

## Inhalt

- typedef
- Funktionen
- Funktionszeiger
- Präprozessor
- Dateistruktur

## typedef

- mit **typedef** können Typnamen erstellt werden, z.B. um die Schreibweise abzukürzen
- der neue Typname ist kein neuer Typ, nur ein neuer Name für einen existierenden Typ

Beispiele:

```
typedef unsigned long ulo;
```

```
ulo a, b, c;    // 3 unsigned long
```

```
ulo* pa = &a;   // Zeiger auf ulo ...
```

```
ulo feld[42];   // Feld von ulos
```

**typedef** wird häufig bei **structs**, **unions** und **enums** verwendet, da diese keywords in C Teil des Typnamens sind und ständig wiederholt werden müssen:

```
struct Str { int i; float f; };  
union Uni { char c; double d; };  
enum Bool { falsch, wahr };  
└────────┘ └────────────────┘
```

Typen

int-Konstanten mit Werten 0, 1

```
struct Str a, b, c;  
union Uni d, e, f;  
enum Bool g, h, i;  
└────────┘
```

Variablen

Jetzt mit **typedef** :

a) Verwendung der Typen von S. 3:

```
typedef struct Str  StrTy;  
typedef union  Uni  UniTy;  
typedef enum Bool  BoolTy;
```

neue Typbezeichner

// Variablen definieren:

```
StrTy x;  UniTy y;  BoolTy z;
```

b) ohne Verwendung der Typen von S. 3:

```
typedef struct {int i; float f; }      StrTy;  
typedef union  { char c; double d; }  UniTy;  
typedef enum   ↑ { falsch, wahr }     BoolTy;
```

ohne Bezeichner Str, Uni, Bool

// Variablen definieren wie oben:

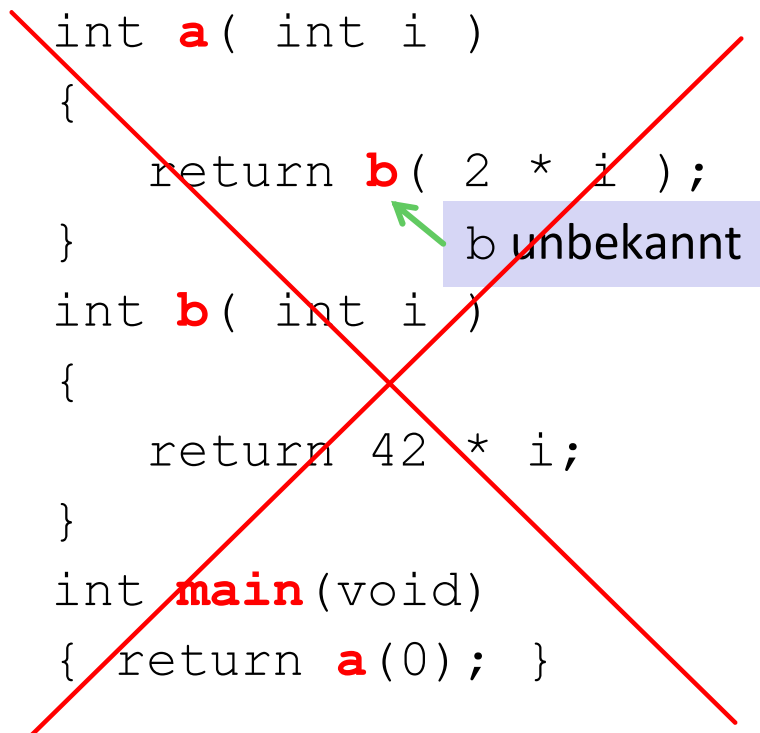

```
StrTy x;  UniTy y;  BoolTy z;
```

# Funktionen


- es gibt keine Funktionshierarchie
- die Reihenfolge von Funktionsdefinitionen im Quelltext ist beliebig, aber:


Vor dem ersten Aufruf einer Funktion muss mindestens ihre Deklaration (Prototyp, Vorwärtsdeklaration) bekannt sein.


```
int a( int i )  
{  
    return b( 2 * i );  
}  
int b( int i )  
{  
    return 42 * i;  
}  
int main(void)  
{ return a(0); }
```

  b unbekannt

```
int b( int i );  
int a( int i )  
{ ... }  
  
int main(void)  
{ return a(0); }  
  
int b( int i )  
{ ... }
```

 optional

 Prototyp

 Definitionen

- Die Parameterübergabe erfolgt ausschließlich ***by-value***.
- Mit Zeiger-Parametern wird ***by-reference*** Verhalten nachgebildet. Die Zeiger selbst sind ***by-value*** Parameter.

```
void f( int i )
{
    i = 21 * i;
}

int main(void)
{
    int j = 2;
    f(j);
    printf("%d", j );
    return 0;
}
```

Ausgabe: 2

```
void f( int* pi )
{
    *pi = 21 * *pi;
}

int main(void)
{
    int j = 2;
    f(&j);
    printf("%d", j );
    return 0;
}
```

Ausgabe: 42

- Der **return-Wert** wird an der Stelle des Funktionsaufrufs eingesetzt – als Wert.
- **return** kann Zeigerwerte liefern, allerdings sollte nur auf Objekte verwiesen werden, die den Funktionsaufruf überleben:
  - keine Zeiger auf lokale Variablen bzw. Parameter
  - nur Zeiger auf statics oder Heap-Objekte
- Arrays können nicht als Wert zurückgegeben werden. Nur der Zeiger auf ein static- oder Heap-Array kann Funktionsergebnis sein.

## Zeiger auf Funktionen:

- Ein Funktionsbezeichner repräsentiert die **Startadresse** der Funktion. Mit einem Zeiger auf diese Adresse kann die Funktion aufgerufen oder als *Parameter an eine andere Funktion* übergeben werden.
- Für die Typisierung eines Funktionszeigers ist die gesamte Funktionssignatur signifikant.

### Beispiel:

Zwei Funktionen mit identischer Signatur seien gegeben:

```
char machWatt1(int i, float f);  
char machWatt2(int i, float f);
```



Deklaration eines Zeigers **zfunc**, der auf alle Funktionen mit dieser Signatur zeigen kann:

```
char  (*zfunc) (int, float) = machWatt1;
```

I                      II                      III                      IV

- I) Rückgabetyp der Funktion(en)
- II) *\*Zeigername* muss geklammert werden, da sonst eine normale Funktionsdeklaration vom Typ **char\*** entsteht
- III) Parametertypen der Funktion(en)
- IV) optionale Initialisierung mit der Funktionsadresse  
(alternativ: **zfunc = machWatt1;** )

## Funktionsaufruf per Zeiger:

```
(*zfunc) (5, 5.5f) ;
```

( identisch zum Aufruf: `machWatt1 (5, 5.5f) ;` )

## Funktionszeiger als Parameter (z.B. bei callbacks):

```
char tuWatt( char (*zf) (int, float) )  
{  
    return (*zf) (1, 1.1f) ;  
}
```

**tuWatt** Aufrufe:

```
tuWatt(zfunc) ;      oder
```

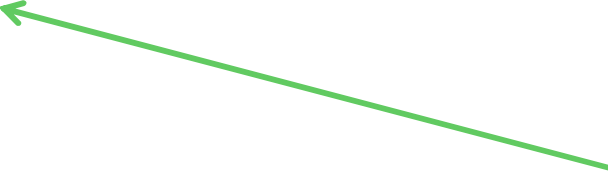
```
tuWatt(machWatt1) ;
```

```
zfunc = machWatt2 ;
```

```
tuWatt(zfunc) ;      oder
```

```
tuWatt(machWatt2) ;
```

Aufruf von `machWattx`



# Präprozessor

siehe z.B.: <http://www.tenouk.com/Module10.html>

- Der Präprozessor wird vor dem Compiler ausgeführt.
- Das Ergebnis – eine Datei – ist Eingabe für den Compiler.
- Verwendung:
  - Quelltextimport (`#include`)
  - bedingter Import oder bedingte Compilierung (`#if`, `#ifdef`, `#elif`, ...)
  - Angabe systemabhängiger Compiler/Linker-Direktiven (`#pragma`)
  - Quelltextersatz (`#define`)
    - symbolische Konstanten (`const` erst ab ANSI C)
    - Makro-Funktionen

## Präprozessor Beispiele

Quelltextimport:

```
#include <math.h>
```

Suche in  
systemabhängigen Pfaden

```
#include "meineHeaderDatei.h"
```

Suche beginnt im  
Verzeichnis dieser Datei

```
#include "../header/meineHeaderDatei.h"
```

```
#include "/usr/.../header/meineHeaderDatei.h"
```

bedingter Quelltextimport, z.B. in der Datei `meineHeaderDatei.h`:

```
#ifndef __42__
```

„if not defined“

```
#define __42__
```

```
int machWatt();
```

```
.
```

```
.
```

```
.
```

```
#endif
```

Selbst definierte Importdateien sollten immer gegen Mehrfachimport abgesichert werden.

Die mit `< >` importierten Dateien werden i.A. nur einmal inkludiert. Sie enthalten bereits die `#ifndef`-Konstruktion.

Symbolische Konstanten und Makros:

```
#define PI 3.14159  
#define kreisFlaeche(r) PI*(r)*(r)
```

```
// Aufruf:  
double f = kreisFlaeche(42);
```

Dies wird vom Präprozessor expandiert zu:

```
double f = 3.14159*42*42;
```

Diese Konstruktionen möglichst nicht mehr verwenden:

- auch C kennt inzwischen den **const** specifier
- Makro-Funktionen ersparen den Funktionsaufruf.  
Seit der Einführung von **inline**-Funktionen entfällt dieses Argument jedoch.

## Dateistruktur

- C-Quelltexte werden üblicherweise aufgeteilt in:
  - Deklarationsdateien, *header-Dateien*, **xyz . h**  
(Prototypen, globale Konstanten, Variablen, Typen, Klassendeklarationen in C++)
  - Definitionsdateien, **xyz . c**  
(Funktionsdefinitionen (*bodies*), C++ Klassendefinitionen)
- Nur die header-Dateien werden als Quelltext für die Compilation mit `include` importiert.  
(Header-Dateien enthalten oft weitere includes ... )
- Die Definitionen werden als Objektdateien vom Linker eingebunden.
- Die Einbindung der Objektdateien ist hochgradig abhängig von: Betriebssystem, Compiler/Linker, Entwicklungsumgebung, ...