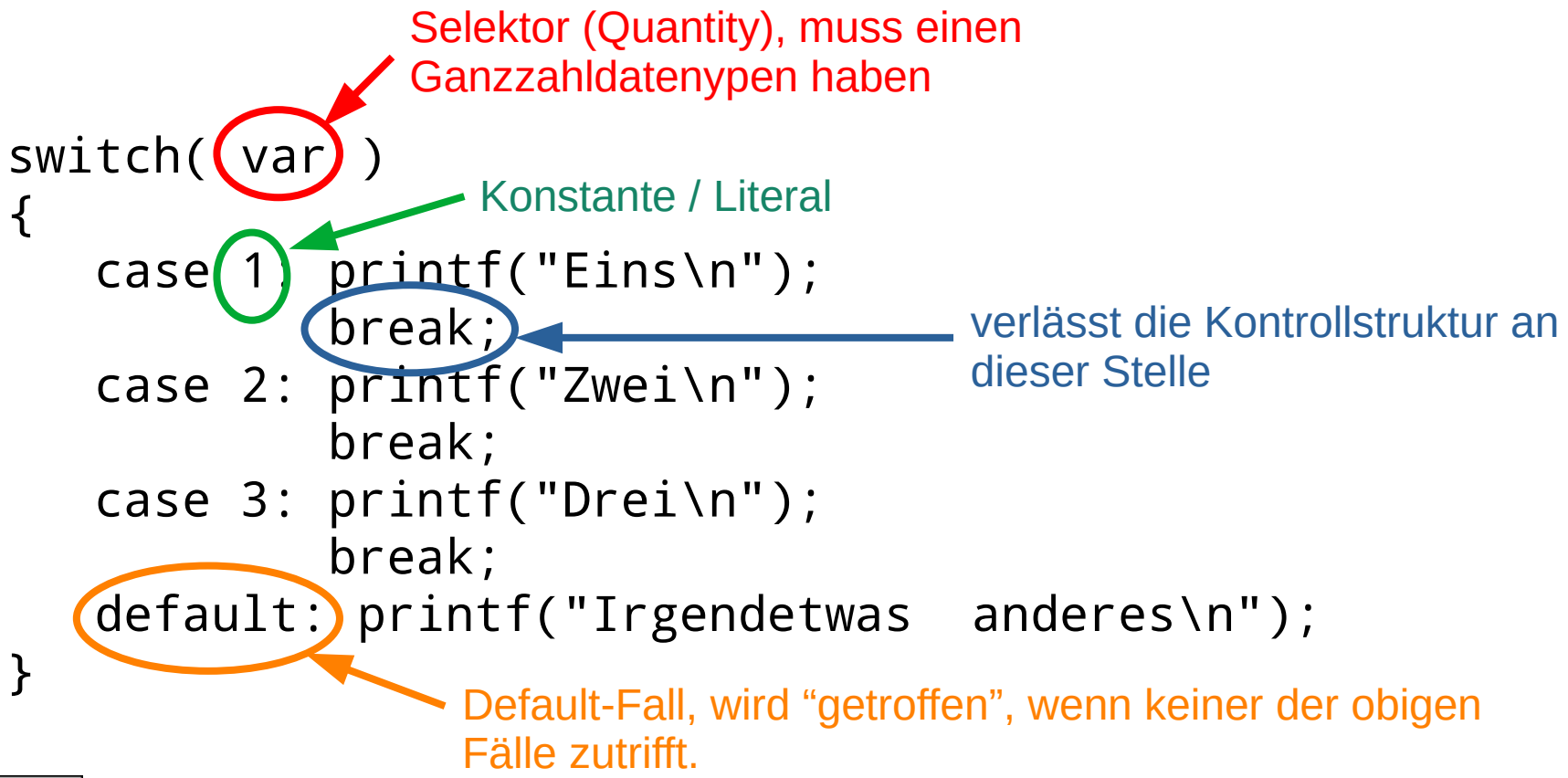

andre.maier@elektronikschule.de

Weglassen von Klammern

- Steht nur einer Anweisung in den geschweiften Klammern nach if,else,while,do,for,... so werden in der Praxis die geschweiften Klammern häufig weggelassen.
- Gefährlich für Anfänger, insbesondere bei späterer Erweiterung des Codes.

andre.maier@elektronikschule.de

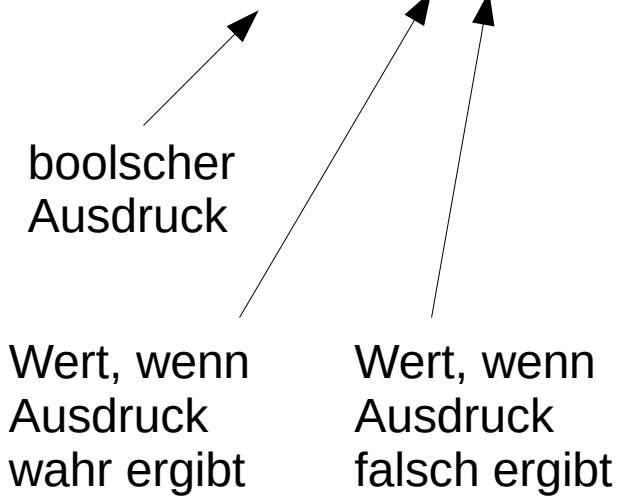
Mehrfachverzweigungen



andre.maier@elektronikschule.de

Bedingte Ausdrücke (1)

```
int a=1;
int n=2*(a<2)?3:4;
```

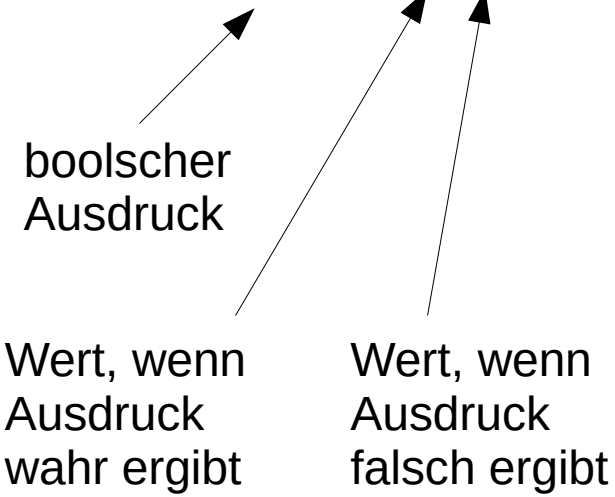


- Ist der boolsche Ausdruck vor dem ?-Operator
  - wahr, so stellt der Gesamtausdruck den Wert nach dem ? dar.
  - falsch, so stelle der Gesamtausdruck den Wert nach dem : dar.

andre.maier@elektronikschule.de

Bedingte Ausdrücke (2)

```
int a=1;
int n=2*(a<2)?3:4;
```




```
int a=1;
int n;
if(a<2)
{
   n=2*3;
}
else
{
   n=2*4;
}
```


andre.maier@elektronikschule.de

Bedingte Ausdrücke (3)

- Hilfreich, um aufwändige if...else-Konstruktionen zu vermeiden, beispielsweise bei Zuweisungen.
- Nachteil: Kann Code unübersichtlich machen.

```
int a=1, b=2;
double c=-0.2;
int x=a<3?c<-0.21?1:b!=0?2:3:4;
```


andre.maier@elektronikschule.de

while-Schleife

Testausdruck (Bedingung)

kopfgesteuert

```
while(...)
{
   // Code, der so oft ausgeführt wird,
   // bis der Testausdruck falsch ergibt.
}
```


andre.maier@elektronikschule.de

do...while-Schleife

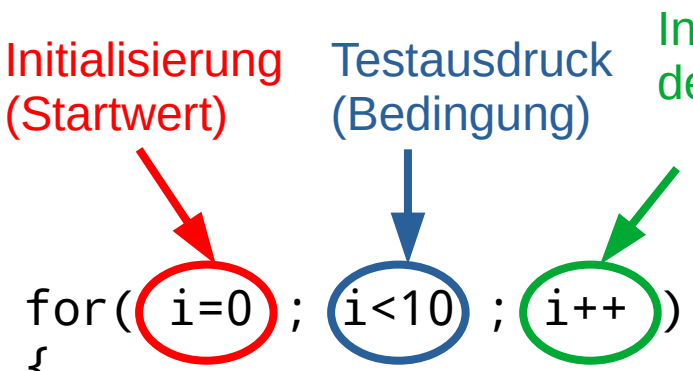
```
do
{
   // Code, der so oft ausgeführt wird,
   // bis der Testausdruck falsch ergibt.
}
while(...);
```

Testausdruck (Bedingung)

fussgesteuert

andre.maier@elektronikschule.de

Zählschleife




```
for( i=0 ; i<10 ; i++ )
{
   // Schleifenkörper wird in diesem Fall genau
   // zehnmal durchlaufen.
}
```


andre.maier@elektronikschule.de

Bewertung der Schleifen

- Eine do...while-Schleife wird mindestens einmal durchlaufen
- for-Schleife soll nur dann eingesetzt werden, wenn aus dem Schleifenkopf genau ersichtlich ist, wie oft die Schleife durchläuft.
- Die for-Schleife ist für die Schülerinnen und Schüler am anspruchsvollsten, die while-Schleife ist syntaktisch und semantisch am einfachsten.
- Auf Maschinencode-Ebene gibt es keine Schleifenkonstrukte, nur bedingte und unbedingte Sprünge.
- Die Funktionalität von Mehrfachverzweigungen kann manchmal auch eleganter umgesetzt werden.

andre.maier@elektronikschule.de

Sprungbefehle

- *break*
  - Verlässt die Kontrollstruktur und setzt das Programm nach der Kontrollstruktur fort
- *continue*
  - Überspringt den restlichen (nachfolgenden) Code innerhalb eines Schleifenkörpers und setzt Ausführung im Schleifenkopf fort
- *goto*
  - Unbedingter Sprung zu einer Sprungmarke

andre.maier@elektronikschule.de

Die “if-Schleife” ;-)

```
        int zaehler=0;
marke:  printf("Zaehlerstand:%d\n",zaehler);
        zaehler++;
        if(zaehler<10)
           goto marke;
        printf("Ende.\n");
```


andre.maier@elektronikschule.de

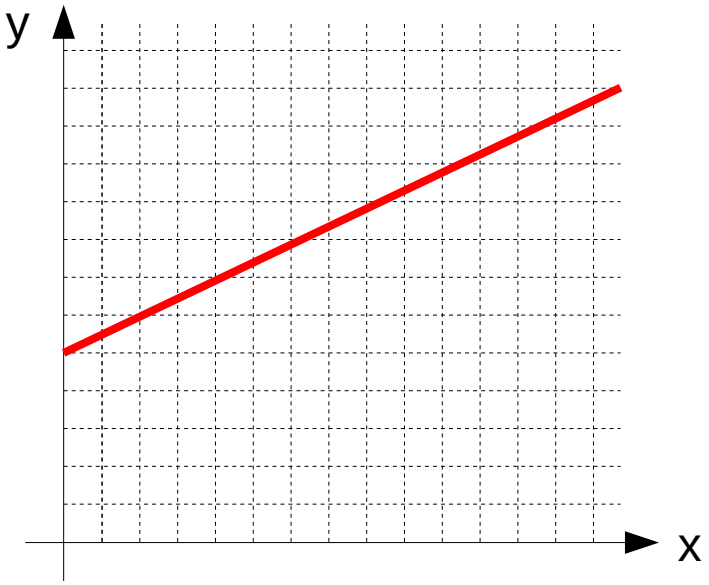
Bewertung der Sprungbefehle

- Eine unnötige Verwendung von Sprungbefehlen wird i.d.R. als schlechter Programmierstil betrachtet, da sie die Lesbarkeit des Sourcecodes verschlechtern.
- *goto* ist in der strukturierten Programmierung unnötig und soll möglichst nicht verwendet werden.

andre.maier@elektronikschule.de

Funktionen (1)

- Definition aus der Sicht von Mathelehrern



Name der Funktion

Parameter

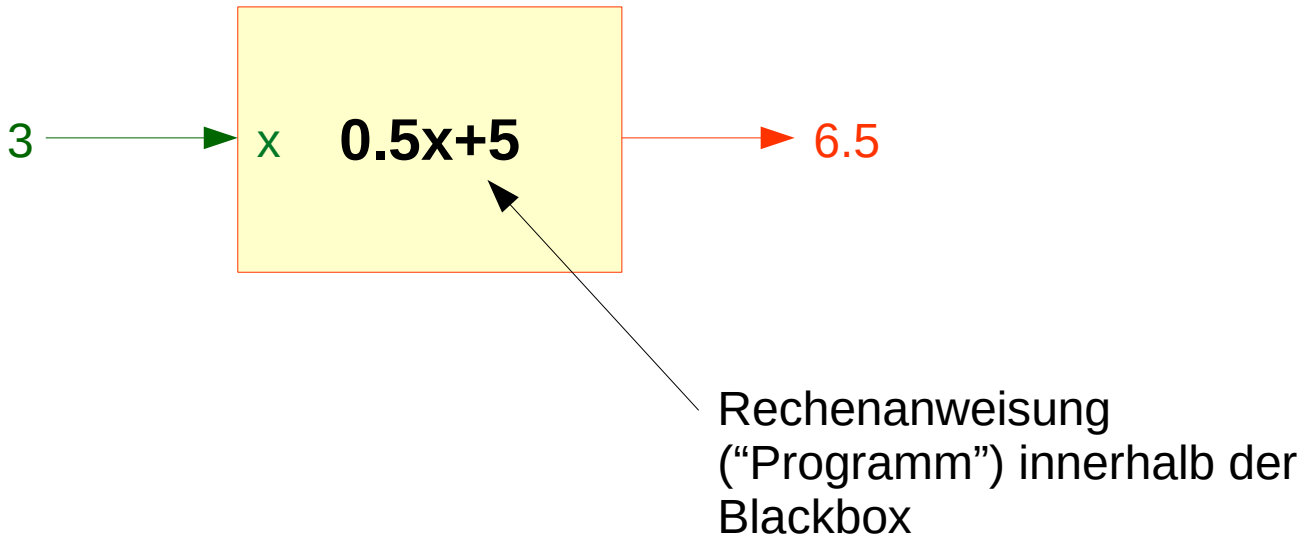
$$y = f(x) = 0.5x+5$$

Gesamter Ausdruck “verkörpert” an dieser Stelle den Ergebniswert für einen bestimmten eingesetzten Wert von x

andre.maier@elektronikschule.de

Funktionen (2)

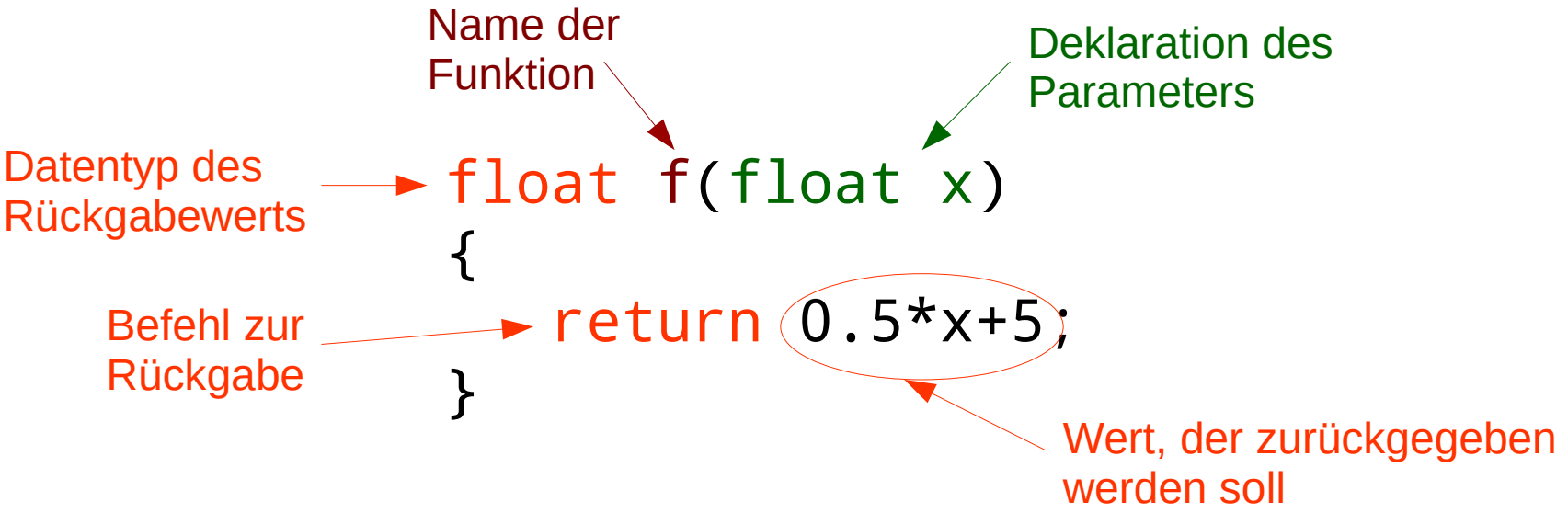
- Betrachtung als “Blackbox”



andre.maier@elektronikschule.de

Funktionen (3)

- Definition aus der Sicht von C-Programmierern



andre.maier@elektronikschule.de

Funktionen (4)

- Funktionen können ...
  - ... keinen, einen, oder mehrere Parameter haben.

```
float berechne_volumen(float laenge, float breite, float hoehe)
```

  - ... keinen oder nur maximal einen einzigen Wert als direkten Rückgabewert haben.
- Das Schlüsselwort *void* bezeichnet den “leeren Datentypen” und bedeutet soviel wie “Nichts”.

andre.maier@elektronikschule.de

Funktionen (5)

- Aufruf (=Verwendung) von Funktionen

```
float ergebnis;
float breite = 1.5;

ergebnis = berechne_volumen(3.5, breite, 2.8);
```

Argumente, die als Parameter in die Funktion übergeben werden. Hier müssen entweder konkrete Werte (Literale) oder Variablen stehen, welche die zu übergebenden Werte enthalten.

Gesamter Funktionsaufruf “wirkt” an dieser Stelle wie der Ergebniswert!

andre.maier@elektronikschule.de

Funktionsprototypen (1)

- Nur bei moderneren Varianten von C kann die Definition einer Funktion auch erst nach dem ersten Aufruf im Sourcecode stehen.
- In älteren Versionen von C, z.B. ANSI-C, muss vor dem Funktionsaufruf zumindest der Funktionsprototyp a.k.a. Funktionskopf genannt worden sein.

andre.maier@elektronikschule.de

Funktionsprototypen (2)

```
#include <stdio.h>

int addiere(int a, int b);

int main()
{
   int c = addiere(1, 2);
   printf("1 + 2 = %d\n",c);
}

int addiere(int a, int b)
{
   return a + b;
}
```

Funktionsprototyp
Hinweis: Die Nennung der Parameter können hier auch verkürzt oder weggelassen werden. D.h. sowohl *int addiere()* als auch *int addiere(int,int)* wären hier ebenfalls möglich.

andre.maier@elektronikschule.de

Gleichnamige Funktionen

- Bei gleichnamigen Funktionen ordnet der Compiler die Aufrufe über die Datentypen der Parameterliste zu.
- Folglich können keine gleichnamigen Funktionen mit exakt der selben Parameterliste (gleiche Datentypen in der gleichen Reihenfolge) definiert werden.

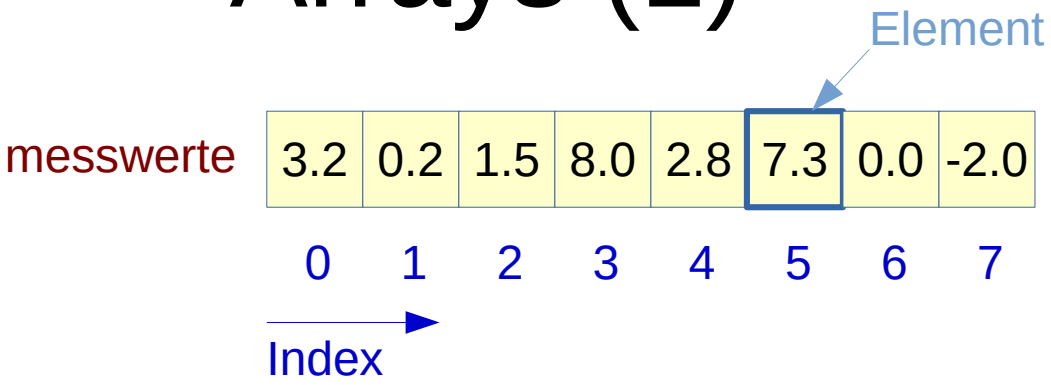
andre.maier@elektronikschule.de

Komplexere Datentypen

- Arrays
  - Zeichenketten (Strings)
- Strukturen (structs)

andre.maier@elektronikschule.de

Arrays (1)



Erzeugung des Array (Deklaration und literale Initialisierung):

```
float messwerte[] = {3.2, 0.2, 1.5, 8.0, 2.8, 7.3, 0.0, -2.0};
```

Erzeugung des Array (nur Deklaration):

```
float messwerte[8];
```

Anzahl der Elemente

andre.maier@elektronikschule.de

Arrays (2)

- Zugriff auf die Elemente des Arrays
  - Schreibend
    z.B. `messwerte[3]=-3.4;`
  - Lesend
    z.B. `summe = messwerte[0] + messwerte[1];`

andre.maier@elektronikschule.de

Arrays (3)

- Mehrdimensionale Arrays
  - Deklaration und literale Initialisierung

```
int a2d[][2] = {{1,2},
                {3,4}};
```

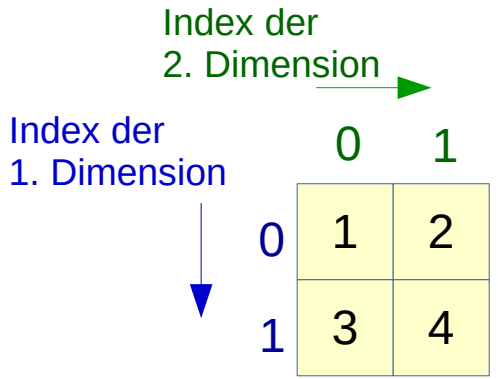
  - Nur Deklaration

```
int a2d[2][2];
```

  - Zugriff auf Elemente

```
a2d[0][1] = 42;
```

Hinweis: Bei mehrdimensionalen Arrays muss, ausser bei der ersten Dimension, die Anzahl der Elemente der weiteren Dimensionen angegeben werden.



andre.maier@elektronikschule.de

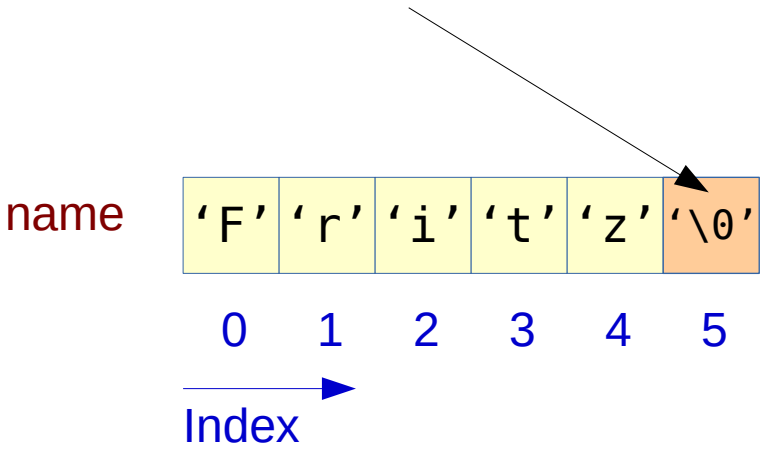
Zeichenketten in C (1)

- Zeichenketten werden in C mithilfe von eindimensionalen Arrays von Zeichen (Datentyp char) dargestellt.
- Das Ende einer Zeichenkette wird durch den ASCII-Wert 0 gekennzeichnet.

andre.maier@elektronikschule.de

Zeichenketten in C (2)

“Backslash-Null” ist das Zeichen-Literal für den ASCII-Wert 0. Es könnte auch einfach nur der Zahlenwert 0 (ohne die einfachen Hochkomma) ins Array geschrieben werden.



Erzeugen der Zeichenkette:

```
char name[]={‘F’,’r’,’i’,’t’,’z’,’\0’};
```

oder

```
char name[]=“Fritz”;
```


andre.maier@elektronikschule.de

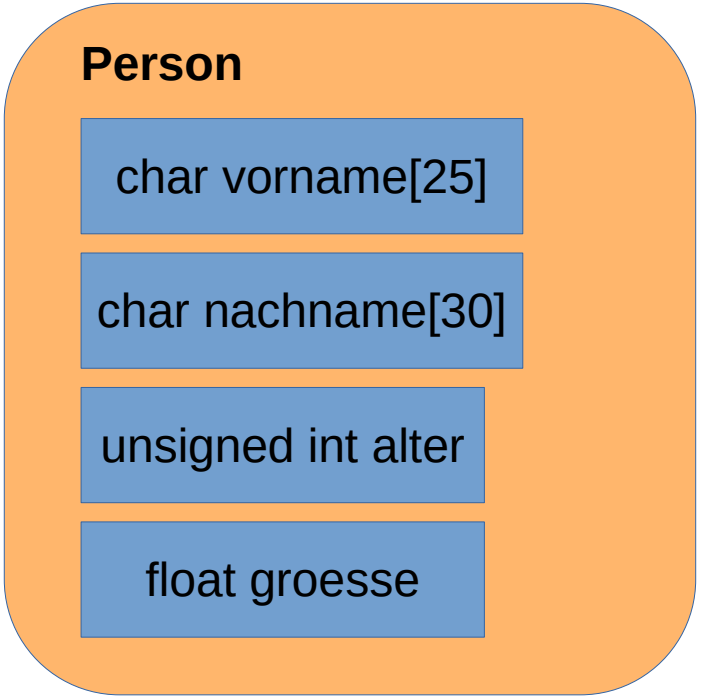
Zeichenketten in C (3)

- Um mit Zeichenketten zu arbeiten, stellt C neben den Zeichenketten-Literalen weitere Hilfsmittel zur Verfügung.
- Beispiele
  - Formatbezeichner %s für printf
    ```
    printf(“%s”, name);
    ```
  - Umfangreiche Funktionsbibliotheken, z.B. *strings.h*

andre.maier@elektronikschule.de

Strukturen / structs (1)

- “Datencontainer”, die mehrere Variablen gleicher oder unterschiedlicher Datentypen enthalten können



Strukturen / structs (2)

- Definition von structs

Name der Struktur

```
struct punkt {
   int x_pos;
   int y_pos;
};
```

- Deklaration einer Variablen, welche die Struktur enthalten kann.

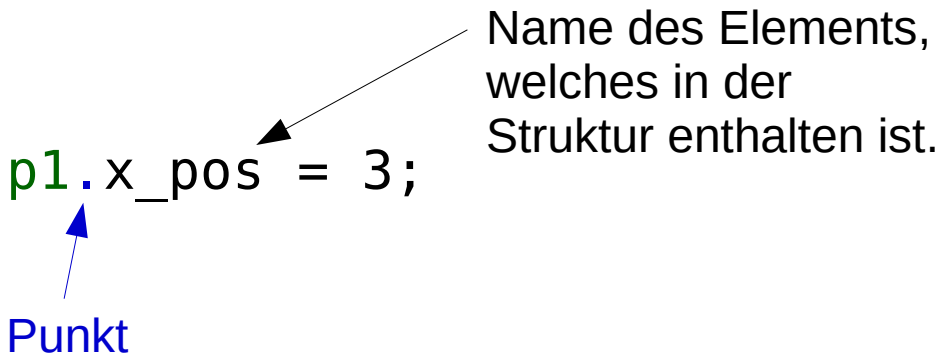
Name der Variablen

```
struct punkt p1;
```


andre.maier@elektronikschule.de

Strukturen / structs (3)

- Zugriff auf Variablen, die eine Struktur enthalten.



andre.maier@elektronikschule.de

Strukturen / structs (5)

- Kombinierte Definition und Deklaration von Variablen

```
struct punkt {
   int x_pos;
   int y_pos;
} p1, p2, p3;
```

Häufig werden eine oder mehrere Variablen für die struct gleich direkt nach der Defintion deklariert.

- Erzeugung eines eigenen Datentyps für die Struktur

```
typedef struct punkt Point2D;
```

Eigener Datentyp, mit dem dann direkt neue Variablen deklariert werden können, z.B. `Point2D p4;`

andre.maier@elektronikschule.de

Pointer (1)

- Hinter jeder Variable verbirgt sich eine Speicheradresse im RAM, an der sich der Inhalt der Variablen befindet.
- Der C-Programmierer hat keinen direkten Einfluss darauf, an welcher absoluten Adresse eine Variable genau angelegt wird.

```
char c=42;
```

Adresse	Inhalt
...	
0815	
0816	
0817	42
0818	
0819	
082A	
082B	
082C	
082D	
...	

andre.maier@elektronikschule.de

Pointer (2)

- Allerdings kann in C der Speicherort einer Variablen ausgelesen und mit ihm gearbeitet werden.
- Eine Variable, die nicht den konkreten Wert, sondern nur einen Speicherort eines konkreten Werts enthält, nennt man Pointer (veraltet auch “Zeiger”).

andre.maier@elektronikschule.de

Pointer (3)

Name des Pointers

```
char c=42;

char *pointer_auf_c=&c;
```

Deklariert wird ein Pointer immer mit dem Datentypen des Werts auf den er zeigt. Vor dem Namen muss bei der Deklaration ein * stehen.

Lies: “Char-Pointer” oder “Pointer auf char”

Adressoperator

Lies: “Adresse von ...”

	Adresse	Inhalt
	...	
	0815	
	0816	
c	0817	42
	0818	
	0819	
	082A	
	082B	
	082C	
pointer_auf_c	082D	0817
	...	

andre.maier@elektronikschule.de

Pointer (4)

- Pointer lassen sich inkrementieren und dekrementieren, z.B. mit `pointer_auf_c++;`
- Dabei erhöht sich die Adresse immer abhängig vom Datentypen (z.B. um vier bei 4-Byte Datentypen.
- Wie “lange” ein Datentyp ist, lässt sich mithilfe des Operators sizeof() herausfinden, z.B. `sizeof(int)`

andre.maier@elektronikschule.de

Pointer (5)

- Wenn über einen Pointer auf den Wert, auf den er verweist, zugegriffen wird, spricht man vom **Dereferenzieren** des Pointers.
- Das Dereferenzieren erfolgt mithilfe eines * vor dem Pointernamen.

```
*pointer_auf_c=41;
```

Lies: “Inhalt an der Stelle, auf die pointer_auf_c zeigt.”

andre.maier@elektronikschule.de

Pointer (6)

- Funktionen können anstelle von Werten auch Pointer als Parameter haben.
- Sie erwarten dann die Übergabe einer Speicheradresse.
- Vorteil: Die Funktion kann die übergebene Speicherstelle sowohl lesen als auch den dort hinterlegten Wert direkt ändern.
- Dieses Verfahren nennt man **call-by-reference** (im Gegensatz zu call-by-value bei direkter Übergabe von Werten).

andre.maier@elektronikschule.de

Pointer (7)

```
#include <stdio.h>

void verdopple(int *n)
{
  *n = *n * 2;
}

int main()
{
  int wert = 3;
  verdopple(&wert);
  printf("%d\n",wert);
}
```

- Der Funktion *verdopple()* wird der Speicherort der Variablen wert übergeben.
- In der Funktion *verdopple()* wird der Pointer dereferenziert, der Wert verdoppelt und das Ergebnis genau an der selben Speicherstelle gespeichert.
- Die Funktion benötigt nun keinen Rückgabewert mehr!

andre.maier@elektronikschule.de

Pointer (8)

- Arrays (z.B. Zeichenketten) können nur per “call-by-reference” als Funktionsparameter übergeben werden. Es wird dabei stets der Speicherort des ersten Elements übergeben.

```
NAME
       strcmp, strncmp - compare two strings

SYNOPSIS
       #include <string.h>

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

       int strncmp(const char *s1, const char *s2, size_t n);

DESCRIPTION
       The  strcmp()  function compares the two strings s1 and s2.  It returns
       an integer less than, equal to, or greater than zero if  s1  is  found,
       respectively, to be less than, to match, or be greater than s2.
```

Ein schönes Beispiel aus der Praxis: Die Bibliotheksfunktion *strcmp* aus *string.h*, die eine Zeichenkette *s1* mit einer anderen Zeichenkette *s2* vergleicht.

andre.maier@elektronikschule.de

Pointer (9)

- Was man sonst noch so alles mit Pointern anstellen kann, aber über die grundlegenden Basics hinausgeht, siehe hier ...
  - https://bitjunkie.wordpress.com/?s=pointer

andre.maier@elektronikschule.de

Pointer auf structs

- Grundsätzlich gilt das selbe Prinzip wie bei Pointern auf Variablen “normaler” Datentypen.

```
struct punkt p1;
struct punkt *pointer_auf_p1 = &p1;
```

- Mithilfe des **->** Operators können über den Pointer auf die Inhalte der in der struct enthaltenen Variablen zugegriffen werden.

```
pointer_auf_p1->x_pos = 3;
pointer_auf_p1->y_pos = 8;
```


andre.maier@elektronikschule.de

Eine kleine Einführung in OOP

- OOP = **O**bjekt**o**rientierte **P**rogrammierung
- Völlig anderer Denkansatz als in der prozeduralen Programmierung
- Programmierer orientiert sich nicht mehr an der “denkweise” der Maschine, sondern an der realen Welt.

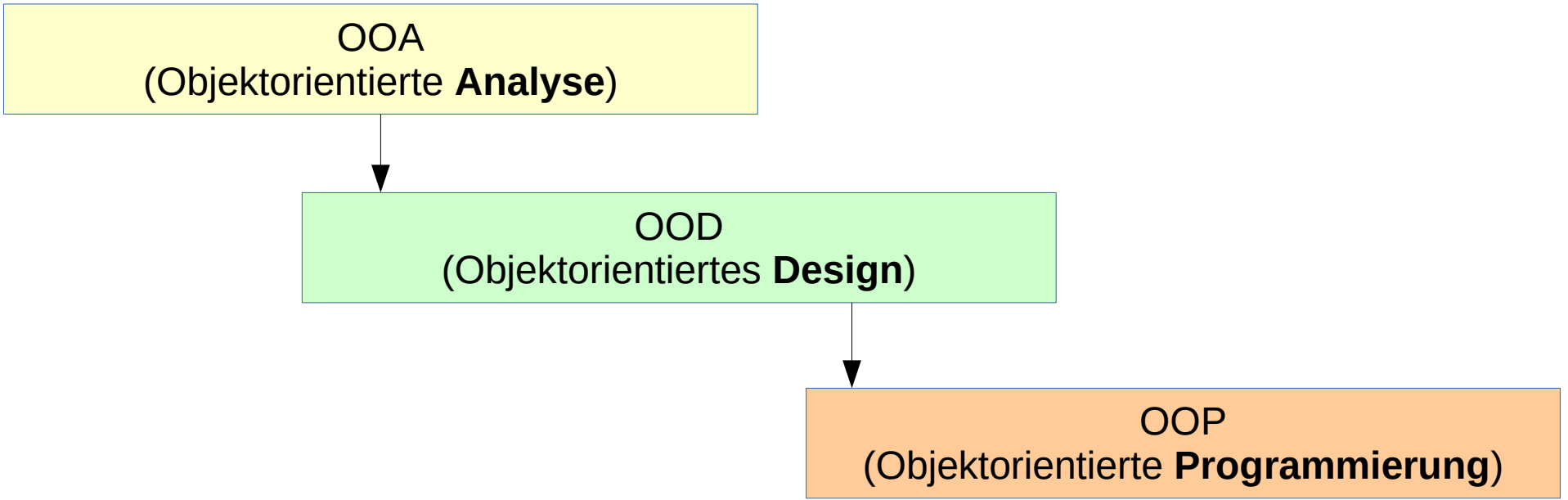
andre.maier@elektronikschule.de

In der realen Welt ...

- ... gibt es Objekte, die ganz bestimmte Eigenschaften und Fähigkeiten haben.
- Objekte haben Gemeinsamkeiten und Unterschiede, sie können untereinander interagieren.
- Ähnliche Objekte können durch (abstrakte) Überbegriffe beschrieben werden.
- u.v.m.

andre.maier@elektronikschule.de

Entwicklung von OO-Software



andre.maier@elektronikschule.de

Objekt vs. Klasse

- Eine Klasse ist ein Bauplan (“Blaupause”) für Objekte. Dieser muss immer zuerst entwickelt werden.
- Mit der Klasse beschreiben wir also einen Bauplan, aus dem beliebig viele konkrete Objekte “gebaut” werden können.
- Objekte nennt man auch Instanzen einer Klasse, das “Bauen” der Objekte heisst in der Fachsprache “instanzieren”.

andre.maier@elektronikschule.de

Ein Beispiel (1)

Eigene Darstellung

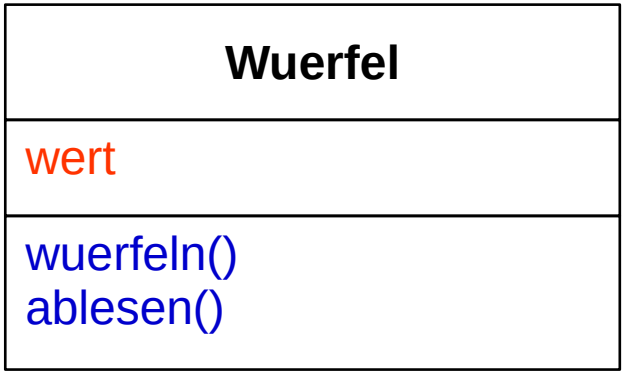
- “Was kann man mit dem Objekt tun?”
  - Der Würfel kann gewürfelt werden.
  - Man kann den gewürfelten Wert ablesen.
- “Welche Eigenschaft(en) verändern sich, wenn das Objekt “bedient” wird?”
  - Es ändert sich der Wert, der oben liegt, also der gewürfelte Wert.

andre.maier@elektronikschule.de

Ein Beispiel (2)

- Veränderbare Eigenschaften (Attribute)
  - Aktueller Wert des Würfels
- “Bedienmöglichkeiten” des Würfels (Methoden)
  - Wuerfeln
  - Ablesen

UML-Klassendiagramm (vereinfacht) für den Würfel

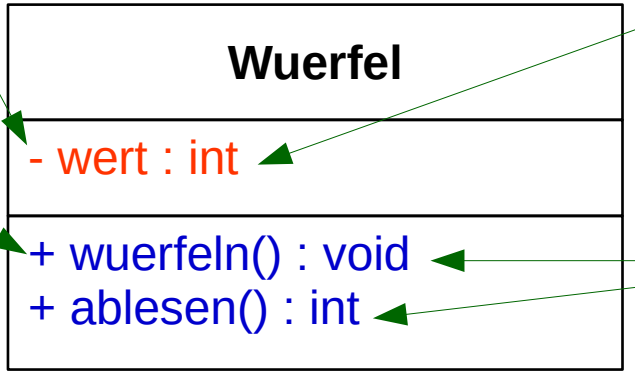


andre.maier@elektronikschule.de

Ein Beispiel (3)

Zugriffsmodifizierer “**private**”: Nur durch die Methoden dieser Klasse veränderbar.

Datentyp der Eigenschaft



Zugriffsmodifizierer “**public**”: Von ausserhalb der Klasse aus verwendbar.

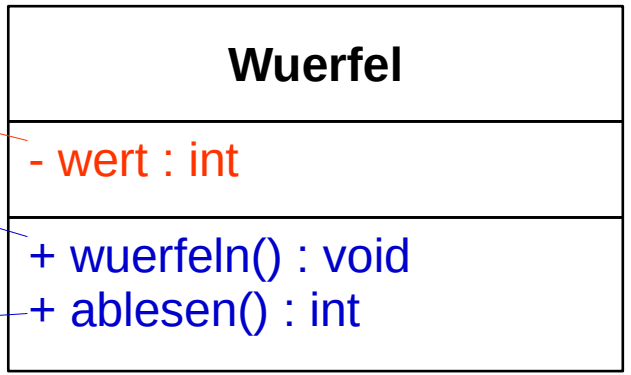
Rückgabedatentyp der Methoden

andre.maier@elektronikschule.de

Klassendefinition in C++

```
class Wuerfel {

   private:
      int wert;
   public:
      void wuerfeln()
      {
        srand(time(NULL));
        wert = 1+rand()%6;
      }
      int ablesen()
      {
         return wert;
      }
};
```



Verwendung der Klasse in C++ (2)

```
int main(...) {

    Wuerfel w1;

    w1.wuerfeln();

    int ergebnis=w1.ablesen();

    ...

    return 0;
}
```

Instanzierung.
Lies: “Baue ein Objekt namens w1 aus einer Klasse namens Wuerfel.”

Aufruf der Methode wuerfeln().
Lies: “Hey, w1 ... wuerfle Dich!”

Aufruf der Methode ablesen().
Lies: “Hey, w1 ... welchen Wert hast Du?”
Der Methodenaufruf “wirkt” an dieser Stelle wie sein Rückgabewert, d.h. wir können ihn deshalb direkt der Variablen ergebnis zuweisen.

andre.maier@elektronikschule.de

The End

- Bewusst nicht behandelte Themen, u.a.
  - Unions
  - Pointer auf Funktionen
  - Dynamisches Speichermanagement
  - Genauer Inhalt von Bibliotheken bei gcc, MinGW, Arduino
  - Erstellen eigener vorcompilierter Bibliotheken
  - Erweiterte Funktionen der IDE
- Fragen
- Feedback-Runde
- Teilnahmebscheinigungen

andre.maier@elektronikschule.de