

PROGRAMMIEREN I

WS 2022

Prof. Dr.-Ing. Kolja Eger
Hochschule für Angewandte Wissenschaften Hamburg

Check-In



Präsenzklausur im Labor (180min)

Dynamische Speicherallokation

Praktikum 7

Strukturen

Praktikum 6

Dateien

Praktikum 5

Zeiger

Praktikum 4

Vektoren (Arrays)

Praktikum 3

Funktionen

Praktikum 2

Kontrollstrukturen

Datentypen & Operatoren

Erste Programme

Praktikum 1

Unser Weg durch das Semester

Was machen wir heute?

- (Nochmal) Zahlensysteme
- Bitoperatoren
- Quellcode auf mehrere Dateien verteilen
- Preprozessor-Befehle
- Call-by-Reference

ZAHLENSYSTEME

Zahlensysteme - Wiederholung

- Was ist ein Zahlensystem?
- Welche Zahlensysteme sind in der Computertechnik gebräuchlich?
- Wie erfolgt eine Umrechnung ins Dezimalsystem?
- Welche Dezimalzahl ist 10101?
- Welche Dezimalzahl ist A4?

Dual → Hexadezimal

Umrechnung von dual in hexadezimal Zahlen

- 4 Ziffern des dualen Zahlensystems lassen sich zu einer hexadezimal Ziffer zusammenfassen
- Wichtig ist, dass man vom Dezimalkomma ausgehend umwandelt (und nicht von der Ziffer mit der höchsten Gewichtung)
- Blöcke von 4 Bits lassen sich gemäß folgender Tabelle in eine hexadezimale Ziffer umwandeln
- Beispiel:

dual	hexadezimal	dual	hexadezimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

$$\begin{aligned} 10010101100_2 &= 0100 \quad 1010 \quad 1100 \\ &= \quad 4 \quad \quad A \quad \quad C \\ &= 4AC_{16} \end{aligned}$$

Hexadezimal → Dual

- Für die Umrechnung in anderer Richtung lässt sich die Tabelle ebenfalls nutzen

- Beispiel:

$$\begin{aligned} 1D4,6_{16} &= 0001\ 1101\ 0100, 0110 \\ &= 111010100,011 \end{aligned}$$

dual	hexadezimal	dual	hexadezimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Analog erfolgt die Umrechnung von/in Oktalzahlen mit Blöcken von 3 Bits!

Dezimal \rightarrow Binär

Umrechnung aus dem Dezimalsystem ins Binärsystem

- Beispiel: 13_{10}

$$13 : 2 = 6 \text{ Rest } 1$$

$$6 : 2 = 3 \text{ Rest } 0$$

$$3 : 2 = 1 \text{ Rest } 1$$

$$1 : 2 = 0 \text{ Rest } 1$$



- Ergebnis

$$13_{10} = 1101_2$$

Dezimal \rightarrow Hexadezimal

- Beispiel: $6671_{10} \rightarrow \text{hex}$

$$6671 : 16 = 416 \quad \text{Rest } 15 \text{ (F)}$$

$$416 : 16 = 26 \quad \text{Rest } 0$$

$$26 : 16 = 1 \quad \text{Rest } 10 \text{ (A)}$$

$$1 : 16 = 0 \quad \text{Rest } 1$$



- Ergebnis

$$6671_{10} = 1A0F_{16}$$

Umrechnung des ganzzahligen Teils in ein beliebiges Zahlensystem

- Beispiel: $6671_{10} \rightarrow \text{hex}$

$$6671 : 16 = 416 \text{ Rest } 15 \text{ (F)}$$

$$416 : 16 = 26 \text{ Rest } 0$$

$$26 : 16 = 1 \text{ Rest } 10 \text{ (A)}$$

$$1 : 16 = 0 \text{ Rest } 1$$

- Ergebnis

$$6671_{10} = 1A0F_{16}$$

- Man führe eine ganzzahlige Division der Dezimalzahl durch die Basis des Zielsystems durch
- Der Rest der Division stellt den Wert der kleinsten ganzzahligen Ziffer im neuen Zahlensystem dar
- Der Quotient dient als Dividend zur Berechnung der nächsten Ziffer nach dem gleichen Schema
- Der Rest der zweiten Division wird links vor die zuvor berechnete Ziffer des ersten Rests geschrieben
- Es werden weitere Divisionen durchgeführt bis der Quotient null ist

Umrechnung des ganzzahligen Teils in ein beliebiges Zahlensystem

- Beispiel: $6671_{10} \rightarrow \text{hex}$

$$6671 : 16 = 416 \text{ Rest } 15 \text{ (F)}$$

$$416 : 16 = 26 \text{ Rest } 0$$

$$26 : 16 = 1 \text{ Rest } 10 \text{ (A)}$$

$$1 : 16 = 0 \text{ Rest } 1$$

- Ergebnis

$$6671_{10} = 1A0F_{16}$$

- Man führe eine ganzzahlige Division der Dezimalzahl durch die Basis des Zielsystems durch
- Der Rest der Division stellt den Wert der kleinsten ganzzahligen Ziffer im neuen Zahlensystem dar
- Der Quotient dient als Dividend zur Berechnung der nächsten Ziffer nach dem gleichen Schema
- Der Rest der zweiten Division wird links vor die zuvor berechnete Ziffer des ersten Rests geschrieben
- Es werden weitere Divisionen durchgeführt bis der Quotient null ist

Dezimal \rightarrow Oktal

- Übung: $668_{10} \rightarrow$ Oktal ?

Umrechnung des Nachkommateils in ein beliebiges Zahlensystem

Dezimal → Dual

Beispiel: $0,375_{10} \rightarrow \text{dual}$

$$\begin{array}{ll} 0,375 \cdot 2 = 0,75 & \text{ganzzahliger Teil: 0} \\ 0,75 \cdot 2 = 1,5 & \text{ganzzahliger Teil: 1} \\ 0,5 \cdot 2 = 1 & \text{ganzzahliger Teil: 1} \end{array} \downarrow = 0,011_2$$

- Man multipliziere den Nachkommateil mit der Basis des Zielsystems.
- Der ganzzahlige Teil ergibt die erste Ziffer nach dem Komma.
- Der Nachkommateil des zuvor berechneten Produktes dient als Ausgangspunkt für die Berechnung der nächsten Ziffer.
- Dies wird fortgeführt, bis die Umrechnung abgeschlossen
- Oder die gewünschte Genauigkeit erreicht ist (da der Nachkommateil häufig auch nach vielen berechneten Ziffern nicht auf Null geht)

Umrechnung des Nachkommateils in ein beliebiges Zahlensystem

Dezimal → Dual

Beispiel: $0,375_{10} \rightarrow \text{dual}$

$$\begin{array}{ll} 0,375 \cdot 2 = 0,75 & \text{ganzzahliger Teil: } 0 \\ 0,75 \cdot 2 = 1,5 & \text{ganzzahliger Teil: } 1 \\ 0,5 \cdot 2 = 1 & \text{ganzzahliger Teil: } 1 \end{array} \quad \downarrow = 0,011_2$$

- Man multipliziert den Nachkommateil mit der Basis des Zielsystems.
- Der ganzzahlige Teil ergibt die erste Ziffer nach dem Komma.
- Der Nachkommateil des zuvor berechneten Produktes dient als Ausgangspunkt für die Berechnung der nächsten Ziffer.
- Dies wird fortgeführt, bis die Umrechnung abgeschlossen
- Oder die gewünschte Genauigkeit erreicht ist (da der Nachkommateil häufig auch nach vielen berechneten Ziffern nicht auf Null geht)

Umwandlung in Visual Studio

Weg 1

```
void dezimal2hex(unsigned dezimal)
{
    unsigned Tmp[8]; // Länge des unsigned Datentyp 32bit --> max. 8 Hex-Zahlen
    int i = 0;

    if (dezimal == 0)
        printf("0");

    // Schritt 1: Ergebnisse der Division in Tmp zwischenspeichern
    while (dezimal) {
        Tmp[i++] = dezimal;
        dezimal /= 16;
    }

    // Schritt 2: Rest mit % berechnen und ausgeben in umgekehrter Reihenfolge
    while (i) {
        i--;
        printf("%c", (Tmp[i] % 16) <= 9 ? '0' + (Tmp[i] % 16) : 'A' - 10 + (Tmp[i] % 16));
    }
}
```

Umwandlung in Visual Studio (II)

Weg 2

```
// Dezimal --> Hex
int z = 6671;
printf("%d = %X\n", z, z);
```

Typ	Datentyp	Darstellung
%d oder %i	signed int	dezimal
%u	unsigned int	dezimal
%o	unsigned int	oktal
%x	unsigned int	hexadezimal (mit kleinen Buchstaben)
%X	unsigned int	hexadezimal (mit großen Buchstaben)
%f	float	immer ohne Exponent
%e	float	immer mit Exponent (durch 'e' angedeutet)
%E	float	immer mit Exponent (durch 'E' angedeutet)
%g	float	nach Bedarf mit Exponent (durch 'e' angedeutet)
%G	float	nach Bedarf mit Exponent (durch 'E' angedeutet)
%c	char	einzelnes Zeichen
%s	char[]	Zeichenkette (String)
%p	void *	Speicheradresse
%n	signed int *	schreibt die Anzahl der bisherigen Zeichen an die angegebene Adresse
%%	-	Ausgabe des Zeichens '%'

Tabelle A.1: Der *Typ* im Formatstring für die elementaren Datentypen.

Übungsaufgabe

- Schreiben Sie eine Funktion, mit der eine Dezimalzahl in ein Zahlensystem mit beliebiger Basis umgewandelt werden kann!
- Nutzen Sie *dezimal2hex()* als Vorlage.
- Nutzen Sie folgende Funktionsdeklaration

```
void dezimalUmwandlung(unsigned dezimal, unsigned basis);
```

- Testen Sie ihre Funktion mit den vorherigen Beispielen.
- Optionale Aufgabe:
 - Geben Sie in einer Tabelle die Zahlen von 0 bis 20 in dezimal, binär, oktal, hexadezimal, terzial (3) und septal (7) aus (vgl. Tabelle 1.1 im Skript von Prof. R. Heß bzw. Tabelle links & n. Slide)

dezimal	dual/binär	oktal	hexadezimal	terzial	septal
Basis 10	Basis 2	Basis 8	Basis 16	Basis 3	Basis 7
0	0	0	0	0	0
1	1	1	1	1	1
2	10	2	2	2	2
3	11	3	3	10	3
4	100	4	4	11	4
5	101	5	5	12	5
6	110	6	6	20	6
7	111	7	7	21	10
8	1000	10	8	22	11
9	1001	11	9	100	12
10	1010	12	A	101	13
11	1011	13	B	102	14
12	1100	14	C	110	15
13	1101	15	D	111	16
14	1110	16	E	112	20
15	1111	17	F	120	21
16	10000	20	10	121	22
17	10001	21	11	122	23
18	10010	22	12	200	24
19	10011	23	13	201	25
20	10100	24	14	202	26

Tabelle 1.1: Die Zahlen von null bis zwanzig in verschiedenen Zahlensystemen.

dezimal	dual/binär	oktal	hexadezimal	terzial	septal
Basis 10	Basis 2	Basis 8	Basis 16	Basis 3	Basis 7
0	0	0	0	0	0
1	1	1	1	1	1
2	10	2	2	2	2
3	11	3	3	10	3
4	100	4	4	11	4
5	101	5	5	12	5
6	110	6	6	20	6
7	111	7	7	21	10
8	1000	10	8	22	11
9	1001	11	9	100	12
10	1010	12	A	101	13
11	1011	13	B	102	14
12	1100	14	C	110	15
13	1101	15	D	111	16
14	1110	16	E	112	20
15	1111	17	F	120	21
16	10000	20	10	121	22
17	10001	21	11	122	23
18	10010	22	12	200	24
19	10011	23	13	201	25
20	10100	24	14	202	26

Tabelle 1.1: Die Zahlen von null bis zwanzig in verschiedenen Zahlensystemen.

Negative Zahlen

- Bei vorzeichenbehafteten Zahlen steht das erste Bit für das Vorzeichen
 - 0 für positive Zahlen
 - 1 für negative Zahlen
- Ohne zusätzliche Änderungen ergebe sich eine Redundanz für die Null (+0 und -0)
- U.a. deswegen wird eine spezielle Darstellung für negative Zahlen gewählt

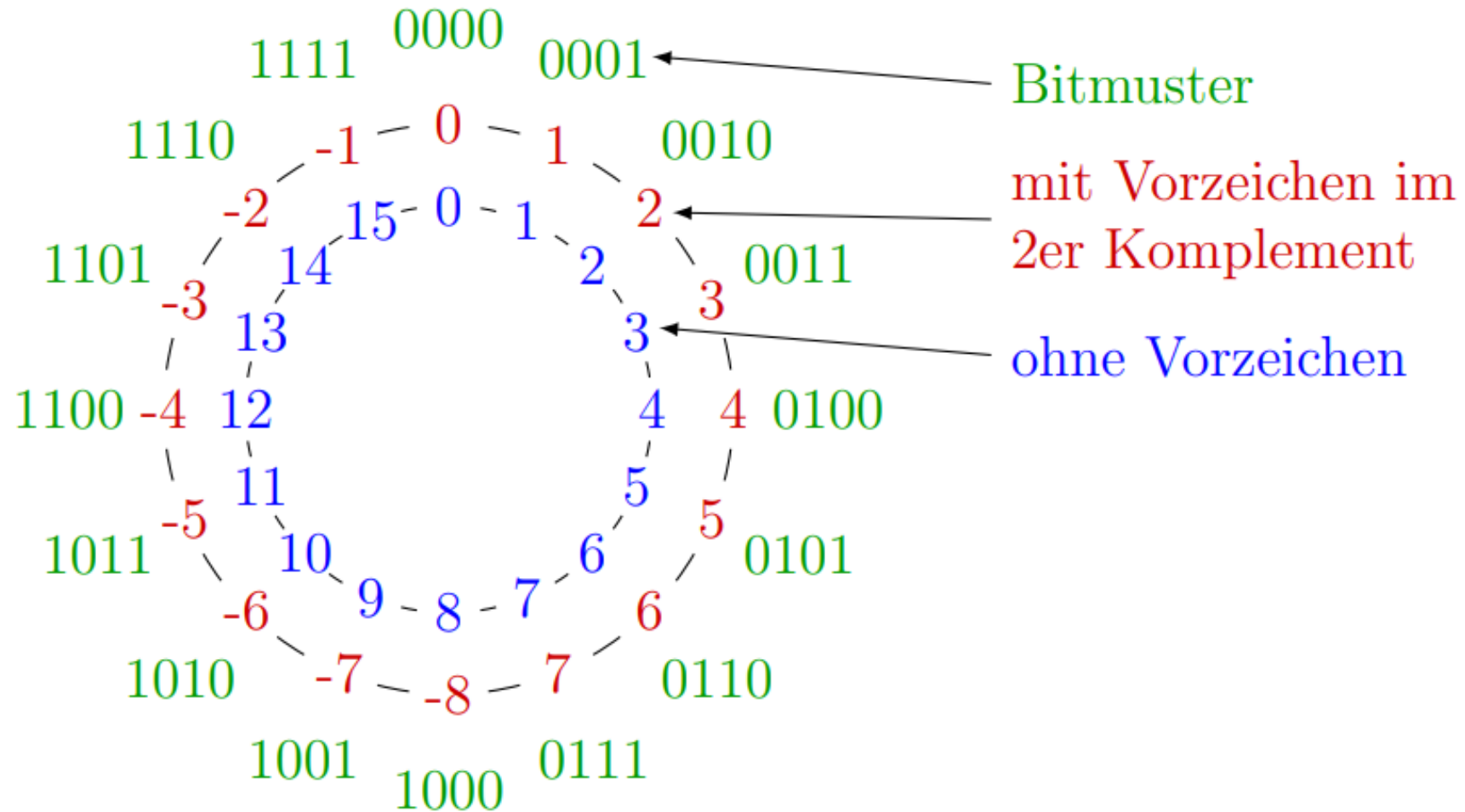
→ Das **Zweierkomplement**

- Positive Zahlen werden wie gewohnt mit dualen Zahlen angegeben
- Für negative Zahlen wird das Zweierkomplement gebildet
 - **Erst werden alle Bits negiert** bzw. umgekehrt (aus Einsen werden Nullen und aus Nullen werden Einsen), **danach die Zahl um Eins erhöht**

Zweierkomplement- Beispiel

- Beispiel: 8-Bit Zahlen im Zweierkomplement
 - $-4_{10} = -0000\ 0100_2 \rightarrow 1111\ 1011 \rightarrow 1111\ 1100$
 - $-75_{10} = -0100\ 1011_2 \rightarrow 1011\ 0100 \rightarrow 1011\ 0101$
- Dual \rightarrow dezimal:
 - Wieder müssen alle Bits negiert werden, dann die Zahl um Eins erhöhen
 - $1111\ 1100_2 \rightarrow 0000\ 0011 \rightarrow -0000\ 0100 \rightarrow -4_{10}$
 - $1011\ 0101_2 \rightarrow 0100\ 1010 \rightarrow -0100\ 1011 \rightarrow -75_{10}$

Darstellung von positiven und negativen Zahlen für vier Bit



BITOPERATOREN & BITMANIPULATION

Logikoperatoren VS. Bitoperatoren

		UND
0	0	0
0	1	0
1	0	0
1	1	1

Logische UND-Verknüpfung

- Operator: &&
- Logikoperatoren arbeiten mit *‚wahr‘* und *‚nicht wahr‘*
- Zahlen werden wie folgt interpretiert
 - Nur null ist *‚nicht wahr‘*
 - Alle anderen Zahlen sind *‚wahr‘*

Beispiel:

$0 \&\& 1 \rightarrow \text{nicht wahr (0)}$

$5 \&\& 6 \rightarrow \text{wahr (1)}$

Bitweise UND-Verknüpfung

- Operator: &
- Zahlen werden als binär Zahlen interpretiert
- Verknüpfung erfolgt bitweise, d.h. für jedes Bit einzeln

Beispiel:

$0 \& 1 \rightarrow 0$

$5 \& 6 \rightarrow 0101 \& 0110 \rightarrow 0100 \rightarrow 4$

Bitmanipulation

Bitoperatoren	
~	Bit-Komplement (unärer Operator)
&	Bitweise UND-Verknüpfung
	Bitweise ODER-Verknüpfung
^	Bitweise Exklusiv-ODER-Verknüpfung
<<	Bit-Verschiebung nach links
>>	Bit-Verschiebung nach rechts

- Operatoren zur Bitmanipulation können auf alle ganzzahligen Datentypen angewendet werden (char, short, int und long, signed wie unsigned)

Bit-Komplement ~

- Unärer Operator
- Bits werden umgekippt
 - aus 1010 wird z.B. 0101
- Code-Beispiel

Ausgabe?

```
// Beispiel unärer Operator
unsigned char c1, c2;
c1 = 0x0f; // in hex-Schreibweise

c2 = ~c1;
printf("%2hX = %d\n", c1, c1);
printf("%2hX = %d\n", c2, c2);
```

Bitweise Verknüpfungen: UND, ODER, EXKLUSIV ODER

Bit 1	Bit 2	UND	ODER	EXKLUSIV ODER
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- UND → Bits können gezielt auf null gesetzt werden
 - Beispiel für c vom Typ char: `c=c & 0x0f` → Ersten 4 Bits werden auf null gesetzt. Die anderen bleiben unverändert
- ODER → Bits können gezielt auf eins gesetzt werden
 - Beispiel: `c=c|0xf0;` → Ersten 4 Bits werden auf eins gesetzt
- Exklusiv-ODER → Bits können gezielt negiert werden bzw. kann ermittelt werden, welche Bits unterschiedlich sind

Bitweise Verknüpfungen: UND, ODER, EXKLUSIV ODER

- Beispiel:

```
// Beispiel UND, ODER,  
EXKLUSIV-ODER  
int a = 0x5C;  
int b = 0x3A;  
int c, d, e;  
  
c = a & b;  
d = a | b;  
e = a ^ b;
```

```
a = 0101 1100  
b = 0011 1010
```

a=92	5C	0000000001011100
b=58	3A	0000000000111010
a&b=24	18	0000000000011000
a b=126	7E	0000000001111110
a^b=102	66	0000000001100110

Bit-Verschiebung:

nach links: << bzw. nach rechts: >>

- Duale Ziffern werden um eine gewünschte Anzahl verschoben
- Beispiel:

```
// Verschiebung nach links  
char v1 = 1;  
v1 = v1 << 3;  
printf("v1=%d \t",v1);
```

$$1 \ll 3 \rightarrow 1000_2 = 8$$

Bit-Verschiebung:

nach links: << bzw. nach rechts: >>

- Beispiele:

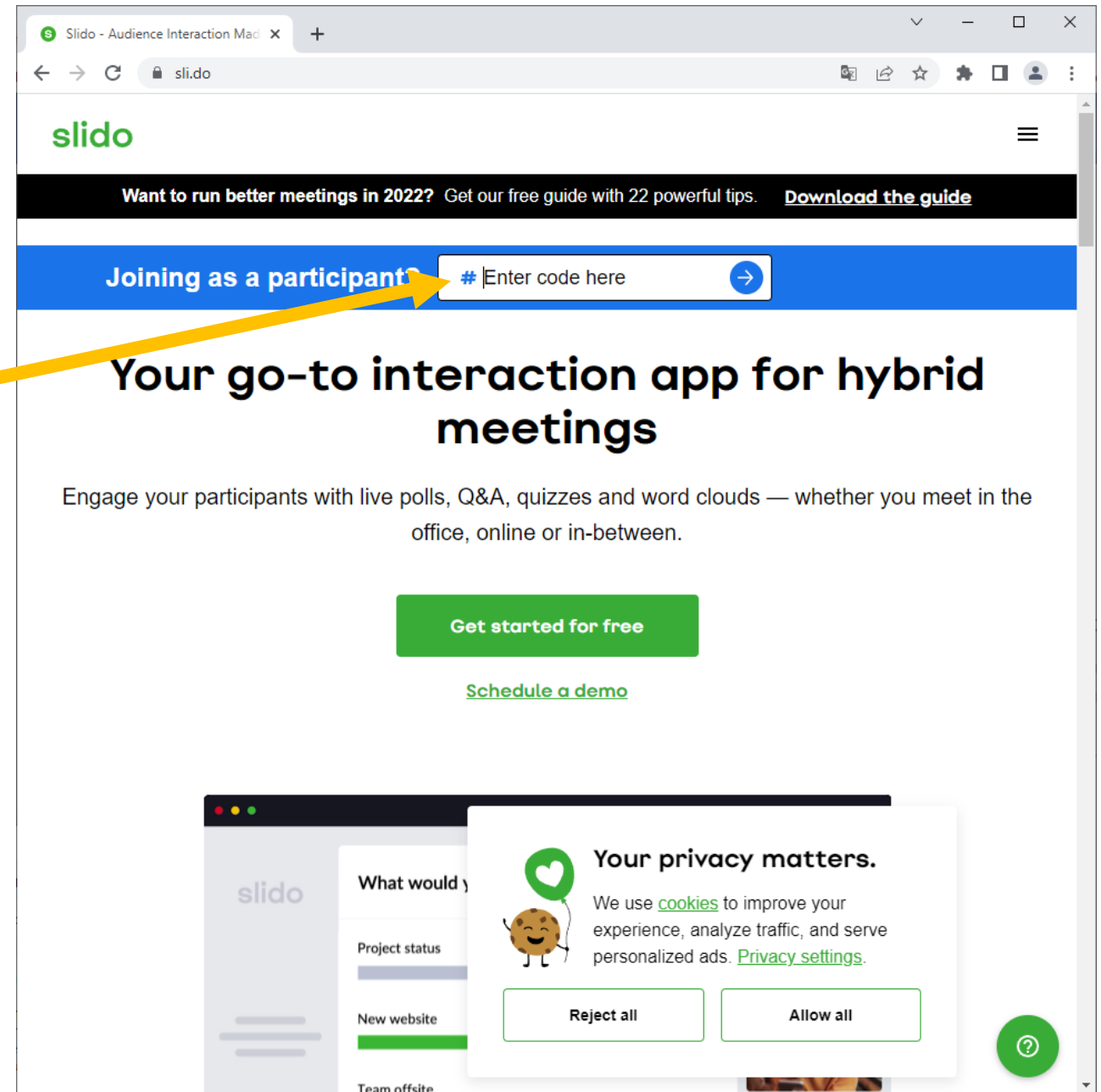
```
Verschiebung nach links:  
v1      = 4      0000000000000100  
v1<<3   = 32     0000000000100000  
v1      = -4     1111111111111100  
v1<<3   = -32    1111111111100000
```

```
Verschiebung nach rechts:  
v2      = 32     0000000000100000  
v2>>3   = 4      0000000000000100  
v2      = -32    1111111111100000  
v2>>3   = -4     1111111111111100
```

- Bei einer **Verschiebung nach links** werden die neuen Stellen mit null aufgefüllt
- Verschiebung um N Stellen nach links kommt einer Multiplikation mit 2^N gleich
- Bei einer **Verschiebung nach rechts** werden die Bits **in Visual Studio** mit dem Vorzeichenbit aufgefüllt
 - System-abhängig – kann für vorzeichenbehaftete Werte (Typ signed) auch null (*logical shift*) sein!

Quiz Time

- Öffne <http://www.slido.com>
- Code eingeben: **HAW-PR1**
- Seien Sie sparsam mit Daten
→ Vorname oder Spitzname
ausreichend

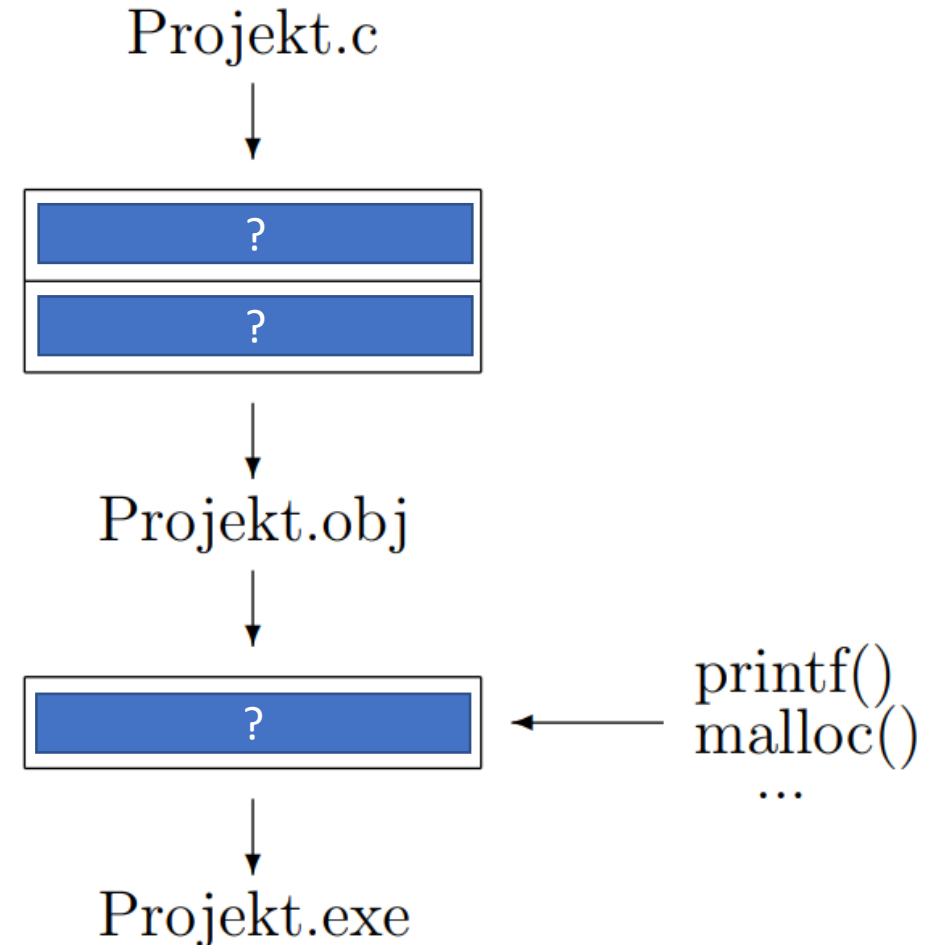


HEADER-DATEIEN

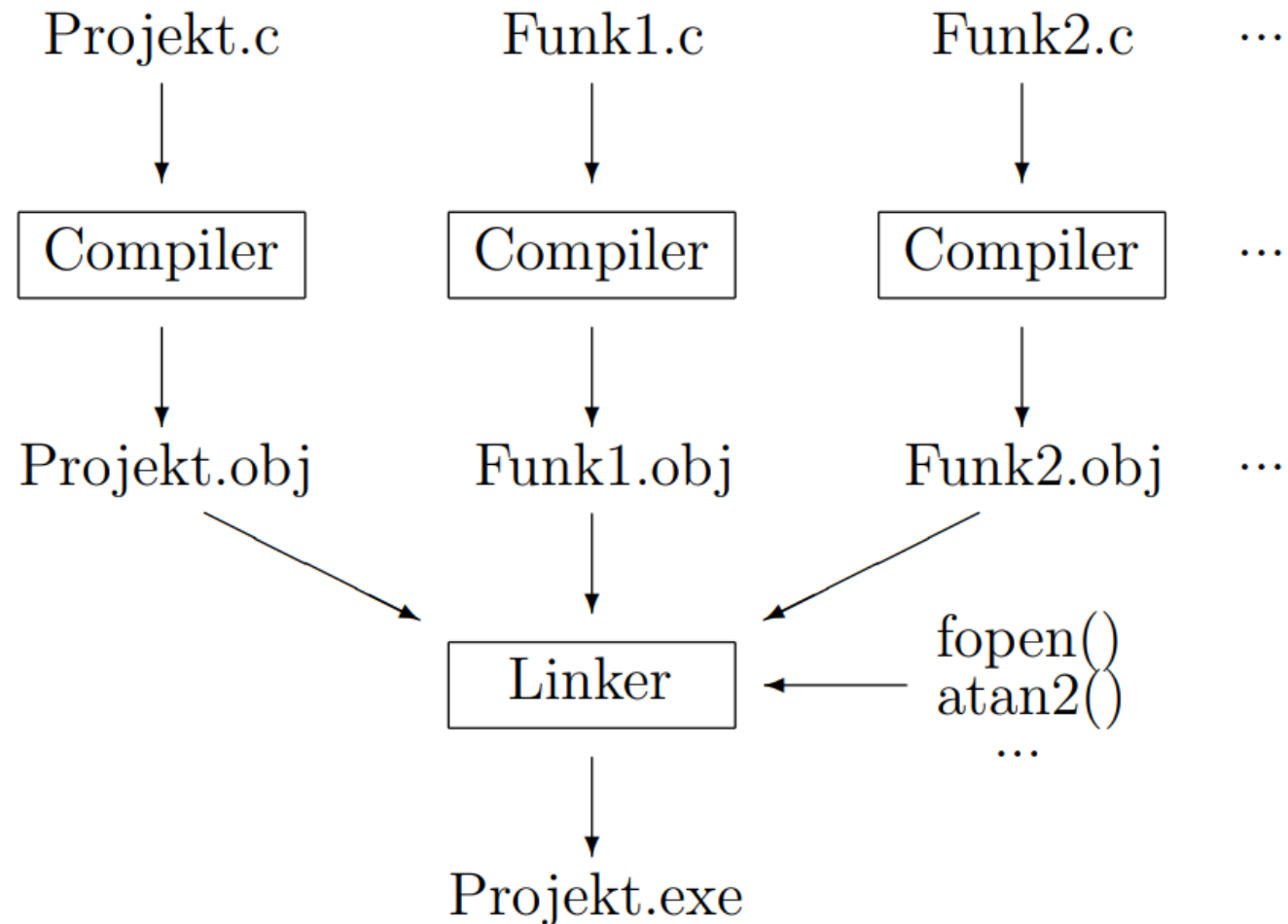
Quellcode auf mehrere Dateien verteilen

Wie wird aus Quellcode eine ausführbare Datei?

- Wenn Sie in Visual Studio ihren Code geschrieben haben und auf Starten drücken, passieren viele Schritte im Hintergrund
- eine vereinfachte Darstellung →
- Schritt 1: Aktionen vor dem Übersetzen durch den ...
- Schritt 2: Übersetzen des Quellcodes durch den ...
- Schritt 3: Zusammenfügen aller Funktionen durch den ...



Übersetzung mehrerer Quellcodedateien



Verknüpfung zwischen Quellcodedateien

- Um Funktionen in anderen Quellcodedateien nutzen zu können, müssen die Funktionsdeklarationen in der Datei bekannt sein
- Hierfür werden Header-Dateien genutzt (mit Dateiendung **.h**)
- In diese werden die Funktionsdeklarationen geschrieben (auch Preprozessor-Befehle; aber kein ausführbarer Code!)
- Die Header-Datei wird überall dort eingebunden, wo diese Funktionen genutzt werden
- Einbindung erfolgt über `#include "header_datei.h"`
 - Anführungsstriche geben an, dass die Header-Datei im Projektordner liegt (im Gegensatz zu Standard-Bibliotheken, die im Bibliotheksverzeichnis liegen)
- Häufig bilden Header- und Quellcode-Datei (.h & .c) ein Pärchen, d.h. in einer Header-Datei werden die Funktionsdeklarationen einer Quellcodedatei geschrieben

Beispiel: Header-Dateien

Eingabe.h

```
#ifndef MODUL_EINGABE
#define MODUL_EINGABE

int getInt(char *text);
float getFloat(char *text);

#endif
```

Main.c

```
#include "Eingabe.h"

int main()
{
    int Alter;
    ...
    Alter = getInt("Alter:");
    ...
    return 0;
}
```

Eingabe.c

```
#include "Eingabe.h"

int getInt(char *text)
{
    ...
}

float getFloat(char *text)
{
    ...
}
```


Preprozessorbefehle in der Header-Datei

- Wenn Header-Dateien mehrfach eingebunden werden, werden auch Funktionen und Makros mehrfach deklariert
- Dies kann zu Warnungen und Fehlermeldungen führen (vor allem bei größeren Projekten)
- In Visual Studio aber nicht immer der Fall, da Abhängigkeiten tw. in Projektstruktur abgebildet
- Auch mit Preprozessorbefehlen kann dies verhindert werden:

Vorteil: einfacher & kürzer
Nachteil: wird nicht von allen
Compilern unterstützt

Option 1: **#PRAGMA ONCE**

```
#pragma once
short getShort(char[], short, short);
// ..
```

Vorteil: Standard-konform
Nachteil: Makros müssen eindeutig sein
(deswegen häufig sehr lang)

Option 2: **Include-Guard**

```
#ifndef GET_SHORT_H
#define GET_SHORT_H
short getShort(char[], short, short);
// ..
#endif
```

Gestaltung einer Header-Datei

- Deklarieren Sie nur Funktionen in der Header-Datei, die auch außerhalb der Datei genutzt werden sollen
- Funktionen, die nur innerhalb der Quellcodedatei genutzt werden, werden weiterhin dort deklariert
- Beispiel:

Sie erstellen in einem Modul die Funktion `getFloat()`, die ihrerseits die Funktion `checkRange()` aufruft.

Erstere soll von außen aufgerufen werden, letztere nur innerhalb des Moduls.



Eingabe.h

```
#ifndef MODULEINGABE
#define MODULEINGABE
float getFloat(char *text, float min, float max);
#endif
```

Eingabe.c

```
#include "Eingabe.h"
int checkRange(float number, float min, float max);
float getFloat(char *text)
{
    ...
}
int checkRange(float number, float min, float max)
{
    ...
}
```

Übung – Projekt aufsetzen & testen

http://www.rrhess.de/PrII_MiniProjekt.php

PREPROZESSOR

Pre-Compiler

Makros werden typischerweise in Großbuchstaben geschrieben!

Makros sind nur innerhalb der aktuellen Quelldatei gültig oder können zurückgesetzt werden, z.B. `#undef WIN32`

- Pre-Compiler (Vor-Übersetzer) erledigt einige Aufgaben bevor die Übersetzung beginnt
- Pre-Compiler-Befehle sind in C alle Befehle die mit einem Doppelkreuz `#` beginnen

Beispiele:

```
#include <include-Datei>
```

Pre-Compiler ersetzt alle include-Anweisungen durch den Inhalt der angegebenen Datei

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#define PI 3.14159
```

```
#define WIN32
```

Definiert ein Makro (d.h. Konstanten/Code, die durch den Pre-Compiler ersetzt werden)

```
#ifdef WIN32
```

```
#include <windows.h>
```

```
#else
```

```
#include <unistd.h>
```

```
#endif
```

Bedingte Ersetzung (Beispiel prüft, ob WIN32-Makro gesetzt wurde und bindet abhängig hiervon eine include-Datei ein)

`#ifdef` → prüft, ob Identifier gesetzt wurde (*if defined*)

`#ifndef` → prüft, ob Identifier nicht gesetzt ist (*if not defined*)

Weitere Beispiele für die Nutzung des Präprozessors: Bedingte Kompilierung

Programmblöcke hinzufügen/auslassen,
z.B. zum Debuggen

```
#define DEBUG

int main(void) {
    int a = 0;
    a++;

    #ifdef DEBUG
    printf("a=%d", a);
    #endif

    a--;
    // ..
}
```

Zwei Programmblöcke alternativ zu kompilieren

```
#ifdef TEST
printf("TEST-Version mit zusätzlichen ...\n");
#else
printf("Release-Version\n");
#endif
```

Weitere Beispiele für die Nutzung des Präprozessors: Ausdrücke & Funktionen

Beispiel:

```
#define MAX(A,B)  A>B?A:B  
#define MIN(A,B) (A)<(B)?(A):(B)
```

Aufruf:

```
printf("%d\n", MAX(1, 10));  
printf("%d\n", MIN(1, -1));
```

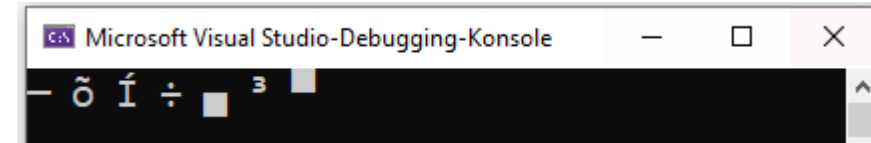
Preprozessor setzt die
Makrodefinition bei Verwendung
des Namens ein, z.B.
`printf("%d\n", 1>10?1:10);`

UMLAUTE & ZEICHENSÄTZE

Umlaute in C

- Problem: Falsche Darstellung der Umlaute und anderer Sonderzeichen in der Ausgabe (z.B. Konsole unter Windows)

```
printf("Ä ä Ö ö Ü ü ß\n\n");
```




- Warum?
 - Editor und Konsole nutzen unterschiedliche Zeichensätze
 - Diese sind zwar identisch für „normale“ Buchstaben (ASCII)
 - Aber unterschiedlich für Umlaute (und Sonderzeichen)

Zeichensätze


- Zeichensätze legen fest wie Buchstaben bzw. Schrift im Computer gespeichert werden (z.B. als Zahlenwert)
- Ihr Wert wird durch den Code festgelegt, der vom verwendeten Zeichensatz definiert ist
- Allgemein gibt es unterschiedliche Zeichensätze, die genutzt werden, z.B.
 - ASCII (American Standard Code for Information Interchange) mit 127 Zeichen
 - Latin-1 (ISO-8859-1) mit 256 Zeichen (Unix, Linux)
 - CODEPAGE-850 mit 256 Zeichen (MS-DOS Eingabeaufforderung unter Windows)
 - Unicode (z.B. UTF-8) um alle gängigen Zeichen aus den weltweiten Schriften darzustellen
- Zeichensätze sehr häufig kompatibel mit ASCII (ersten 127 Zeichen identisch)
- In C ist der Datentyp char nur 1 Byte groß und kann nur bis zu 256 Zeichen
- Für größere Zeichensätze (z.B. Unicode) werden spezielle Bibliotheken genutzt

Beispiele für Zeichensätze


Beispiele für unicode



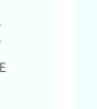
U+723B




U+221E




U+708E



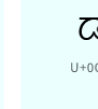
U+3001




U+1F23A




U+0CA1




U+270E



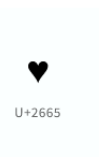
U+FF9F



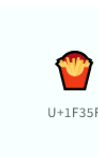
U+4E14




U+3071



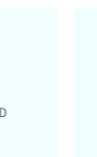
U+2665




U+1F35F




U+30ED



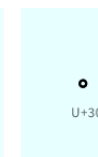
U+1F3A4




U+1F9E2



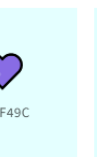
U+3002




U+1F49C



U+0296



U+0573



U+2014

ASCII-Tabelle

Dez/Hex/Okt	Zeichen	Dez/Hex/Okt	Zeichen	Dez/Hex/Okt	Zeichen	Dez/Hex/Okt	Zeichen
0/00/000	NUL	32/20/040	SP	64/40/100	@	96/60/140	`
1/01/001	SOH	33/21/041	!	65/41/101	A	97/61/141	a
2/02/002	STX	34/22/042	"	66/42/102	B	98/62/142	b
3/03/003	ETX	35/23/043	#	67/43/103	C	99/63/143	c
4/04/004	EOT	36/24/044	\$	68/44/104	D	100/64/144	d
5/05/005	ENQ	37/25/045	%	69/45/105	E	101/65/145	e
6/06/006	ACK	38/26/046	&	70/46/106	F	102/66/146	f
7/07/007	BEL	39/27/047	'	71/47/107	G	103/67/147	g
8/08/010	BS	40/28/050	(72/48/110	H	104/68/150	h
9/09/011	TAB	41/29/051)	73/49/111	I	105/69/151	i
10/0A/012	LF	42/2A/052	*	74/4A/112	J	106/6A/152	j
11/0B/013	VT	43/2B/053	+	75/4B/113	K	107/6B/153	k
12/0C/014	FF	44/2C/054	,	76/4C/114	L	108/6C/154	l
13/0D/015	CR	45/2D/055	-	77/4D/115	M	109/6D/155	m
14/0E/016	SO	46/2E/056	.	78/4E/116	N	110/6E/156	n
15/0F/017	SI	47/2F/057	/	79/4F/117	O	111/6F/157	o
16/10/020	DLE	48/30/060	0	80/50/120	P	112/70/160	p
17/11/021	DC1	49/31/061	1	81/51/121	Q	113/71/161	q
18/12/022	DC2	50/32/062	2	82/52/122	R	114/72/162	r
19/13/023	DC3	51/33/063	3	83/53/123	S	115/73/163	s
20/14/024	DC4	52/34/064	4	84/54/124	T	116/74/164	t
21/15/025	NAK	53/35/065	5	85/55/125	U	117/75/165	u
22/16/026	SYN	54/36/066	6	86/56/126	V	118/76/166	v
23/17/027	ETB	55/37/067	7	87/57/127	W	119/77/167	w
24/18/030	CAN	56/38/070	8	88/58/130	X	120/78/170	x
25/19/031	EM	57/39/071	9	89/59/131	Y	121/79/171	y
26/1A/032	SUB	58/3A/072	:	90/5A/132	Z	122/7A/172	z
27/1B/033	ESC	59/3B/073	;	91/5B/133	[123/7B/173	{
28/1C/034	FS	60/3C/074	<	92/5C/134	\	124/7C/174	
29/1D/035	GS	61/3D/075	=	93/5D/135]	125/7D/175	}
30/1E/036	RS	62/3E/076	>	94/5E/136	^	126/7E/176	~
31/1F/037	US	63/3F/077	?	95/5F/137	_	127/7F/177	DEL

Latin-1

iso-8859-1										
+	0	1	2	3	4	5	6	7	8	9
160		í	ñ	£	¤	¥	¦	§	¨	©
170	ª	«	¬	­	®	¯	°	±	²	³
180	´	µ	¶	·	¸	¹	º	»	¼	½
190	¾	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
200	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ
210	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û
220	Ü	Ý	Þ	ß	à	á	â	ã	ä	å
230	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù
250	ú	û	ü	ý	þ	ÿ				

CODEPAGE-850

ASCII Table (close: ctrl-F4)										
+	0	1	2	3	4	5	6	7	8	9
00	*	☐	☐	♥	♦	♣	♠	•	◻	◻
10	▶	◀	†	!!	¶	§	¶	†	↓	→
20		!	"	#	\$	%	&	'	()
30	0	1	2	3	4	5	6	7	8	9
40	@	A	B	C	D	E	F	G	H	I
50	P	Q	R	S	T	U	V	W	X	Y
60	`	a	b	c	d	e	f	g	h	i
70	p	q	r	s	t	u	v	w	x	y
80	Ç	ü	é	â	ä	à	ã	ç	ê	ë
90	Ê	æ	ŕ	ô	ö	ò	û	ÿ	Ö	Ü
A0	á	í	ó	ú	ñ	Ñ	¸	¸	¿	¿
B0	Ł	ł	Ł	ł	Ł	ł	Ł	ł	Ł	ł
C0	Ł	ł	Ł	ł	Ł	ł	Ł	ł	Ł	ł
D0	ø	ð	È	É	Ê	Ë	Ì	Í	Î	Ï
E0	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü
F0	-	±	=	¼	½	¾	¸	¸	¸	¸

Umlaute in der Konsole ausgeben

- Windows-Konsole nutzt (typischerweise) Codepage 850
- Umlaute können über mehrere Wege ausgegeben werden
- Ein möglicher Weg ist über den Platzhalter `%c`
- Beispiel

```
// Umlaute werden in der Konsole falsch dargestellt
printf("Ä ä Ö ö Ü ü ß\n\n");

// Wie können sie richtig dargestellt werden?
printf("Gänsefüßchen oder G%cñsef%c%cchen\n", 132, 129, 225);
```

`/* CODEPAGE 850:`

Zeichen	Dezimal
=====	
'Ä'	142
'ä'	132
'Ö'	153
'ö'	148
'Ü'	154
'ü'	129
'ß'	225

`*/`

Umlaute in der Konsole ausgeben (II)

```
// Umlaute werden in der Konsole falsch dargestellt
printf("Ä ä Ö ö Ü ü ß\n\n");

// Wie können sie richtig dargestellt werden?
// Weg 1
printf("Gänsefüßchen oder G%cnef%c%cchen\n", 132, 129, 225);

// Weg 1b - Makros
printf("Gänsefüßchen oder G%cnef%c%cchen\n", ae, ue, ss);

// Weg 1b - 2.Beispiel
printf("%c %c %c %c %c %c %c\n", AE, ae, OE, oe, UE, ue, ss);

// Weg 2 über Escape-Sequenz und direkte Zeichenauswahl als oktal oder
hexadezimal Zahl
// hexadezimal:
// ist das nächste Zeichen auch hexadezimal, wird dieses ohne
Leerzeichen als Teil des hexadezimal Zahl interpretiert
// um dies zu verindern, können sie Strings mit " abschließen/starten
printf("G\x84nef\x81\xE1"chen\n");

// oktal
printf("G\204nef\201\341chen\n");
```

Makrodefinitionen:

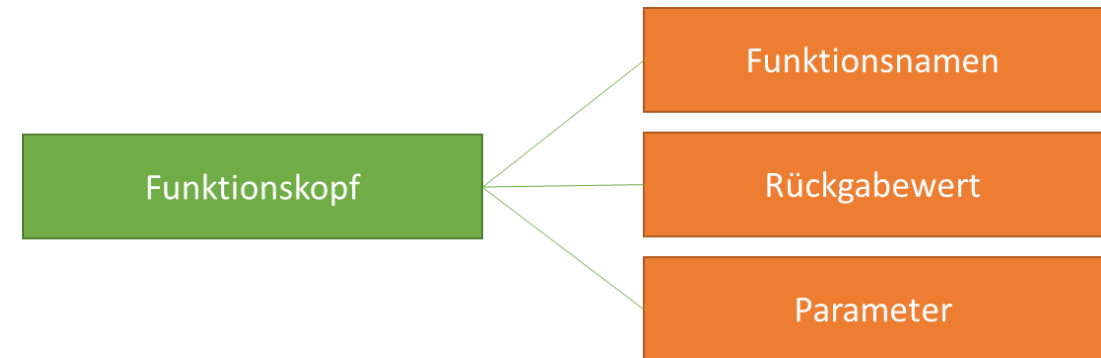
```
#define AE 142
#define ae 132
#define OE 153
#define oe 148
#define UE 154
#define ue 129
#define ss 225
```

FUNKTIONEN

- CALL BY VALUE**
- CALL BY REFERENCE**

Funktionen (Wiederholung)

- Einer Funktion können Parameter übergeben werden
- Diese sind im Funktionskopf angegeben (Datentyp & Name)
- Beispiel:



Typ des Rückgabewerts → `int` `meineMaxFunktion` (Name der Funktion) `(int a, int b)` (Typ und Name der übergebenen Parameter) `// Funktionskopf` (Mehrere Parameter werden durch Komma getrennt)

Call-by-Value & Call-by-Reference

- In den bisher vorgestellten Funktionen
 - kann nur ein Wert **zurückgegeben** werden (→ Rückgabewert)
 - Werden die Parameter als Kopien übergeben
 - d.h. wenn sie die Parameter in der Funktion ändern, hat dies keine Auswirkung auf den Wert außerhalb der Funktion
 - Dies wird auch als **Call-By-Value** (Wertparameter) bezeichnet
- Anstatt von Kopien können auch Referenzparameter übergeben werden → **Call-by-Reference**
 - Hierfür werden **Zeiger** verwendet
 - Zeiger sind Variablen auf eine Speicherstelle und können angeben, wo andere Variablen gespeichert sind
 - Zeiger werden ausführlich in den nächsten Vorlesungen behandelt

Beispiel: Zwei Zahlen austauschen, SwapInt



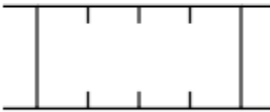

Es sollen die Werte für die Variable a und b ausgetauscht werden:

```
temp = a;  
a = b;  
b = temp;
```

Wie sieht eine entsprechende Funktion aus?

```
void swapInt_falsch(int a, int b) {  
    int temp;  
  
    temp = a;  
    a = b;  
    b = temp;  
  
}
```

Falscher Ansatz: Variable wird kopiert!

Typ:	int	int	int	int
Name:	a	b	value1	value2
Speicher:				
Wert:	5	9	9	5

Beispiel: Zwei Zahlen austauschen, SwapInt

Richtige Lösung mit Zeigern

```
void swapInt(int* a, int* b) {  
    int temp;  
  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Mit * werden
Zeigervariablen definiert

Funktionsaufruf mit Zeigern auf
Speicherstelle

```
a = 5;  
b = 9;  
  
swapInt(&a, &b);  
printf("a=%d\tb=%d\n", a, b);
```

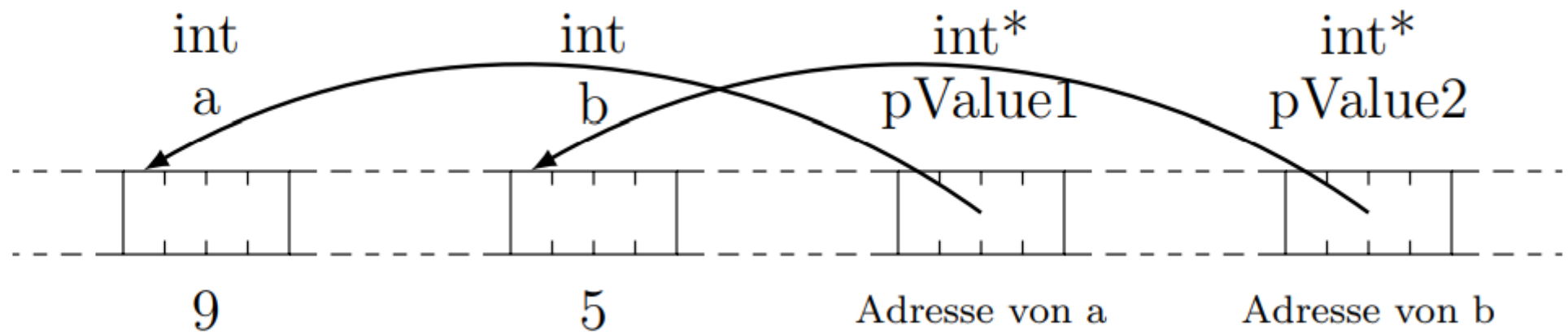
Mit & wird die
Speicheradresse einer
Variablen ermittelt

Typ:

Name:

Speicher:

Wert:



Vollständiger Quellcode swapInt

```
#include<stdio.h>

void swapInt_falsch(int value1, int value2); // falscher
Ansatz mit Call-by-Value
void swapInt(int* pValue1, int* pValue2); // Tausch mithilfe
von Zeigern (Call-by-Reference)

int main() {
    int a, b;

    a = 5;
    b = 9;

    printf("a=%d, b=%d\n", a, b);

    swapInt_falsch(a, b);
    printf("nach swapInt_falsch... a=%d, b=%d\n", a, b);

    swapInt(&a, &b);
    printf("nach swapInt... a=%d, b=%d\n", a, b);

    return 0;
}
```

```
void swapInt_falsch(int value1, int value2)/* falsch
oder richtig? */
{
    int tmp;
    tmp = value1;
    value1 = value2;
    value2 = tmp;
}

void swapInt(int* pValue1, int* pValue2)
{
    int tmp;
    tmp = *pValue1;
    *pValue1 = *pValue2;
    *pValue2 = tmp;
}
```

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT!