

□ **Termin:**

**30. Juni 2005 18.00 -19.00 Uhr**

□ **Anmeldung im Tutorium**

□ **Hörsalverteilung auf der TI-Homepage (am 29. Juni)**

### 1.3 Historische Entwicklung von Prozessoren

➤ **1976:**

- **TMS 9000 von Texas Instruments:**  
16 Bit Prozessor, verwaltete seine Register im Schreib-Lesespeicher => rascher Programmwechsel, aber langsame Verarbeitungszeit
- **Z80 der Firma Zilog:**  
8 Bit Prozessor, aufwärtskompatibel zu 8080, aber mit höherer Leistungsfähigkeit und mehr Befehlen
- **8085 von Intel:**  
Erweiterung des 8080 mit verbesserter Unterbrechungsverwaltung, verbesserter Peripheriesteuerung

### 1.3 Historische Entwicklung von Prozessoren

➤ **1979:**

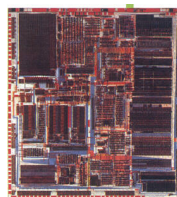
**Erste Signalprozessoren, z.B. 2929 von Intel:** spezialisiert auf die Verarbeitung von analogen Signalen, die durch interne AD/DA-Wandler digitalisiert werden.

**1979 existierten ca. 80 verschiedene Mikroprozessoren, es wurden bis dahin ca. 75 Millionen Mikroprozessoren verkauft**

### 1.3 Historische Entwicklung von Prozessoren

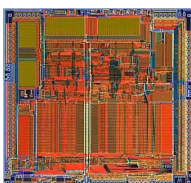
➤ **1985: 80386 von Intel**

- 32 Bit Prozessor
- CMOS Technologie
- 275 000 Transistoren
- virtuelle Speicherverwaltung, Segmentierung, Paging



➤ **1986: 68020 von Motorola**

- 32 Bit Prozessor
- ca. 200 000 Transistoren
- virtueller Adressraum



# Kapitel 1

1.1 Motivation

1.2 Historische Entwicklung von Rechenmaschinen

1.3 Historische Entwicklung von Mikroprozessoren

### 1.3 Historische Entwicklung von Prozessoren

➤ **1978:**

**8086 von Intel:** Erster 16 Bit Prozessor von Intel, HMOS Technologie (High Density MOS), ca. 27000 Transistoren (aber 30% mehr Fläche als ein 8080), virtuelle Speicherverwaltung, 16 Bit Datenbus, 20 Bit Adressbus (1MByte)

➤ **1979:**

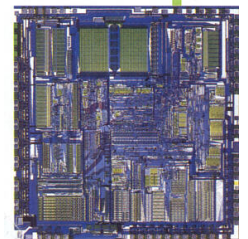
- **68000 von Motorola:** 16 Bit Prozessor, intern jedoch 32 Bit Registersatz, HMOS Technologie, ca. 68000 Transistoren, 24 Bit Adressbus (16 MByte), orthogonaler Befehlsatz
- **Z8000 von Zilog:** 16 Bit Prozessor, Nachfolger des Z80

### 1.3 Historische Entwicklung von Prozessoren

➤ **1982:**

• **80286 von Intel**

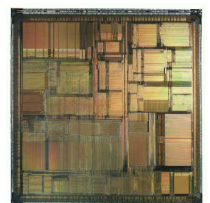
- Nach dem 80186 der zweite Nachfolger des 8086
- ca. 130000 Transistoren
- Erweiterter Adressraum (16 Mbyte)
- Mehrere Betriebsarten
- Multitasking-Unterstützung
- Jahrelang in vielen Personal Computern (z. B. IBM AT) eingesetzt



### 1.3 Historische Entwicklung von Prozessoren

➤ **RISC Mikroprozessoren:**

- **Advanced Micro Devices Am29000 (~1987)**
- **Sun Microsystems SPARC (April 1987):**
  - 32 Bit CPU mit über 55.000 Transistoren
  - Alle Befehle waren 32 Bit Breit
  - Ausführungszeit lag bei nur 1.3 Takten pro Befehl
  - Eine Sun war 3 mal schneller als 386er Rechner bei einem Fünftel dessen Komplexität.
  - Der SPARC verfügte in der ersten Version über 128 Register
- **MIPS technologies (MIPS R2000 / MIPS R3000)**

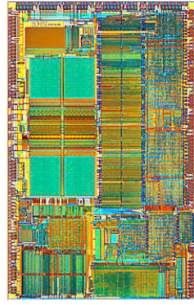


Sun Microsystems SPARC

### 1.3 Historische Entwicklung von Prozessoren

#### > 1989: 80486 von Intel

- Erweiterung des 80386 um integrierten Cache und integrierten numerischen Coprozessor
- ca. 1 200 000 Transistoren
- Multiprozessor-Unterstützung



#### > 1990: 68040 von Motorola

- Nach 68030 zweiter Nachfolger des 68000
- ca. 1 200 000 Transistoren



### 1.3 Historische Entwicklung von Prozessoren

#### > 1992: Pentium von Intel

- Nachfolger des 80486
- ca. 3 100 000 Transistoren
- intern teilweise 64 Bit Architektur
- 2 fach Superskalar, Code und Datencache

#### > 1992-95: Power PC's MPC601, MPC603, MPC604, MPC620 von Motorola/IBM/Apple

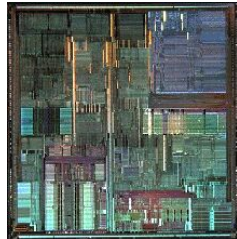
- RISC Architektur, teilweise 64 Bit Architektur (Daten)
- Superskalar
- ca. 4 000 000 Transistoren (MPC620)



### 1.3 Historische Entwicklung von Prozessoren

#### > 1995: Pentium Pro von Intel

- Nachfolger des Pentium
- ganz anderer interner Aufbau
- 3-5 fach Superskalar
- ca. 14 stufige Befehlspipeline
- 5 500 000 Transistoren
- Zwei eingebaute Cache-Speicher-Ebenen
- speculative execution, dynamic branch prediction



### 1.3 Historische Entwicklung von Prozessoren

#### > 1996: Pentium II

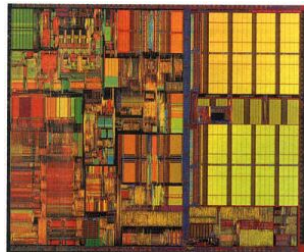
- Nachfolger vom Pentium Pro mit speziellen Multimedia-Erweiterungen (MMX)
- 3-5 fach Superskalar
- ca. 14 stufige Befehlspipeline
- 7 500 000 Transistoren
- speculative execution, dynamic branch prediction



### 1.3 Historische Entwicklung von Prozessoren

#### > 1998: Pentium III

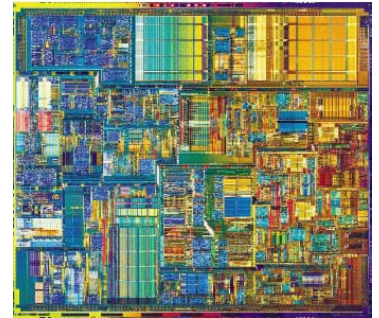
- Nachfolger vom Pentium II mit Internet Streaming SIMD Extension (ISSE)
- (*SIMD = Single Instruction, Multiple Data*)
- 16 KByte Daten- und Befehls-Cache mit vollem Prozessortakt.
- 2nd-Level-Cache mit halbem Prozessortakt
- Anbindung an die Außenwelt über einen mit 100 - 133 MHz arbeitenden Systembus.



### 1.3 Historische Entwicklung von Prozessoren

#### > 2000: Pentium 4:

- komplette Neuentwicklung
- Intel® NetBurst™ micro-architecture
- Nachfolger vom Pentium III mit Internet Streaming SIMD Extensions 2
- Enhanced floating point/multimedia
- Advanced dynamic execution
- Hyper-threading technology
- Execution trace cache and advanced transfer cache
- 400MHz System Bus



## Geschichte der Rechnerstrukturen

Tabellarische Auflistung der wichtigsten Rechanlagen, Mikrorechner und Mikroprozessoren von 1936 bis 2003 auf der TI-Homepage

<http://i61www.ira.uka.de/users/asfour/TI/Geschichte>



## Vorlesungsgliederung

### □ Einführung

- > Motivation, Historische Anmerkungen

### □ Anforderungen höherer Programmiersprachen

- > Programmkonstrukte
- > Variable und Konstante

### □ Ein grundlegendes Rechnermodell

- > Leitwerk, Rechenwerk
- > Speicherwerk
- > Ein-Ausgabewerk
- > Verbindungsstrukturen
- > Maschinenbefehlszyklus



## Anforderungen höherer Programmiersprachen: Die Programmiersprache C

- > Vom Quellcode zum ausführbaren Programm
- > Die Entwicklungsgeschichte von C
- > Grundlagen: Datentypen, Operatoren, Ausdrücke
- > Kontrollstrukturen
- > Funktionen und Programmstruktur
- > Zeiger und Vektoren
- > ...

- **Maschinensprache:** Repräsentation von Anweisungen, die für einen Mikroprozessor unmittelbar verständlich sind, z. B.

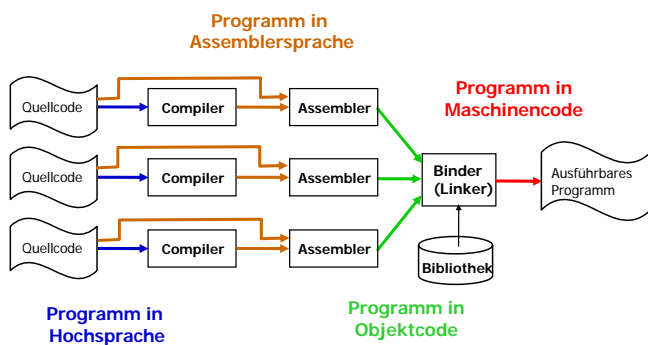
```
00000000110000100011000000100001
```

- **Assemblersprache:** Symbolische Repräsentation der Maschinensprache, die für den Menschen verständlich und anschaulich ist, z. B.

```
add $s2, $s1, $s0 # $s2:= $s1 + $s0
```

- **Symbolischer Befehl ≡ Maschinen-Befehl**

## 2.1 Vom Quellcode zum ausführbaren Programm



## 2.1 Vom Quellcode zum ausführbaren Programm

- **Assembler:** Programm, das einen Quellcode in Assemblersprache in eindeutiger Weise in Maschinensprache übersetzt
- **Objektcode:** Repräsentation eines Maschinenprogramms, in dem noch ungelöste Referenzen auf externe Unterprogramme oder Speicherbereiche enthalten sind
- Zusätzlich können im Objektcode Informationen enthalten sein, die die Fehlersuche mit einem **Debugger** ermöglichen
- **Binder (Linker):** Programm, das die ungelösten Referenzen mehrere Objektcode-Module auflöst und sie zu einem ausführbaren Programm verbindet

## 2.2 Entwicklungsgeschichte von C

- **1969:** Ken Thomson (Bell Laboratories) erstellte erste Version von UNIX in Assembler
- **1970:** Ken Thomson entwickelte auf einer PDP/7 die Sprache B als Weiterentwicklung der Sprache BCPL. B ist eine typlose Sprache, sie kennt nur Maschinenworte
- **1974:** Weiterentwicklung von B zu C durch Dennis M. Ritchie. Erste Implementation auf einer PDP 11
- **Heute:** C ist eigenständige, betriebssystemunabhängige Programmiersprache. Sie ist auf praktisch allen Rechnerplattformen vom PC bis hin zum Supercomputer und unter allen wichtigen Betriebssystemen verfügbar.

## 2.2 Entwicklungsgeschichte von C

- Im Laufe der Zeit entstanden "C-Dialekte", welche die dringende Notwendigkeit einer Standardisierung zeigten.
- 1988 hat das ANSI-Komitee X3J11 diesen Sprachstandard für die Programmiersprache C veröffentlicht, der kurz **ANSI C** genannt wird. Neuere C-Compiler sollten dem ANSI-Standard entsprechen.
- **The C programming language** von Brian W. Kernighan und Dennis M. Ritchie, 9. Auflage, ANSI-Standard
- Die Entwicklung von ist C eng mit der UNIX-Entwicklung verbunden

## Kurzdarstellung der Sprache C

- C nimmt eine Zwischenstellung zwischen Assembler und Hochsprache ein. Sie vereint zwei an sich widersprüchliche Eigenschaften:
  - > gute Anpassung an die Rechnerarchitektur (Hardware)
  - > hohe Portabilität → einfachere und vor allem schnellere Anpassungen an eine andere Hardware
- Einfachere Programmierung → kürze Entwicklungszeiten für die Software
- C-Compiler erzeugen sehr effizienten Code sowohl bzgl. Laufzeit als auch bzgl. Programmgröße → für die meisten Anwendungsfälle kann auf den Einsatz eines Assemblers verzichtet werden.

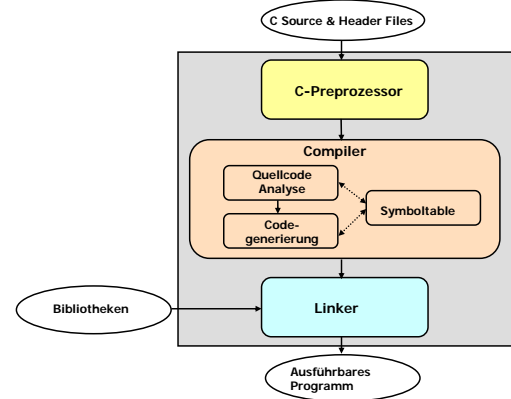
## Kurzdarstellung der Sprache C

- C kennt nur 4 Datentypen: **char**, **int**, **float** und **double**.
- Es gibt einfache Kontrollstrukturen: Entscheidungen, Schleifen, Zusammenfassungen von Anweisungen (Blöcke) und Unterprogramme.
- Ein-/Ausgabe ist nicht Teil der Sprache C sondern wird über Bibliotheksfunktionen erledigt. Keine Operationen auf zusammengesetzten Objekten als Ganzes (z. B. Zeichenketten).
- Parameter werden als Werte an Funktionen übergeben, daher kann eine Funktion diese nicht ändern.
- Möglichkeit von Zeigern (Adressen) als Parametern, deren Zielobjekt geändert werden kann.

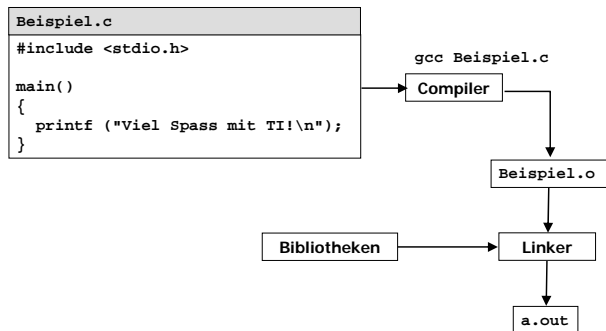
# Kurzdarstellung der Sprache C

- Zeiger stellen sehr mächtige und vielseitige Anwendungsmöglichkeiten bereit. Sie werden durch eine automatische Adressarithmetik unterstützt.
- Für Datenwandlungen zwischen den einzelnen Typen gibt es kaum Beschränkungen
- Weiterhin gibt es z. B. mittels der Speicherklasse "register" Bezüge zur Hardware.
- ...

# Programmerstellung und -ausführung



## Einfaches Beispiel



## 2.3 Grundlagen: Datentypen

### 4 Grunddatentypen

- char**: einzelnes Zeichen (Charakter); meist 1 Byte
- int**: Integerzahl; 2 oder 4 Byte
- float**: Gleitkommazahl; meist 4 Byte
- double**: Gleitkommazahl in doppelter Genauigkeit; meist 8 Byte
- Wertebereiche dieser Grunddatentypen ist **rechnerabhängig**
- Beispiele:

- `char buchstabe = 'T';`
- `int i = 4;`
- `double pi = 3.1415;`

## 2.3 Grundlagen: Operatoren

### Arithmetische Operatoren:

Operator Symbol	Operation	Beispiel
*	Multiplikation	x * y
/	Division	x / y
%	Modulo	x % y
+	Addition	x + y
-	Subtraktion	x - y

## 2.3 Grundlagen: Operatoren

### Bit-Operatoren:

Operator Symbol	Operation	Beispiel
~	Bitweise NOT	~x
<<	links Schieben	x << y
>>	rechts Schieben	x >> y
&	Bitweise AND	x & y
^	Bitweise XOR	x ^ y
	Bitweise OR	x   y

## 2.3 Grundlagen: Operatoren

### Vergleichsoperatoren:

Operator Symbol	Operation	Beispiel
>	größer als	x > y
>=	größer gleich	x >= y
<	kleiner als	x << y
<=	kleiner gleich	x <= y
==	gleich	x == y
!=	ungleich	x != y

## 2.3 Grundlagen: Operatoren

### Spezial-Operatoren:

Operator Symbol	Operation	Beispiel
++	Inkrement (postfix)	x++
--	Dekrement (postfix)	x--
++	Inkrement (präfix)	++x
--	Dekrement (präfix)	--x
+=	add and assign	x += y
-=	subtract and assign	x -= y
*=	multiply and assign	x *= y
/=	divide and assign	x /= y
%=	modulus and assign	x %= y
&=	and and assign	x &= y
=	or and assign	x  = y
^=	xor and assign	x ^= y
<<=	left-shift and assign	x <<= y
>>=	right-shift and assign	x >>= y

## 2.3 Grundlagen: Operatoren

### Spezial-Operatoren:

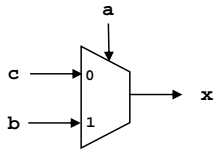
Operatorpaar „?:“ `expr1 ? expr2 : expr3`

#### Beispiel 1:

```
z = (a > b) ? a : b; /* z = max(a,b) */
```

#### Beispiel 2:

```
x = a ? b : c /* a true dann x = b */  
/* sonst x = c */
```



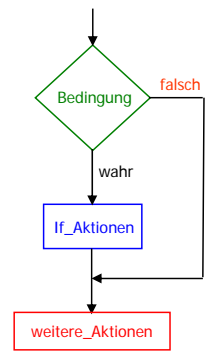
Bedingter Ausdruck entspricht der logischen Funktion eines Multiplexers

## 2.4 Kontrollstrukturen

Kontrollstrukturen definieren die Reihenfolge von Berechnungen

### if Anweisung:

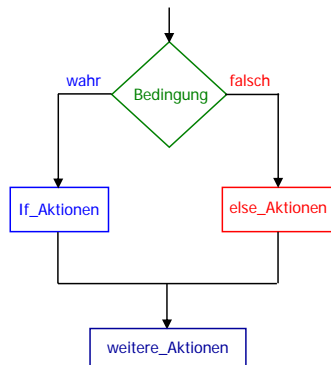
```
if (Bedingung_ist_wahr)  
{  
    if_Aktionen  
}  
weitere_Aktionen;
```



## 2.4 Kontrollstrukturen

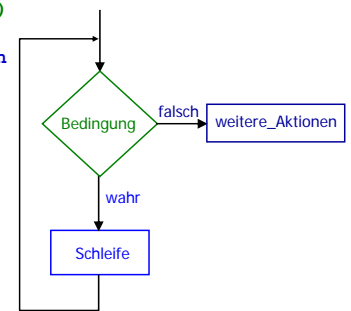
### if-else Anweisung:

```
if (Bedingung_ist_wahr)  
{  
    if_Aktionen  
}  
else  
{  
    else_Aktionen  
}  
weitere_Aktionen;
```



### while Schleifen:

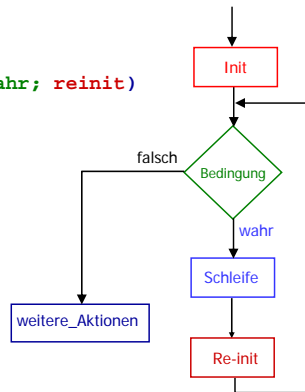
```
while (Bedingung_ist_wahr)  
{  
    while_Aktionen  
}  
weitere_Aktionen;
```



## 2.4 Kontrollstrukturen

### for Schleifen:

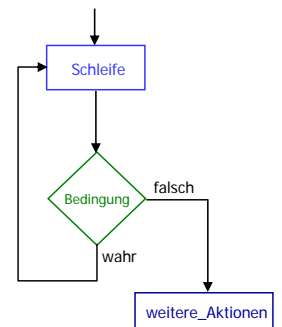
```
for (init; Bedingung_ist_wahr; reinit)  
{  
    for_Aktionen  
}  
weitere_Aktionen;
```



## 2.4 Kontrollstrukturen

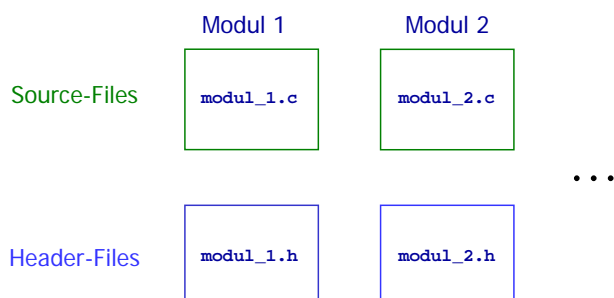
### do-while Schleife:

```
do {  
    do_Aktionen  
}  
while (Bedingung_ist_wahr)  
weitere_Aktionen;
```



## 2.5 Funktionen und Programmstruktur

### Aufbau eines Programms



## 2.5 Funktionen und Programmstruktur

### Aufbau eines Programms

**Preprozessor Anweisungen**

```
#include <stdio.h> /* Einbinden einer Bibliothek (I/O-Funktionen) */  
#include "modul_1.h" /* Einbinden eines Moduls „modul_1“ */  
  
#define COLOR_OF_EYES blau /* Globale Textersetzung im Programmtext, Makrodefinition */
```

### Globale Deklarationen und Definitionen:

```
int i; /* Deklaration */  
int j=13; /* Definition */  
int fakultaet (int n); /* Funktionsprototyp */
```

## 2.5 Funktionen und Programmstruktur

```
int fakultaet (int n) { /* Funktionsdefinition */
...
}
```

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

- Jedes Programm enthält eine Funktion „main“ von Typ void
- Unterprogramme werden in C in Form von **Funktionen** definiert, die **Parameterlisten** und **Ergebnistypen** haben.
- Die Abarbeitung eines Programms beginnt stets mit der Ausführung von „main“. Innerhalb von „main“ können die weiteren definierten Funktionen aufgerufen werden.



## 2.5 Funktionen und Programmstruktur

- Funktionen werden global definiert
- rekursive Funktionsaufrufe sind zulässig: eine Funktion darf sich selbst aufrufen
- Beispiel (Fakultätsberechnung):

```
fakultaet(int n){
    if ( n == 1 )
        return(1);
    else
        return( n * fakultaet(n-1) );
}
```



## Parameterübergabe an Funktionen

### □ call by value

- Normalfall in C
- Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
- Funktionen können den Wert der Kopien ändern, ohne Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer.

### □ call by reference

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen.
- Wenn die Funktion den Wert des Parameters verändert, ändert sie den Inhalt der zugehörigen Speicherzelle → der Wert der Variablen beim Aufrufer der Funktion ändert sich auch



## Globale und Lokale Variablen

- **Globale Variablen** sind im gesamten Programm einschließlich aller Unterprogramme bekannt. (Sie sollten vermieden werden)
- **Lokale Variablen** sind innerhalb einer Funktion oder eines „Blockes“ deklarierte Variable.

```
#include <stdio.h>

int global = 0; /* Das ist eine globale Variable */

main (){
    int lokal = 1; /* Das ist eine Lokale Variable in main */
    printf("Global %d, Lokal %d\n", global, lokal);
    {
        int lokal = 2; /* Das ist eine lokale Variable im Block */
        printf("Global %d, Lokal %d\n", global, lokal);
    }
    printf("Global %d, Lokal %d\n", global, lokal);
}
```



## Speicherklassen

Definieren die Lebensdauer und den Gültigkeitsbereich (Sichtbarkeit) von Variablen und/oder Funktionen.

In C gibt es die Speicherklassen:

- **auto** wird für lokale Variablen eingesetzt. Objekte, die als auto gelten nur solange wie der Bereich, in dem sie definiert wurden.
- **register** wird verwendet, um eine lokale Variable in einem Register der CPU zu speichern. „register“ sollte nur für Variablen verwendet werden, auf die schnell zugegriffen werden muss, z. B. Zähler
- **static** wird bei der Definition globaler Variablen verwendet.
- **extern** definiert eine globale Variable, die in allen Programm-Modulen sichtbar ist.

