



## Algorithmen und Programmierung

### 3. Kapitel Typen und Speicher

Prof. Matthias Werner

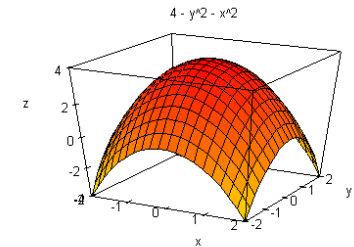
Professur Betriebssysteme

## Deklaration von Funktionen

- ▶ Auch in C haben Funktionen einen Definitions- und einen Wertebereich
    - ▶ Wir haben bereits das Schlüsselwort `int` für die Menge der ganzen Zahlen kennengelernt
  - ▶ Zur Erinnerung: Eine Funktion in C muss **deklariert** werden
    - ▶ Ausnahme: Sie wird **vor** ihrer ersten Nutzung definiert
  - ▶ Ein Deklaration enthalten
    - ▶ den Namen
    - ▶ den Wertebereich
    - ▶ den Definitionsbereich
- ```
int euclid(int, int)
```
- ▶ Darüber hinaus **kann** die Deklaration noch andere Dinge enthalten (später mehr)
  - ▶ Mathematisch ausgedrückt:  $\text{euclid}: \text{int} \times \text{int} \rightarrow \text{int}$

## 3.1 Typen

- ▶ Erinnerung an Mathematik-Unterricht: Jede Funktion hat einen **Definitionsbereich**<sup>1</sup> und einen **Wertebereich**<sup>2</sup>
- ▶ Beides sind definierte **Mengen**
- ▶ **Beispiel:** die Funktion  $z = f(x, y) = 4 - x^2 - y^2$ , wie sie (auszugsweise) im Graph zu sehen ist, hat einen Definitionsbereich  $D \subseteq \mathbb{R} \times \mathbb{R}$  und einen Wertebereich  $C \subset \mathbb{R}$
- ▶ **Schreibweise:**  $f: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$



<sup>1</sup>auch: Quellmenge, Domain  
<sup>2</sup>auch: Zielmenge, Codomain

## Typ und Signatur

- ▶ „ $\text{int} \times \text{int} \rightarrow \text{int}$ “ ist der **Typ** der Funktion „`euclid`“

### Achtung!

Unter C-Programmierern hat es sich eingebürgert, **nur** den **Wertebereich** als Typ zu betrachten. Das ist vollständig okay für C, da bestimmte Aspekte von Typen in C keine Rolle spielen. In anderen Sprachen ist dieser Unterschied aber wesentlich.

- ▶ Im verkürzenden C-Jargon hätte „`euclid`“ also den Typ „`int`“, korrekter wäre der Begriff „**Rückgabotyp**“
- ▶ Der Typ bildet zusammen mit dem Namen die **Signatur** einer Funktion
  - ➔ Die Deklaration macht also den Compiler mit der Signatur einer Funktion bekannt

## Typ und Signatur (Forts.)

### Wofür braucht man Typen und Signaturen überhaupt?

1. Viele Operationen (z.B. Addition  $\rightarrow$  „+“) sehen zwar immer gleich aus, werden aber unterschiedlich durchgeführt, je nachdem welchen Typ man betrachtet
  - ▶ Beispiel: natürliche vs. komplexe Zahlen

Typisierung hilft dem Compiler, die richtige Methode zu finden
2. Typen können dem Programmierer helfen, Denkfehler zu verhindern
  - ▶ Nicht alle Programmiersprachen verlangen Deklaration von Typen  $\rightarrow$  andere Sprachen können Typen automatisch ableiten (**Typinferenz**)
  - ▶ C ist relativ nachlässig mit den Konsequenzen aus der Typisierung andere Sprachen sind da konsequenter (**stark typisiert**)

- ▶ Warum gibt es für z.B. Ganzzahlen unterschiedliche Typen?
- $\rightarrow$  Unterschiedlich große Werte brauchen unterschiedlich viel Speicherplatz
- ▶ Wieviel?
- $\rightarrow$  Der Computer speichert Werte mit Hilfe von Bits (= *binary digits*)
  - ▶ In  $n$  Bits können also  $2^n$  verschiedene Werte gespeichert werden
  - ▶ Typischerweise ist die kleinste adressierbare Einheit ein **Byte** (=8 Bit)
  - ▶ Werte werden deshalb in einem oder mehreren Bytes gespeichert
- ▶ Die Wahl des „richtigen“ Typs ist deshalb immer ein Kompromiss zwischen der **benötigten Anzahl** unterschiedlicher Werte und dem **Speicherverbrauch** (und der Zugriffsgeschwindigkeit)

## Grundtypen

- ▶ In C sind nur wenige Möglichkeiten für die Beschreibung von Definitions- und Wertebereich (Grundtypen) enthalten
  - ▶ Allerdings bietet C die Möglichkeit, weitere Typen „zu bauen“
- ▶ Die Mengen, die durch Typen beschrieben werden, sind in C nicht unendlich, bilden also z.B. nur eine Teilmenge der Menge natürlichen Zahlen ( $\mathbb{N}$ )
- ▶ C99 kennt folgende Grundtypen
  - ▶ Teilmengen der **Ganzzahlwerte** ( $\mathbb{Z}$ )
  - ▶ Teilmengen der **rationalen Zahlenwerte** ( $\mathbb{Q}$ )
  - ▶ Menge der **booleschen Werte** (vollständig! 😊)
  - ▶ Teilmengen der **komplexen Zahlenwerte** ( $\mathbb{C}$ )
  - ▶ leere Menge
  - ▶ Speicheradressen<sup>3</sup>
- ▶ Im Sprachumfang von C90 gab es weder boolesche noch komplexe Zahlen, sie ließen sich aber leicht „nachbauen“ und waren in Bibliotheken vorhanden.

<sup>3</sup>Präziser: Adressen sind abgeleitete Typen.

## Ganze Zahlen

- ▶ C stellt folgende Schlüsselwörter zur Beschreibung von Ganzzahltypen zur Verfügung:

| Basis | Größe | Vorzeichen |
|-------|-------|------------|
| int   | short | signed     |
| char  | long  | unsigned   |

- ▶ **int** kann durch die Schlüsselwörter **short**, **long**, **signed** und **unsigned** modifiziert werden
- ▶ **char** kann durch die Schlüsselwörter **signed** und **unsigned** modifiziert werden
- ▶ Wenn **int** modifiziert wird, kann das Schlüsselwort **int** selbst weggelassen werden
- ▶ Insgesamt lassen sich die folgenden Ganzzahl-Typen bilden: **char**, **unsigned char**, **signed char**, **int**, **unsigned int**, **signed int**, **short int**, **unsigned short int**, **signed short int**, **short**, **unsigned short**, **signed short**, **long int**, **unsigned long int**, **signed long int**, **long**, **unsigned long**, **signed long**, **long long int**<sup>4</sup>, **long long**<sup>4</sup>

<sup>4</sup>nicht vor C99

## Ganze Zahlen (Forts.)

Es gelten folgende Regeln:

- ▶ Alle Typen mit `unsigned` im Namen sind vorzeichenlos
- ▶ Alle Typen mit `signed` im Namen sind vorzeichenbehaftet
- ▶ Bei `int` entspricht der Typ ohne Vorzeichen-Modifikation einem vorzeichenbehafteten Typ
- ▶ Bei `char` ist dies dem Compiler überlassen
- ▶ **Es sind keine feste Wertebereiche für die meisten Typen definiert!**
  - ▶ Allerdings gilt:
    - ▶  $card(char) = 256$
    - ▶  $card(short\ int) \geq 65536$
    - ▶  $card(int) \geq \max(65536, card(short\ int))$
    - ▶  $card(long\ int) \geq \max(4294967296, card(int))$
    - ▶  $card(long\ long\ int) \geq \max(18446744073709551616, card(long\ int))$

## Darstellung

- ▶ Der C90-Standard legt nicht fest, wie ein Wert im Speicher „aussieht“, d.h., welches Bit-Muster welchem Wert entspricht
- ▶ Allerdings nutzen alle gängige Plattformen ein **binäres Positionssystem**, mit **Zweierkomplement** für die Darstellung negativer Zahlen

vorzeichenlos: Wert =  $\sum_{i=1}^n b_i \cdot 2^{i-1}$

$$\text{vorzeichenbehaftet: Wert} = \begin{cases} \sum_{i=1}^{n-1} b_i \cdot 2^i & \text{wenn } b_n = 0 \\ \left( - \sum_{i=1}^{n-1} (1 - b_i) \cdot 2^i \right) - 1 & \text{wenn } b_n = 1 \end{cases}$$

$b_i$  ist dabei das  $i$ . Bit

- ▶ Mehr in  $\Rightarrow$  LV „Grundlagen Technische Informatik“ (Kapitel 2) oder „Einführung in die Funktion von Computern“ (Kapitel 1)

## Ganze Zahlen (Forts.)

- ▶ Seit C99 gibt es die Headerdatei „`stdint.h`“
- ▶ Darin sind Ganzzahltypen mit fester Bitbreite definiert, so sie auf der jeweiligen Plattform zur Verfügung stehen:
  - ▶ `int8_t` und `uint8_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit genau 8 Bit (d.h., die Kardinalität ist 256)
  - ▶ `int16_t` und `uint16_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit genau 16 Bit (d.h., die Kardinalität ist 65536)
  - ▶ `int32_t` und `uint32_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit genau 32 Bit (d.h., die Kardinalität ist 4294967296)
  - ▶ `int64_t` und `uint64_t`: vorzeichenbehaftete und vorzeichenlose Ganzzahlen mit genau 64 Bit (d.h., die Kardinalität ist 18446744073709551616)

## Zeichen

Warum heißt ein Ganzzahltyp „`char`“?

- ▶ Ursprünglich gedacht für Werte im Bereich des **ASCII-Codes** (7 Bit) bzw. des **ANSI-Codes** (8 Bit)  $\Rightarrow$  *Character*
- ▶ C kennt (anders als andere Sprachen) **keinen** expliziten Typ für Zeichen
  - ▶ Verwendung liegt **ausschließlich** in der **Interpretation** des Wertes einer Ganzzahl bei der Ausgabe, z.B. entsprechend einem Code
  - ▶ Je nach Plattform kann daher der gleiche Wert für unterschiedliche Zeichen stehen
- ▶ Heute sollte für Zeichen vorwiegend der Typ `wchar_t` benutzt werden
  - ▶ Vereinfacht Internationalisierung
  - ▶ Einbindung über Headerdatei `wchar.h`
- ▶ Wir werden aber zur Vereinfachung bis auf Weiteres von ASCII/ANSI-Code ausgehen

## Zeichen (Forts.)

```

/* char.c -- interpretation of char type */
#include<stdio.h>

char addchar(char c1, char c2)
{
    return c1+c2;
}

int main()
{
    printf("Ergebnis ist %c mit dem Code %d\n",
        addchar('a',1),addchar('a',1));
    return 0;
}
    
```

Interpretation

↓

↘

```
> ./char Ergebnis ist b mit dem Code 98
```

## Rationale Zahlen (Forts.)

- ▶ Rationale Zahlen in C haben außer einem beschränkten Wertebereich eine beschränkte Präzision
- ▶ Wieder wird offiziell nichts festgelegt, außer Mindestwerten
  - ▶ `float` besitzt einen Wertebereich von mindestens  $\pm 10^{\pm 37}$  und eine Präzision von mindestens 6 Dezimalstellen
  - ▶ `double` besitzt mindestens den Wertebereich von `float` und eine Präzision von mindestens 10 Dezimalstellen
  - ▶ `long double` ist wenigstens so „gut“ wie `double`
- ▶ De facto folgen aber praktisch alle Compiler dem IEEE-754-Standard, der binäre Darstellungen für Gleitkommazahlen spezifiziert

## Rationale Zahlen

- ▶ Durch die interne Darstellung kann C nur rationale Zahlen ( $\mathbb{Q}$ ) beschreiben, keine reellen Zahlen ( $\mathbb{R}$ )
- ▶ Diese Zahlentypen werden nach der verbreiteten „wissenschaftlichen Schreibweise“ auch **Gleitkommazahlen** (*floating point numbers*) genannt

$$1234,5 = 1,2345 \cdot 10^3 = 1,2345E3$$

- ▶ C stellt folgende Typen zur Beschreibung von Gleitkommazahlen zur Verfügung:
  - ▶ `float`
  - ▶ `double`
  - ▶ `long double`

## Der Typ void

- ▶ C kennt den Basistyp `void` („void“  $\Rightarrow$  engl. leer, nichtig)
- ▶ `void` ist eigentlich ein Anti-Typ: er wird dann genutzt, wenn man **keinen** Typ haben will  $\Rightarrow$  leere Menge
- ▶ **Wozu ist das gut?**
  - ▶ C kennt nur **Funktionen** als Einheiten der Ausführung, also eine Abbildung von einer Menge ( $\Rightarrow$  Funktionsparameter) in eine andere ( $\Rightarrow$  Rückgabewert)
  - ▶ Mitunter kommt es nur auf die **Ausführung** an, und Parameter oder/und Rückgabewert wird nicht gebraucht.

## Der Typ void (Forts.)

```

#include<stdio.h>

void say_something(void)
{
    printf("Something!\n");
    return;
}

int main()
{
    say_something();
    return 0;
}
    
```

- ▶ Bei Funktionen mit Rückgabtyp `void` kann `return` weggelassen werden.
- ▶ Ist das einzige Element der Parameterliste `void`, so kann es weggelassen werden

## Typenkonvertierung (Forts.)

Man betrachte den folgenden Code:

```

1 /* cast.c -- type cast */
2 int printf(const char*,...);
3 long ladd(long, long);
4
5 int main()
6 {
7     printf("Ergebnis: %d\n", ladd(23,42));
8     return 0;
9 }
10
11 long ladd(long x, long y)
12 {
13     return x+y;
14 }
    
```

- ▶ Bei der Übersetzung gibt es folgende Warnung:

```

> cc -std=C99 -Wall -pedantic -Wall -Wextra cast.c -o cast
cast.c:7:28: warning: format specifies type 'int' but the argument has type 'long' [-Wformat]
    
```

## Typenkonvertierung

- ▶ In Ausdrücken (z.B. `a + b`) können verschieden Teilausdrücke verschiedene Typen haben, z.B. `unsigned char` und `signed int`
- ▶ In diesen Fällen werden Teilausdrücke **automatisch konvertiert**
  - ▶ Allgemein gilt: es wird stets auf den höher auflösenden Wert konvertiert, mindestens auf `int`
- ▶ Neben der automatischen Konvertierung kann auch eine Konvertierung erzwungen werden, indem der Zieltyp in Klammern vor den Ausdruck gesetzt wird
  - ▶ Man spricht von einem **Cast-Operator** (*to cast* = in eine Form gießen)
  - ▶ Dabei kann auch ein kleinerer Wert erzwungen werden

## Typenkonvertierung (Forts.)

- ▶ Die Warnung verschwindet, wenn man in Zeile 7 den Compiler anweist, den Wert als `int` zu interpretieren

```

1 /* cast.c -- type cast */
2 int printf(const char*,...);
3 long ladd(long, long);
4
5 int main()
6 {
7     printf("Ergebnis: %d\n", (int)ladd(23,42));
8     return 0;
9 }
10
11 long ladd(long x, long y)
12 {
13     return x+y;
14 }
    
```

## Literale

- ▶ Häufig kommen im Programmtext **Werte** eines bestimmten Typs vor
- ▶ Einen solchen direkten Wert nennt man ein **Literal**
- ▶ Literale eines bestimmten Typs verlangen eine bestimmte Schreibweise, sonst wird eine (unnötige) Konvertierung vorgenommen

int:           Ganzahlwerte können im **Dezimal-**, **Oktal-** oder **Hexadezimalsystem** geschrieben werden, jeweils ggf. mit Vorzeichen

- ▶ **Dezimalwert:** nur Ziffern 0...9, **nicht** mit 0 beginnend
  - ▶ z.B. 42
- ▶ **Oktalwert:** Präfix 0, dann nur Ziffern 0...7
  - ▶ z.B. 052
- ▶ **Hexadezimalsystem:** Präfix 0x dann nur Ziffern 0...9 und Buchstaben a...f bzw. A...F
  - ▶ z.B. 0x2a

char:           Als **Zeichen** in Hochkommata

- ▶ z.B. '\*'

## Zeichenlitterale

Für nichtdruckbare Zeichen gibt es (sowohl für `char` als auch für `wchar_t`) Ersatzsequenzen:

|               |                 |                         |                   |
|---------------|-----------------|-------------------------|-------------------|
| Newline       | <code>\n</code> | horizontaler Tabulator  | <code>\t</code>   |
| Backspace     | <code>\b</code> | vertikaler Tabulator    | <code>\v</code>   |
| Wagenrücklauf | <code>\r</code> | Seitenvorschub          | <code>\f</code>   |
| Hochkomma (') | <code>\'</code> | Anführungsstiche (")    | <code>\"</code>   |
| Piep          | <code>\a</code> | Rückstrich (\)          | <code>\\</code>   |
| NUL           | <code>\0</code> | Zeichen mit Hex-Code hh | <code>\xhh</code> |

## Literale (Forts.)

unsigned int: **Wie** `int`, aber mit Suffix „u“ bzw. „U“

- ▶ z.B. 123U

long/unsigned long: **Wie** `int/unsigned int` mit Suffix „L“ bzw. „l“

- ▶ z.B. 123L, 123UL

long long/unsigned long long: **Wie** `int/unsigned int` mit Suffix „LL“ bzw. „ll“

- ▶ z.B. 123LL, 123ULL

wchar\_t:       **Wie** `char` mit Präfix „L“

- ▶ z.B. L'a'

## Literale (Forts.)

double: **Nur** als Dezimalzahl mit Ganzzahlteil, Dezimalpunkt, Nachkommateil, Buchstabe „e“ oder „E“ und Exponent

- ▶ Ganz- oder Nachkommateil (nicht beides) kann ausgelassen werden
- ▶ Dezimalpunkt oder Buchstabe+Exponent (nicht beides) kann ausgelassen werden
- ▶ z.B. 1.23e10, .23 oder 1e10

float: **Wie** `double` mit Suffix „f“ oder „F“

- ▶ z.B. 1.23e5f, .23F oder 1e5f

long double: **Wie** `double` mit Suffix „l“ oder „L“

- ▶ z.B. 1.23e5L, .23l oder 1e5L

### Achtung

Bei Gleitkommazahlen wird (entsprechend der englischen Sprache) ein Punkt „.“ als Dezimalzeichen benutzt, nicht (wie im Deutschen) ein Komma „,“.

Der Code `double pi; pi=3,14;` ist in C syntaktisch korrekt (und gibt daher i.d.R. keine Warnung), bewirkt aber, dass `pi` den Wert 3 erhält.

## Dynamische Typen und Typinferenz

- ▶ Eine Deklaration wie in C ist nicht in allen Sprachen notwendig
  - ▶ In andere Sprachen mit statischen Typen kann der Compiler den Typ ableiten (Typinferenz) → z.B. Haskell, ML
  - ▶ Andere Sprachen besitzen **dynamische Typen**, die zur Laufzeit bestimmt werden → z.B. Python

```
>>> def foo(x):
...     return 2*x
...
>>> type(42)
<type 'int'>
>>> type(foo)
<type 'function'>
>>> type(foo(42))
<type 'int'>
>>> type(foo(4.2))
<type 'float'>
>>> type(foo('Hallo'))
<type 'str'>
>>>
```

## 3.2 Speicher

- ▶ Bisher folgten unsere C-Programme im Wesentlichen dem Funktionsmodell → **zustandsfrei**
- ▶ Zur Realisierung des Zustandsmodell ist die Speicherung von Daten notwendig
- ▶ Prinzipiell zwei Möglichkeiten:
  - ▶ Speicherung im **Dateisystem** (gut für große Datenmengen, aber langsam)
  - ▶ Speicherung im **Hauptspeicher** (schneller, aber beschränkter)
- ▶ Betrachten hier letzteren Ansatz

### Achtung!

C verlangt, dass man sich vielen Fällen explizit Gedanken über Speicherung macht. Bei anderen Sprachen ist dies nicht so, da sie

- ▶ eine automatische Speicherverwaltung besitzen (Logo, Python) und/oder
- ▶ das Zustandsmodell gar nicht erst unterstützen (Haskell, ML)

## Python

- ▶ Python kennt auch Ganzzahlen und Gleitkommazahlen
- ▶ Darüber hinaus besitzt es Grundtypen, die es in C nicht gibt, z.B.:
  - ▶ Zahlen mit beliebiger Größe und Genauigkeit
  - ▶ Zeichenketten (Strings)

### Merke

Falls es nicht auf Geschwindigkeit ankommt, eignet sich Python bei Problemen wie Textmanipulation oft besser als C.

- ▶ Größer sind die Unterschiede bei den abgeleiteten Typen → später

## Variablen

- ▶ Prinzipiell kann in C jede Stelle des Hauptspeichers gelesen und geschrieben werden
- ▶ **Dies ist jedoch gefährlich und wird von vielen Betriebssystemen verhindert**
- ▶ Daher gibt es Möglichkeiten, Speicher(bereiche) zu **reservieren**
- ▶ Drei prinzipielle Ansätze:
  - ▶ **benannten Speicher**, Reservierung zur Übersetzungszeit
  - ▶ **anonymen Speicher**, Reservierung zur Laufzeit
  - ▶ Parameter, Reservierung zur Laufzeit

### Variablen und Konstanten

Soll der Speicherinhalt eines reservierten Speicherbereichs geändert werden (was genau Sinn des Zustandsmodells ist), nennt man den Speicherbereich eine (benannte oder anonyme) **Variable**.  
Soll der Speicherinhalt dagegen unverändert bleiben, spricht man von einer (Laufzeit-) **Konstanten**.

### 3.2.1 Benannte Variablen

- ▶ Benannte Variablen müssen in C deklariert werden
- ▶ Deklaration einer Variable erfolgt analog zur Deklaration einer Funktion
- ▶ Die Deklaration **muss** enthalten:
  - ▶ Typ der Variablen
  - ▶ Name der Variablen

```
int ganzzahl;
```

*Typ Name*

- ▶ Mehrere Variablen des gleichen Typs können gemeinsam deklariert werden

```
int x, y, z;
```

### Deklaration und Definition (Forts.)

- ▶ Die Deklaration einer Variablen kann erfolgen:
  - ▶ **Außerhalb** jeder Funktion → **globale Variable**
  - ▶ Innerhalb<sup>5</sup> jedes **Blockes** aus geschweiften Klammern „{ ... }“ → **lokale Variable**
- ▶ Mit Ausnahme von Deklarationen, die die Speicherklasse „extern“ tragen, werden benannte Variablen bei der Deklaration auch **definiert**, d.h. Speicherplatz reserviert
- ▶ Ist bei einer Deklaration einer Variablen die Speicherklasse „extern“ angegeben, so wird die Variable **nur deklariert** (dem Compiler bekannt gegeben)
  - ▶ Sie muss dann noch woanders (in einem anderen Modul, vergleiche Kapitel 11) **global** definiert werden

<sup>5</sup>Bis C90 dürfen Variablendeklarationen nur an **Beginn** eines Blocks stehen, d.h. **vor** allen anderen Anweisungen in diesem Block.

### Deklaration und Definition

- ▶ Bei einer Variablen-Deklaration **kann** zusätzlich eine **Speicherklasse** (*storage class*) und/oder **Speicherattribut** (*type qualifier*) für die Variable angeben
- ▶ Speicherklasse wird durch eines der Schlüsselwörter
  - ▶ **auto**, **static**, **extern** oder **register**
 deklariert
- ▶ Die Speicherklasse hat Auswirkungen auf **Lebensdauer** und **modulübergreifende Verwendbarkeit** (*linkage*)
  - ▶ Wird keine Speicherklasse angegeben, wird „auto“ angenommen → automatische Variable
  - ▶ Später mehr...
- ▶ Speicherattribute werden durch die Schlüsselwörter
  - ▶ **const**, **volatile** oder **restrict**
 deklariert
- ▶ Speicherattribute gibt dem Compiler Hinweise über die **Verwendung** der Variablen
  - ▶ Auch hier später mehr...

### Bezeichner

#### Bezeichner

Die Namen von Funktionen, Variablen und Typen werden **Bezeichner** (**Identifier**) genannt.

Jeder Bezeichner darf global oder innerhalb eines Blockes jeweils **nur einmal** definiert sein.

Ein Bezeichner muss mit einem Buchstaben oder dem Unterstrich „\_“ anfangen, dem eine beliebige Kombination aus Buchstaben, Zahlen oder Unterstrichen folgen kann.

Ein Schlüsselwort darf nicht als Bezeichner verwendet werden. Auch empfiehlt es sich nicht, Bezeichner aus der Standardbibliothek zu benutzen.

#### Achtung!

Es gibt Sprachen, bei denen es auf Groß-/Kleinschreibung nicht ankommt. In C sehr wohl: *abc*, *Abc*, *ABC* und *aBC* sind **unterschiedliche** Bezeichner.



## Zuweisung

- ▶ Einer Variable kann man mit dem „=“-Operator einen Wert zuweisen
- ▶ Dies geht auch schon bei der Deklaration → **Initialisierung**

```
int x=0;
x=42;
```

- ▶ Bei der Zuweisung kann auch auf den vorherigen Wert der Variable Bezug genommen werden

```
x=2*x; // vor Zuweisung
```

nach Zuweisung

## Lebensdauer

Jede benannte Variable hat eine **Lebensdauer (Gültigkeit)**, in der sie den jeweils letzten zugewiesenen Wert behält

Die Lebensdauer...

- ▶ ... einer **globalen Variable** ist die gesamte Laufzeit des Programms
- ▶ ... einer **automatischen lokalen Variable** die Zeit, in dem sich das Programm in dem Block befinden, in dem sie deklariert ist
- ▶ ... einer lokalen Variable der Speicherklasse „**static**“ (→ **statische Variable**) die gesamte Laufzeit des Programms

### Achtung!

Statische lokale Variablen „überleben“ das Verlassen einer Funktion.

Wird eine statische Variable bei der Deklaration initialisiert, so wird diese Initialisierung **nur einmal** vorgenommen.

## Sichtbarkeit

- ▶ Jede benannte Variable ist ab der Stelle, an der sie deklariert ist, **sichtbar**
- ▶ Sie verliert ihre Sichtbarkeit, wenn
  - ▶ der Block, in dem sie deklariert wurde, endet
  - ▶ wenn in einem tiefer verschachtelten Block ein gleichlautender Bezeichner deklariert wird → **Überdeckung**

```
#include<stdio.h>

int global=42;

int main()
{
    int local1,local2;
    local1=111;
    local2=222;
    {
        int local1=23;
        printf("global=%d, local1=%d, local2=%d \n", global,local1,local2);
    }
    printf("global=%d, local1=%d, local2=%d \n", global,local1,local2);
    return 0;
}
```

## Lebensdauer (Forts.)

```
/* static.c -- lifetime of static variables */
#include<stdio.h>

void count(void){
    static int remember=1;
    printf("remember=%d\n",remember);
    remember=remember+1;
}

int main()
{
    count();
    count();
    count();
    return 0;
}
```

```
> ./static
remember=1
remember=2
remember=3
```

## Adressen

- ▶ Jede Variable hat eine **Adresse**
- ▶ Der Operator **&** gibt die Adresse einer Variablen zurück

```

/* addr.c -- address of a variable */
#include<stdio.h>

int main()
{
    int var=42;
    printf("Variable var has the address %p and the value %d\n",
        &var, var);
    return 0;
}
    
```

```

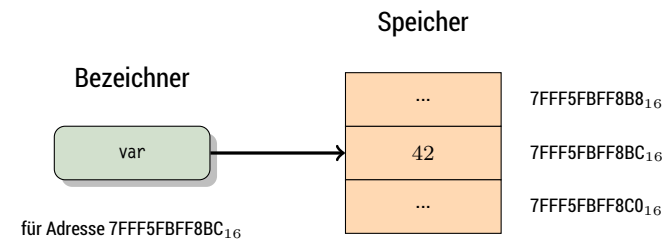
> cc -std=c99 -Wall addr.c -o addr
> ./addr
Variable var has the address 0x7fff5fbff8bc and the value 42
    
```

## Zeiger

- ▶ Variablen können auch selbst Adressen beinhalten
- ▶ Durch den Präfix „**\***“ vor dem Bezeichner wird eine Variable zu einer Adressvariablen (**Zeiger, Pointer**)
- ▶ Es ist für den Compiler wesentlich, wovon eine Adresse gebildet wird
- ▶ Entsprechend ist ein Zeiger immer ein Zeiger auf einen **bestimmten** anderen Typ
- ▶ Z.B.:
  - ▶ `int *p;` ⇒ Zeiger auf eine Ganzzahl
  - ▶ `float *p;` ⇒ Zeiger auf eine Gleitkommazahl
  - ▶ `unsigned int *p;` ⇒ Zeiger auf eine natürliche Zahl
- ▶ Wenn nur allgemein eine Speicheradresse (ohne speziellen Typ) gemeint ist, wird als Grundtyp „**void**“ verwendet, z.B.
  - ▶ `void *p;` ⇒ Zeiger auf eine Adresse
  - ▶ Bei **Zuweisungen** sind **void**-Zeiger zu allen anderen Zeiger-Typen kompatibel ⇒ keine Warnung

## Adressen (Forts.)

- ▶ Tatsächlich ist der Bezeichner einer Variablen nur ein Synonym für eine Adresse



- ▶ Konkret wird in C ein Variablenbezeichner als (evtl. noch durch andere Operanden modifizierte) Adresse gesehen und wie folgt behandelt:
  - ▶ Er steht **links** eines Gleichheitszeichens ⇒ der Wert des Ausdrucks rechts des „**=**“ wird in die Speicherstelle mit dieser Adresse **geschrieben**
  - ▶ Sonst ⇒ der **Inhalt** (Wert) der Speicherstelle wird **gelesen** und weiterverarbeitet
  - ▶ Da C mehrere Gleichheitszeichen in einem Ausdruck erlaubt, kann auch beides geschehen

## Zeiger (Forts.)

```

/* addr2.c -- pointer to a variable */
#include<stdio.h>

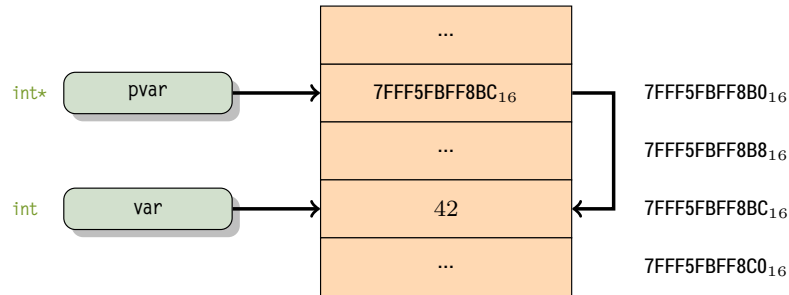
int main()
{
    int var=42, *pvar;
    pvar= &var ;
    printf("Variable var has the address %p and the value %d\n",
        pvar, var);
    printf("Variable pvar has the address %p and the value %p\n",
        &pvar, pvar);
    return 0;
}
    
```

- ▶ Nach Ignorieren von Warnungen erhält man bei Aufruf beispielsweise:

```

> ./addr2
Variable var has the address 0x7fff5fbff8bc and the value 42
Variable pvar has the address 0x7fff5fbff8b0 and the value 0x7fff5fbff8bc
    
```

## Zeiger (Forts.)



### Achtung!

Ein Typ und dessen abgeleiteter Zeigertyp (z.B. `int` und `int*`) sind **unterschiedliche** Typen!

## L-Value

- ▶ Bezeichner und Zeiger sind zwei Möglichkeiten, Speicheradressen von Variablen in C auszudrücken
- ▶ Wir werden noch mehr Möglichkeiten kennenlernen
- ▶ Daher gibt es ein allgemeines Konzept: L-Value

### Definition 3.1 (L-Value)

Jeder Ausdruck, der eine gültige **getypte** Adresse ergibt, wird **L-Value (L-Wert, Linkswert)** genannt.

- ▶ Die Bezeichnung steht für „left value“, da links einer Zuweisung solch ein Ausdruck stehen muss
- ▶ Eine ungetypte Adresse ist kein L-Value

## Dereferenzierung

- ▶ Ein Zeiger kann durch den „\*“-Operator vor dem Zeiger **dereferenziert** werden
- ▶ Damit wird der Inhalt der Speicherstelle angesprochen, auf die der Zeiger zeigt
- ▶ Für dereferenzierte Zeiger gelten in Zuweisungen die gleichen Regeln wie für einfache Bezeichner

```

/* deref.c -- deref a pointer */
#include<stdio.h>

int main()
{
    int y=23, *py=&y;

    printf("y=%d\n",*py); /* same effect as printf("y=%d\n",y); */
    *py=42;                /* same effect as y=42 */

    printf("y=%d\n",y);
    return 0;
}
    
```

## L-Value (Forts.)

```

/* lvalue.c -- this program does not compile! */
int main()
{
    int i,j;
    int *pi;
    void *pv;

    i = 42; /* correct */
    pi = &j; /* correct */
    pv = pi; /* correct */
    *pi = 23; /* correct */
    *pv = 23; /* NOT correct */
    42 = j; /* NOT correct */

    return 0;
}
    
```

- ▶ Bei diesem Programm gibt es Compilerfehler, da zweimal keine L-Values auf der linken Seite einer Zuweisung stehen

## Python

- ▶ Die Regeln für Bezeichner sind die gleichen wie in C (allerdings andere Schlüsselwörter)
- ▶ Python besitzt eine **automatische Speicherverwaltung**
- ▶ Variablen müssen nicht deklariert werden
- ▶ Sie existieren, sobald ihnen ein Wert zugewiesen wird
- ▶ Sie sind stets lokal (und überdecken damit ggf. andere), können aber als global vereinbart werden

```
global var
```

- ▶ Python nutzt (wie z.B. auch C++) das Konzept von **Namensräumen** (*name spaces*), die eine feingranulare Nutzung auch gleicher Bezeichner erlauben

### 3.2.2 Anonyme Variablen

Zurück zu C

- ▶ **Anonyme Variablen** haben keinen Namen; es wird aber Speicherplatz reserviert
- ▶ Sie werden mit Hilfe von Funktionen der C-Standardbibliothek **zur Laufzeit** angelegt
- ▶ Anonyme Variablen werden **nicht** deklariert
- ▶ Nutzung folgender Funktionen, in `stdlib.h` deklariert
  - ▶ `void *malloc(size_t size)`  
Reserviert `size` Bytes im Speicher
  - ▶ `void *calloc(size_t count, size_t size)`  
Reserviert `count` × `size` Bytes im Speicher und initialisiert sie mit dem Wert 0
- ▶ Beide Funktionen geben eine **Adresse** der anonymen Variable zurück

## Python (Forts.)

Beispiel für lokale und globale Variablen in Python:

```
var1=42
var2=23
var3=4711

def func():
    global var1
    var1=1
    var2=2
    print("var1=",var1," var2=",var2," var3=",var3)
    return

print("var1=",var1," var2=",var2," var3=",var3)
func()
print("var1=",var1," var2=",var2," var3=",var3)
```

Die Ausführung dieses Codes führt zu:

```
var1= 42 var2= 23 var3= 4711
var1= 1 var2= 2 var3= 4711
var1= 1 var2= 23 var3= 4711
```

### Der sizeof-Operator

- ▶ **Frage:**  
Wieviel Platz soll denn bei der Reservierung angefordert werden?
- ▶ **Antwort:**  
Soviel, wie der Typ eines Wertes, der dort gespeichert werden soll, braucht.
- ▶ **Frage:**  
Ich will eine anonyme Variable vom Typ `int` in einem portierbaren Code. Die Größe von `int` ist nicht immer gleich. Woher weiß ich denn, wieviel Speicher `int` braucht?
- ▶ **Antwort:**  
Dafür gibt es den `sizeof`-Operator.

`sizeof`

„`sizeof`“ ist ein Schlüsselwort in C.

Der `sizeof`-Operator kann sowohl auf einen Typ als auch auf eine Variable angewendet werden.

Beispielsweise hat `sizeof(int)` auf diesem System (MacOSX auf Intel) den Wert 4.

## Verwaltung anonymer Variablen

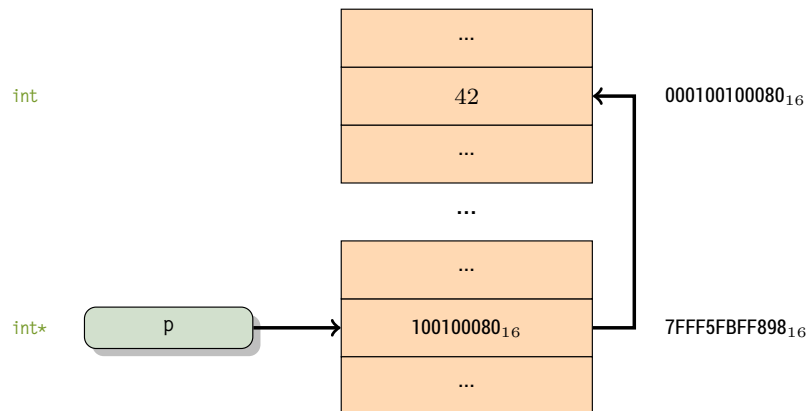
Die Verwaltung einer anonymen Variable obliegt dem Programm

- ▶ **Lebensdauer:** solange der Programmierer will → Freigabe zur gegebenen Zeit
- ▶ **Sichtbarkeit:** keine → der Programmierer muss dafür sorgen, dass die Existenz „gemerkt“ wird
- ▶ Freigabe über die Funktion `void free(void*)`, der die Adresse der anonymen Variablen übergeben wird

### Man beachte

Die Adresse einer anonymen Variable sollte **stets** in einer anderen (ggf. anonymen) Zeigervariable gespeichert werden!

## Beispiel (Forts.)



## Beispiel

```

/* malloc.c -- anonymous variables */
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int *p;

    p= malloc(sizeof(int)); /* reserves memory */
    *p=42; /* set value */
    printf("Pointer p has address %p and points to %p\n",
        (void*)&p,(void*)p);
    printf("Anonymous variable has the value %d\n",*p);
    free(p); /* releases memory */
    return 0;
}
    
```

```

> ./malloc
Pointer p has address 0x7fff5fbff898 and points to 0x100100080
Anonymous variable has the value 42
    
```

## Fehlerquellen

- ▶ Die Nutzung anonymer Variablen hat sich als eine der wesentlichsten Fehlerquellen in C erwiesen
- ▶ Daher zwei Warnungen:

### Warnung I

Das „**Vergessen**“ und eine in Folge nicht mehr mögliche Freigabe einer anonymen Variable ist eine häufige Fehlerquelle → **memory leak**.

### Warnung II

Die Nutzung einer **bereits freigegebenen** anonymen Variable ist ebenfalls eine häufige Fehlerquelle → **dangling pointer**

## Fehlerquellen (Forts.)

Zwei Regeln helfen, diese Fehler zu vermeiden:

1. Wenn Sie Code zur Reservierung (z.B. `malloc`) schreiben, schreiben sie **sofort** auch den entsprechenden Freigabecode
2. Wenn Sie eine anonyme Variable freigeben, weisen Sie ihrem Zeiger den (symbolischen) Wert `NULL` zu, der ebenfalls in „`stdlib.h`“ definiert ist
  - ▶ Es ist garantiert, dass nie eine Variable an der durch `NULL` beschriebenen Adresse liegt
  - ▶ Zur Laufzeit führt die Dereferenzierung von `NULL` zu einem Fehler
  - ▶ Falls Bibliotheksfunktionen wie `malloc` aus irgendeinem Grund scheitern, geben sie ebenfalls `NULL` zurück
  - ▶ Im Zweifel empfiehlt es sich daher, Zeiger vor dem Gebrauch gegen den Wert `NULL` zu testen

## Parameterübergabe

### Definition 3.2 (Formale und tatsächliche Parameter)

Die Parameter, die innerhalb eines Funktionsrumpfes verwendet werden, heißen **formale Parameter**.

Die Parameter, die beim Aufruf der Funktion verwendet werden, heißen **tatsächliche Parameter**<sup>6</sup> oder **Argumente**.

- ▶ Bei der Verwendung von Parametern in Programmiersprachen gibt es zwei Probleme zu klären:
  - ▶ Welche **Zuordnung** gibt es zwischen tatsächlichen und formalen Parametern?
  - ▶ Wie erfolgt die **Übergabe** zwischen tatsächlichen und formalen Parametern?

<sup>6</sup>Mitunter findet man in der (älteren) Literatur die Bezeichnung „aktueller Parameter“, die von der falschen Übersetzung des englischen Ausdrucks „*actual parameter*“ stammt.

## 3.2.3 Parameter

- ▶ Neben den benannten – zur Übersetzungszeit reservierten – und den anonymen – zur Laufzeit reservierten – Variablen gibt es noch eine dritte Sorte von Variablen, die wir schon ständig benutzt haben  
→ **Parameter**
- ▶ Parameter sind Variablen, die automatisch beim Aufruf einer Funktion **angelegt** und **initialisiert** werden
  - ▶ **Lebendauer**: Verweilzeit des Programms in der Funktions**instanz**
  - ▶ **Sichtbarkeit**: kompletter Funktionskörper (falls nicht überdeckt)
- ▶ Parametervariablen werden mit der Funktionsdefinition deklariert und definiert
- ▶ Sie dürfen keine **Speicherklasse**, aber **Speicherattribute** haben

## Parameterübergabe (Forts.)

Es gibt in Programmiersprachen verschiedene Varianten der Zuordnung von tatsächlichen und formalen Parametern

- ▶ **Positionsparameter**:  
Die Zuordnung ergibt sich aus der Position eines tatsächlichen Parameters bei Funktionsaufruf
- ▶ **Namensparameter**:  
Die Zuordnung ergibt sich aus beim Funktionsaufruf benutzten Namen

**C**

C benutzt **ausschließlich** Positionsparameter.

**Python**

In Python können sowohl Positions- als auch Namensparameter benutzt werden.

## Parameterübergabe (Forts.)

Es gibt in Programmiersprachen verschiedene Varianten der **Übergabe** von tatsächlichen zu formalen Parametern:

- ▶ **Call by value**: Der tatsächliche Parameter wird ausgewertet und der formale Parameter erhält diesen Wert → **Wertübergabe**
- ▶ **Call by name**: Der formale Parameter wird durch den Namen des tatsächlichen Parameters ersetzt → **Namensersetzung**
- ▶ **Call by reference**: Der formale Parameter wird zu einem „Stellvertreterobjekt“ für den tatsächlichen Parameter, so dass alle Änderungen sofort außerhalb der Funktion wirksam werden → **Referenzübergabe**
- ▶ **Call by copy/return**<sup>7</sup>: Der tatsächliche Parameter wird ausgewertet und der formale Parameter erhält diesen Wert zu Beginn der Funktion; bei Beendigung erhält der tatsächliche Parameter den Wert des formalen Parameters

<sup>7</sup>In der Literatur auch: „call by value/return“, „call by value and result“ oder „copy in/copy out“

## Ersatz von Referenzparametern

- ▶ C kennt **keine** Referenzparameter
- ➔ Konzept von **Prozeduren** (wie z.B. `var`-Parameter in Pascal) **nicht** (ohne Weiteres) realisierbar
- ▶ **Idee**: Statt eines Referenzparameters wird ein **Zeiger auf das Argument** übergeben
  - ▶ Durch Dereferenzierung können Änderungen von Daten **außerhalb** der Funktion erreicht werden
  - ➔ Wirkung entspricht (im Wesentlichen) Referenzparametern
  - ▶ Die Nutzung von Zeigern statt Referenzvariablen ist eine **häufige Praxis** in C

### Definition 3.3 (Seiteneffekt)

Eine Speicheränderung außerhalb der lokalen Variablen einer Funktion (zu der auch die Parameter zählen) oder eine Ein-/Ausgabe nennt man einen **Seiteneffekt**.

- ▶ Dieser Begriff „Seiteneffekt“ stammt von der inkorrekten Übersetzung des Begriffes „side effect“

## Parameterübergabe (Forts.)

### C

In C gibt es **ausschließlich** eine Form der Parameterübergabe: „Call by value“ (Wertübergabe).

### Python

Python kennt (eigentlich) ausschließlich Referenzparameter<sup>8</sup> (call by reference). Jedoch unterscheidet Python zwischen „**unveränderlichen**“ (*immutable*) Typen (wie z.B. Ganzzahlen oder Zeichenketten) und „**veränderlichen**“ (*mutable*) Typen. Erstere werden bei der Funktionsübergabe wie Werteparameter behandelt.

<sup>8</sup>Die Wirklichkeit ist etwas komplizierter, siehe Exkurs etwas später in diesem Abschnitt.

## Ersatz von Referenzparametern (Forts.)

```

/* sideeffect.c -- simulated reference parameter */
#include<stdio.h>

void sideeffect(int *);

int main()
{
    int x=42;
    printf("x=%d\n",x);
    sideeffect(&x);
    printf("x=%d\n",x);
    return 0;
}

void sideeffect(int *p)
{
    *p=23;
}
    
```

x=42  
x=23

## scanf()

- ▶ Ein Einsatzfall, bei dem man Referenzparameter braucht, ist die Eingabe von mehr als einem Wert
- ▶ Die Funktion `scanf()` aus der Standardbibliothek übernimmt genau diese Funktion
- ▶ Ähnlich wie `printf()` beginnen die Parameter mit einem Formatstring, gefolgt von einer Liste von Zeigern auf die Variablen
- ▶ Rückgabewert ist die Anzahl der korrekt gelesenen Variablen

### Formatierung

Welchen Typ ein einzulesender Wert ist, wird durch ein Ausdruck bestimmt, der mit einem Prozentzeichen beginnt, u.a.:

|                 |                         |                 |                                    |
|-----------------|-------------------------|-----------------|------------------------------------|
| <code>%d</code> | ganzahliger Dezimalwert | <code>%i</code> | Ganzahl, Basis hängt vom Präfix ab |
| <code>%f</code> | Gleitkommazahl          | <code>%s</code> | Zeichenkette                       |

Die %-Ausdrücke können auf verschiedene Arten modifiziert werden. Eine Übersicht gibt Anhang ?? des Skripts.

## Exkurs: Variablen und Referenzen in Python

- ▶ Die nutzersichtbaren Variablen in Python sind eigentlich **Referenzen** auf **anonyme** Variablen (Objekte), die wiederum vom Python-Laufzeitsystem verwaltet werden
- ▶ Es gibt eine zum `&`-Operator ähnliche Funktion, die die Identität eines Objektes zurückgibt: `id()`
  - ▶ In manchen Implementationen (z.B. CPython) ist es tatsächlich die Speicheradresse
  - ▶ Wird `id()` auf eine (Nutzer-)Variable (also Referenz) angewendet, gibt es die Identität des referenzierten Objekts, nicht die der Referenz

## scanf() (Forts.)

```

/* scanf.c -- input via scanf */

int scanf(char*,...);
int printf(char*, ...);

int main(){
    int x,y;
    printf("Give the point as \"(x,y)\": ");
    if(scanf("%d,%d",&x,&y) == 2){
        printf("You provided: (%d,%d).\n",x,y);
    }
    return 0;
}

```

```

./scanf
Give the point as "(x,y)": (10,4)
You provided: (10,4).

```

## Exkurs: Variablen und Referenzen in Python (Forts.)

- ▶ Man betrachte folgenden Code:

```

a=23
print("a=",a," id(a)=",id(a))
b=42
print("b=",b," id(b)=",id(b))
a=a+19
print("a=",a," id(a)=",id(a))

```

Bei der Ausführung erhält man:

```

a= 23 , id(a)= 4298184952
b= 42 , id(b)= 4298186472
a= 42 , id(a)= 4298186472

```

← Identisch!



## Exkurs: Variablen und Referenzen in Python (Forts.)

- ▶ Bei **unveränderlichen** (*immutable*) Objekten (z.B. Ganzzahlen) wird bei einer Variablenzuweisung (z.B. `a=a+19`) **nicht** das Objekt geändert, sondern es wird entweder
  - ▶ ein **neues Objekt** erzeugt und referenziert  
oder
  - ▶ falls ein derartiges Objekt **bereits existiert**, dieses ggf. referenziert
- ▶ Im letzten Beispiel gab es schon ein Objekt, das der Ganzzahl 42 entsprach und von `b` referenziert wurde
- ▶ Einige komplexere Objekte sind **veränderlich** (*mutable*)
  - ▶ Bei Operationen mit ihnen werden sie verändert und kein neues Objekt angelegt
  - ▶ Die Referenz bleibt erhalten

## 3.3 Abgeleitete Typen

- ▶ C kennt sogenannte **abgeleitete Typen**
- ▶ Eine Art von abgeleiteten Typen haben wir schon kennengelernt: Zeiger
- ▶ Andere Typen sind:
  - ▶ Verbund-Datentypen (`struct` und `union`)
  - ▶ Aufzählungstypen (`enum`)
  - ▶ Felddatentypen (`arrays`)
- ▶ Abgeleitete Typen werden manchmal auch „komplexe Typen“ genannt

## Exkurs: Variablen und Referenzen in Python (Forts.)

- ▶ Beim Parameterruf wird die Referenz des Objekts übergeben
  - ▶ *De facto*: Kopie der bei Aufruf verwendeten Referenz
- ▶ Bei unveränderlichen Objekten führen Berechnungen in der Funktion zu Referenzierung **anderer** Objekte
- ▶ Außerhalb wird noch das gleiche Objekt referenziert → Wirkung ist wie bei **Werteübergabe**
- ▶ Bei veränderlichen Objekten wird keine neue Referenz gebildet → **Referenzübergabe** bleibt sichtbar

## Verbundtyp struct

- ▶ Mit Hilfe des `struct`-Typs können mehrere Variablen (Elemente, *members*) unterschiedlichen Typs im Zusammenhang behandelt werden
- ▶ Eine Deklaration besteht aus dem Schlüsselwort `struct`, einem **Namen** (Tag), einer Liste von **Deklarationen** von Elementen bereits bekannter Typen in einem Block aus geschweiften Klammern

```
struct point{
    int x;
    int y;
};
```

- ▶ Für Pascal-Kenner: Der `struct`-Typ entspricht etwa dem `record`
- ▶ Mit der Deklaration ist der **Typ** bekannt – mit ihm können auf die übliche Weise Variablen deklariert/definiert werden:

```
struct point pt;
```

## Verbundtyp struct (Forts.)

- ▶ Typen- und Variablendeklaration kann auch zusammen geschehen:

```

struct point{
    int x;
    int y;
} pt;
    
```

- ▶ In diesem Fall kann der Tag (Name) entfallen

### Vorsicht

Wenn der Tag weggelassen wird...

- ▶ ... kann keine weitere Variable dieses Typs deklariert werden
- ▶ ... ist selbst ein strukturell identischer Typ **nicht** kompatibel

## Verbundtyp struct (Forts.)

- ▶ Eine `struct`-Variable kann mit einer Liste von konstanten Ausdrücken initialisiert werden, die in geschweiften Klammern steht

```

/* struct2.c -- initialization of a struct */
#include<stdio.h>

struct point{
    int x;
    int y;
} pt = { 42, 25-23};

int main()
{
    printf("member of pt: x=%d, y=%d\n",
        pt.x,pt.y);
    return 0;
}
    
```

## Verbundtyp struct (Forts.)

- ▶ Auf ein Element eines `struct` kann mit dem „.“-Operator zugegriffen werden
- ▶ Form: `variable_name.member_name`
- ▶ Beispiel:

```

/* struct.c -- access to members */
#include<stdio.h>

struct point{
    int x;
    int y;
} pt;

int main()
{
    pt.x=42;
    pt.y=23;
    printf("member of pt: x=%d, y=%d\n",
        pt.x,pt.y);
    return 0;
}
    
```

## Verbundtyp struct (Forts.)

- ▶ Elemente können auch über ihren Namen (mit vorgesetztem Punktoperator) angesprochen werden, so dass eine nur teilweise Initialisierung oder eine andere Reihenfolge der Elemente möglich ist:

```

/* struct2a.c -- C99 initialization */
#include<stdio.h>
#include<stdlib.h>

struct point{
    int x;
    int y;
};

int main()
{
    struct point pt = { .y=23, .x=42 };

    printf("member of pt: x=%d, y=%d\n",
        pt.x,pt.y);
    return 0;
}
    
```

## Verbundtyp struct (Forts.)

- ▶ Für eine `struct`-Variable sind nur wenige Operationen erlaubt:
  - ▶ Sie darf zugewiesen/kopiert werden („`=`“-Operator)
  - ▶ Ihre Adresse darf ermittelt werden („`&`“-Operator)
  - ▶ Auf ihre Elemente darf zugegriffen werden („`.`“-Operator)

### Zeiger auf struct

In C kommt es häufig vor, dass Zeiger auf (in der Regel anonyme) `struct`-Variablen benutzt werden.

**Diese Konstruktion ist die Basis fast aller abstrakter Datentypen (ADT) in C, die im nächsten Semester genauer betrachtet werden.**

## Verbundtyp struct (Forts.)

- ▶ Für das Konstrukt `(*pointer).member` gibt es einen Extraoperator: „`->`“
- ▶ Der folgende Code ist also äquivalent zu `struct3.c`:

```

/* struct4.c -- -> operator */
#include<stdio.h>
#include<stdlib.h>

struct point{
    int x;
    int y;
};

int main()
{
    struct point *ppt = malloc(sizeof(struct point));
    ppt->x=42;
    ppt->y=23;
    printf("member of pt: x=%d, y=%d\n",
        ppt->x,ppt->y);
    return 0;
}
    
```

## Verbundtyp struct (Forts.)

```

/* struct3.c -- anonymous struct */
#include<stdio.h>
#include<stdlib.h>

struct point{
    int x;
    int y;
};

int main()
{
    struct point *ppt = malloc(sizeof(struct point));
    (*ppt).x=42;
    (*ppt).y=23;
    printf("member of pt: x=%d, y=%d\n",
        (*ppt).x,(*ppt).y);
    return 0;
}
    
```

## Typenalias mit typedef

- ▶ Die ständige Nutzung des Schlüsselworts `struct` kann umgangen werden, indem ein Typenalias definiert wird
- ▶ Dazu gibt es das Schlüsselwort `typedef`
- ▶ Allgemeine Form: `typedef type alias_name;`

```

/* typedef.c -- define alias types */
#include<stdio.h>
#include<stdlib.h>

typedef unsigned short int usint_t;
typedef struct {
    usint_t x;
    usint_t y;
} point_t;

int main()
{
    point_t pt={42,23};

    printf("member of pt: x=%d, y=%d\n", pt.x,pt.y);
    return 0;
}
    
```

## Verbundtyp union

- ▶ Der Verbundtyp `union` ist ähnlich zum Typ `struct`
- ▶ **Unterschied: Alle Elemente liegen an der gleichen(!) Adresse**
  - ➔ Das Schreiben eines Elements zerstört den Inhalt der anderen
- ▶ Wird nur selten gebraucht (typischerweise zur Optimierung oder für hardwarenahe Programmierung) und wird hier nicht weiter betrachtet

## Aufzählungstyp (Forts.)

- ▶ `enum`-Werte werden zur **Übersetzungszeit** festgelegt und bilden damit eine Möglichkeit, **Konstanten** zu definieren
  - ▶ Dies geht aber **nur** für Ganzzahl-Konstanten
- ▶ Im Gegensatz zu `const` (siehe später in diesem Abschnitt) benötigt ein „ungebrauchtes“ `enum` keinen Speicherplatz zur Laufzeit
- ▶ **Achtung:** Da `enums` im Wesentlichen wie `ints` behandelt werden, bieten sie nur eine eingeschränkte Typensicherheit

## Aufzählungstyp

- ▶ Mit Hilfe von **Aufzählungstypen** können Typen mit diskreten benannten Werten definiert werden
- ▶ Schlüsselwort: `enum` (für *enumeration*)
- ▶ Beispiel:

```
enum card_suit {club, spade, heart, diamond};
```

- ▶ Für alle praktischen Belange ist ein Aufzählungstyp ein `int`
- ▶ Man kann gezielt Werte zuweisen:

```
enum month {jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
```

```
enum escape {NL='\n', BACKSPACE='\b', TAB='\t', RETURN='\r', BELL='\a'};
```

## Arrays

- ▶ Häufig werden zusammengehörige gleichartige Werte benötigt
- ▶ In der Mathematik mit Index:  $x_1, x_2, \dots$
- ▶ Beispiele:
  - ▶ Komponenten eines Vektors (Index: Dimension)
  - ▶ Die jeweiligen Tagesumsätze eines Monats (Index: Tag)
  - ▶ Die Pixel eines Bildschirms (Indizes: x-Achse, y-Achse)
  - ▶ ...
- ▶ C stellt dafür den abgeleiteten **Arraytyp** (häufig auch: **Feldtyp**) zur Verfügung
- ▶ Deklaration benötigt kein Schlüsselwort, sondern geschieht mit Hilfe des **Indexoperators** (eckige Klammern), der die Elementzahl enthält und hinter den Variablenbezeichner<sup>9</sup> geschrieben wird
- ▶ Deklaration eines Arrays von 10 Ganzzahlen:

```
int xa[10];
```

<sup>9</sup>Für Java-Programmierer: Achtung, nicht dem Typ!

## Arrays (Forts.)

- ▶ Auch der Zugriff auf Element eines Array erfolgt über den Indexoperator

```
int x1=xa[0];
```

### Achtung!

Eine Array mit  $n$  Elementen hat **immer** die Indizes 0 bis  $n - 1$ .

Der Zugriff auf das Element  $n$  ist ein **Fehler!**

- ▶ Der Grundtyp eines Arrays können beliebige andere Typen sein
- ▶ Betrachte folgende Deklarationen:

```
int xa[10]; /* Array of 10 integers */
int *pxa[10]; /* Array of 10 pointers to integers */
double f[10]; /* Array of 10 floating point numbers */
struct point{
    int x;
    int y;
} pt[10]; /* Array of 10 point structs */
```

## Variable Arrays

- ▶ Seit C90 können lokale Arrays angelegt werden, deren Größe erst zur Laufzeit ermittelt wird (*variable length array = VLA*)

```
int func(int s){
    double a[s];
    // ...
}
```

- ▶ Jedoch wird diese Möglichkeit in C11 nur noch als optional zugelassen, so dass man sich nicht unbedingt darauf verlassen sollte, wenn Portabilität angestrebt wird
- ▶ VLA haben einige Einschränkungen:
  - ▶ Sie können nicht statisch (Schlüsselwort `static`) sein
  - ▶ Der Steuerfluss darf niemals **hinter** die Deklaration eines VLA in dessen Gültigkeitsbereich gelangen
  - ▶ VLAs können nicht Teil einer `struct` sein

## Mehrdimensionale Arrays

- ▶ Auch mehrdimensionale Arrays sind möglich
- ▶ Konzept: Array eines anderen Array-Typs:

```
int xa[4][3]; /*declares a 4x3 array */
```

- ▶ Das Beispiel deklariert ein  $4 \times 3$ -Array
- ▶ Genauer: Es wird ein Array mit 4 Elementen deklariert, wobei jedes Element ein Array mit 3 Elementen ist
- ▶ Dies Prinzip lässt sich auf beliebig viele Dimensionen ausweiten

```
int xa[4][3][2]; /*declares a 4x3x2 array */
```

### Achtung

Eine Separierung der Indizes durch Komma oder Semikolon (z.B. wie in Pascal) funktioniert nicht!

## Initialisierung

- ▶ **Initialisierung:** wie `struct`-Typen können Arrays über über Listen nicht-variabler Ausdrücke in geschweiften Klammern initialisiert werden:

```
int xa[4]={0,1,2,3}; /*initialize elements with their indices */
```

- ▶ Dies funktioniert auch bei mehrdimensionalen Arrays:

```
int xa[3][4]={
    {0,1,2,3},
    {1,2,3,4},
    {2,3,4,5}}; /* initialize elements with sum of their indices */
```

- ▶ Wenn ein Array bei der Definition initialisiert wird, kann die letzte Dimension weggelassen werden
- ▶ Das folgende Beispiel ist äquivalent zum ersten:

```
int xa[]={0,1,2,3}; /*initialize elements with their indices */
```

## Initialisierung (Forts.)

- ▶ Seit C90 ist auch eine teilweise Initialisierung möglich
- ▶ Die Reihenfolge ist dabei egal

```
int a[10] = { [3]=42, [2]=23 };
```

- ▶ Alle nicht explizit initialisierte Elemente sind dann 0
- ▶ Beide Initialisierungsarten lassen sich mischen
- ▶ Wie bei `enum` wird die indexlose Initialisierung für das nächst Element fortgesetzt

```
int a[8] = { [3]=42, 4, 5, [0]=23 };
```

- ▶ ...ergibt: 23, 0, 0, 42, 4, 5, 0, 0

## Array als Parameter (Forts.)

- ▶ Zum Glück funktioniert der Indexoperator (eckige Klammern) auch mit Zeigern
- ▶ Deshalb kann (ähnlich wie bei der Initialisierung) die Größe (der letzten Dimension) eines Arrays bei der Übergabe weggelassen werden, oder gleich ein Zeiger übergeben werden
- ▶ Die folgenden Deklarationen sind äquivalent:

```
void func(double[]); // parameter is array of double
void func(double*); // parameter is pointer to double
```

- ▶ **Problem:** Es besteht keine Möglichkeit herauszufinden, wie groß das Array vor der Übergabe war
- ▶ Deshalb benötigen viele Funktionen, die Array als Parameter haben, als zusätzlichen Parameter die Anzahl der Elemente

## Array als Parameter

- ▶ Arrays sind gegenüber anderen Typen benachteiligt: Sie lassen sich **nicht** an Funktionen übergeben
- ▶ **Genauer:** Man kann zwar Parameter als Array deklarieren, in Wirklichkeit wird aber ein **Zeiger auf den Grundtyp** des Arrays übertragen!
- ▶ Man sagt: Das Array „zerfällt“

```
/* array2.c -- an array decays */
#include<stdio.h>

typedef int myarray[10];

void func(myarray a){
    printf("Size of a: %ld\n",sizeof(a));
}

int main() {
    myarray x;
    printf("Size of x: %ld\n",sizeof(x));
    func(x);
    return 0;
}
```

```
> ./array2
Size of x: 40
Size of a: 8
```

## Array als Parameter (Forts.)

```
/* array_param.c -- auxiliary parameter */
#include<stdio.h>

void print_int_array(int,int[]);

int main() {
    int xa[3]={1,2,3};
    print_int_array(3,xa);
    return 0;
}

void print_int_array(int count,int array[]) {
    int i=1;
    printf("%d",array[0]);
    while(i<count) {
        printf(",%d",array[i]);
        i=i+1;
    }
    printf("\n");
}
```

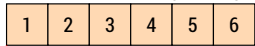
```
> ./array_param.c
(1,2,3)
>
```



- ▶ **Achtung!** Falsche Annahmen über Array-Größen sind ein häufiger Programmierfehler

## Speicherlayout

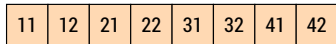
- ▶ Warum können wir ein Array als Pointer „auffassen“? → festgelegtes Speicherlayout
- ▶ Die Elemente eines Arrays liegen hintereinander im Speicher: `int x[6]={1,2,3,4,5,6};`



Adresse von x

- ▶ Dies gilt auch für mehrdimensionale Arrays:

`int x[4][2] = {{11,12},{21,22},{31,32},{41,42}};`



Adresse von x

- ▶ Genau genommen ist `a[i]` nur eine andere Schreibweise für `*(&a+i)`
  - ▶ **Anmerkung:** Bei der Addition zwischen einem Zeiger und einem Integer wird der Integer immer mit der Größe des Typs, auf den der Zeiger zeigt, multipliziert

## Stringfunktionen

- ▶ Die Null am Ende ist eine **Konvention**
- ▶ Viele Funktionen der Standardbibliothek verlassen sich darauf, dass `char`-Array sogenannte **C-Strings** sind
- ▶ Wir haben schon zwei kennengelernt:
  - ▶ `int printf(char*,...)`
  - ▶ `int atoi(char*)`
- ▶ Eine Konvention kann man brechen → **Gefahr**

```

/* printf-failure -- danger of C strings */
#include<stdio.h>

char str[]={ 'H', 'a', 'l', 'l', 'o', '\0' };
char bla[]="Das ist top secret.\n";

int main(){
    printf(str);
    return 0;
}
    
```

```

> ./print-failure
Hallo!
Das ist top secret.
    
```

## Spezialfall Zeichenarray

- ▶ Wie schon erwähnt, kennt C keine Zeichenketten
- ▶ Daher werden `char`-Arrays als Zeichenketten missbraucht
- ▶ Dafür kennt C einen Spezialfall der Initialisierung

```
char str[]="Hallo!";
```

ist äquivalent zu

```
char str[]={ 'H', 'a', 'l', 'l', 'o', '\0' };
```

angehängter Wert "0"

- ▶ Der Wert 0 (oder `'\0'`) wird als End-Marke genutzt
  - Er darf in keiner Zeichenkette enthalten sein
- ▶ **Genauer:** `"Hallo!"` ist ein **Stringliteral**, der in der internen Darstellung mit einer 0 endet

## Parameter von `main()`

- ▶ Nun haben wir alles zusammen, die Parameter der `main()`-Funktion zu betrachten
- ▶ Ihr werden die Befehlszeilenparameter als C-String übergeben
- ▶ Die korrekte Signatur von `main()` ist<sup>10</sup>:

```
int main(int argc, char *args[]);
```

- ▶ Rückgabetype `int`: wird an aufrufendes Programm oder Betriebssystem gegeben, Konvention ein Fehlercode
- ▶ `int argc`: Anzahl der Parameter des Programms
  - ▶ Da immer der Aufrufname des Programms mit übergeben wird, ist `argc` mindestens 1
- ▶ `char *args[]`: Array von Zeigern auf C-Strings
  - ▶ Jeder dieser C-Strings enthält ein Kommandozeilenargument (inklusive Programmname)

<sup>10</sup>Die eigentliche Deklaration ist noch etwas komplizierter.

## Parameter von main() (Forts.)

```

/* main.c -- gives command line arguments */
#include<stdio.h>

int main(int argc, char *args[])
{
    printf("Program name: %s\n",args[0]);
    if (argc>1) { printf("1. argument: %s\n",args[1]); }
    if (argc>2) { printf("2. argument: %s\n",args[2]); }
    if (argc>3) { printf("3. argument: %s\n",args[3]); }
    if (argc>4) { printf("4. argument: %s\n",args[4]); }
    if (argc>5) { printf("5. argument: %s\n",args[5]); }
    if (argc>6) { printf("There are even more arguments.\n"); }
    return 0;
}
    
```

```

> ./main 1 alpha "Ein kurzer Satz."
Program name:./main
1. argument: 1
2. argument: alpha
3. argument: Ein kurzer Satz.
    
```

## Nochmals Speicherklassen und -attribute

- ▶ Nachdem wir alle C-Typen beisammen haben, betrachten wir nochmals die Speicherklassen und -attribute
- ▶ Schon betrachtet: **Speicherklassen** „static“ und „extern“<sup>11</sup>
  - static: Macht Lebensdauer unabhängig von Stand der Abarbeitung; schränkt Sichtbarkeit nach „außen“ ein
  - extern: Nur Deklaration, verlangt zusätzliche Definition, evtl. „außerhalb“
    - „Außerhalb“ heißt Bibliothek oder Modul, wird bei der Modularisierung genauer betrachtet
- ▶ Speicherklasse **register**: Compiler soll den Zugriff auf Variable beschleunigen
  - ▶ Nur „Hinweis“, Compiler entscheidet
  - ▶ Bei CPUs mit Universalregistern wird Variable in einem Register gehalten → daher der Name
  - ▶ Programmierer darf von **register**-Variable **keine** Adresse ermitteln
  - ▶ Einsatz bei **zeitkritischen** Funktionen, z.B. Treibern

<sup>11</sup>... und „auto“ 😊

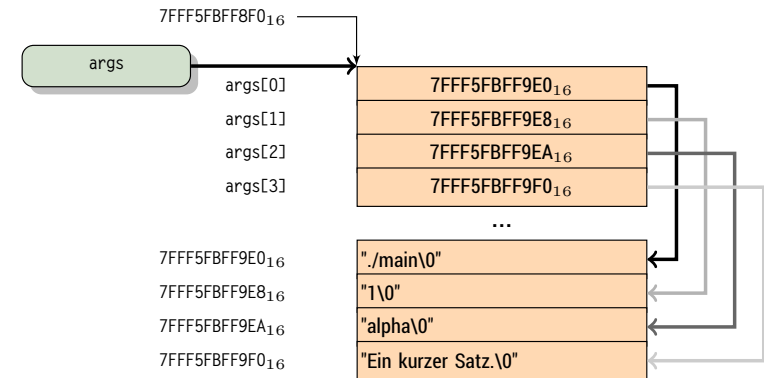
## Parameter von main() (Forts.)

Die Deklaration...

```
char *args[]
```

... bedeutet Array von Zeigern auf char.

Beim obigen Aufruf ergibt sich (beispielsweise) folgendes Speicherlayout:



## Nochmals Speicherklassen und -attribute (Forts.)

```

/* register.c -- program does not compile */
#include<stdio.h>

int main()
{
    register int fast=42;
    int pf;

    pf = &fast;
    printf("fast= %d\n", fast);
    return 0;
}
    
```

```

cc -std=c99 -pedantic -Wall register.c -o register
register.c: In function 'main':
register.c:9: error: address of register variable 'fast' requested
make: *** [register] Error 1
    
```



## Nochmals Speicherklassen und -attribute (Forts.)

- ▶ Speicherattribut „**const**“: Wert der Variablen soll nicht verändert werden
  - ➔ Variable bleibt **konstant**
    - ▶ Ermöglicht Compiler Optimierung
    - ▶ Compiler **unterbindet Zuweisung** an **const**-Variable, außer bei Initialisierung
    - ▶ Variable wird jedoch sonst wie jede andere Variable behandelt, eignet sich insbesondere **nicht** für Angaben von Array-Größen oder Initialisierung von Verbundtypen oder Arrays
      - ▶ **Aber**: Seit C99 sind dort Variablen erlaubt

### Achtung!

Insbesondere bei Zeigern ist die Deklaration z.T. verwirrend. Man beachte den Unterschied:

`const int * p;` Deklariert einen **variablen** Zeiger auf eine **konstante** Ganzzahlvariable

`int * const p;` Deklariert einen **konstanten** Zeiger auf eine **variable** Ganzzahlvariable

## Nochmals Speicherklassen und -attribute (Forts.)

- ▶ Speicherattribut „**restrict**“ ist neu in C99
- ▶ Nur bei Zeigervariablen (und daraus abgeleitet bei Arrays)
- ▶ Zusicherung an den Compiler, dass das Speicherobjekt, auf das der Zeiger zeigt, nur (direkt oder indirekt) über diesen Zeiger zugegriffen wird
- ▶ Konkrete Definition ist etwas umfangreicher ➔ Interessierte lesen bitte §6.7.3.1 der ISO/IEC 9899
- ▶ Dient zur Programmoptimierung durch den Compiler

## Nochmals Speicherklassen und -attribute (Forts.)

- ▶ Das Speicherattribut „**volatile**“ ist das Gegenteil von „**const**“
- ▶ „**volatile**“ ➔ (engl.) flüchtig, sprunghaft
- ▶ **Hinweis** an Compiler: Wert kann sich seit dem letzten Schreiben **verändert** haben
- ▶ Nur bei **nebenläufigen** Programmen relevant

### Hinweis

Eine Deklaration wie

```
extern const volatile int rt_clock;
```

ist durchaus sinnvoll. Der Wert wird von „außen“ (z.B. durch einen Treiber) gesetzt und kann vom aktuellen Programm nur gelesen, aber nicht geändert werden.

## Komplexe Typen in Python

- ▶ Es gibt in Python eine Reihe von vordefinierten Typen, die ähnliche (und weitere) Fähigkeiten wie die besprochenen abgeleiteten C-Typen haben

### Merke:

Da „Ableitung“ in der Objektorientierung etwas anderes bedeutet und die hier besprochenen Typen auch keine Ableitungsbeziehung im Sinne von C haben, wird nur der Begriff „komplexe Typen“ verwendet.

- ▶ Betrachten (kurz):
  - ▶ Strings (Zeichenketten)
  - ▶ Listen
  - ▶ Tupels
  - ▶ Mengen
  - ▶ Dictionaries

## Python Strings

- ▶ **Strings** sind eine Zeichen-Sequenz fester Länge
- ▶ Genau genommen **kein** komplexer Typ
- ▶ String-Literale werden in einfache (') oder doppelte (") Anführungszeichen gesetzt<sup>12</sup>
- ▶ Ausnahmezeichen sind die gleichen wie in C-Literalen
  - ▶ Beginnt ein Literal mit einem **r** oder **R**, so werden Ausnahmezeichen nicht interpretiert → **rohe Strings** (*raw strings*)
- ▶ Strings sind **unveränderlich** → String-Operationen erzeugen also neue Strings (Auch wenn man es als Programmierer nicht unbedingt merkt)

<sup>12</sup>Es gibt für Spezialzwecke noch dreifache Quotes.

## Python Listen (Forts.)

```
# list1.py -- list examples
L=['John','Paul','Georg','Pete'] # list of string
print("list:",L)
print("first element:",L[0])
print("next to last element:",L[-2])
L[-1]='Ringo'
print("list:",L)
L.append('Brian')
print("list:",L)
```

```
list: ['John', 'Paul', 'Georg', 'Pete']
first element: John
next to last element: Georg
list: ['John', 'Paul', 'Georg', 'Ringo']
list: ['John', 'Paul', 'Georg', 'Ringo', 'Brian']
```

## Python Listen

- ▶ Listen sind das, was den C-Arrays am nächsten kommt → eine geordnete Sammlung anderer „Objekte“
  - ▶ **Unterschied:** Elemente in der Liste können **unterschiedliche** Typen besitzen
- ▶ Die Elemente einer Liste werden in in eckige Klammern geschrieben und durch Kommata getrennt
- ▶ **Indizierung:**
  - ▶ Klammeroperator wie in C, beginnend mit 0
  - ▶ Negative Indizes zählen von rechts (beginnend mit -1)

```
L=[] # empty list
L=[1,2,3,4] # list of numbers
L=['John','Paul','Georg','Ringo'] # list of string
L=[42,'a',3.14] # list of different types
```

- ▶ Listen sind veränderlich, wie der folgende Code demonstriert

## Python Tupels

- ▶ **Tupels** sind sehr ähnlich zu Listen
  - ▶ Unterschied: Tupels sind **unveränderlich**
- ▶ **Schreibweise:** Statt eckigen werden runde Klammern verwendet
  - ▶ Sonderfall: Einertupels werden mit abschließenden Komma geschrieben, um sie von „normalen“ Klammerausdrücken unterscheiden zu können

```
T1=() # empty tuple
T2=(0,) # one element
T3=(1,'a',[42,'z']) # mixed types
```

- ▶ Da Tupels unveränderlich sind, gibt es keine `append()` Funktion
- ▶ Allerdings können Tupels verkettet werden → ein neuer Tupel entsteht

## Python Tupels (Forts.)

```
# tuple2.py -- concatenation
T=(1,2)
print("tuple:",T," with id:",id(T))
T=T+(3,4)
print("tuple:",T," with id:",id(T))
```

```
tuple: (1, 2) with id: 4299594352
tuple: (1, 2, 3, 4) with id: 4299405632
```

## Mengen in Python (Forts.)

- ▶ Für Mengen sind verschiedene aus der Mengenlehre bekannte Operationen definiert, u.a.:
  - ▶ `len(s)`: gibt Mächtigkeit der Menge `s`
  - ▶ `s1 | s2`: Vereinigungsmenge von `s1` und `s2`
  - ▶ `s1 & s2`: Schnittmenge von `s1` und `s2`
  - ▶ `s1 - s2`: Differenzmenge von `s1` und `s2`
  - ▶ `s1 ^ s2`: Symmetrische Differenzmenge von `s1` und `s2`

```
s1=set("Python")
s2=set("Monty")
print("s1|s2=",s1|s2)
print("s1&s2=",s1&s2)
print("s1-s2=",s1-s2)
print("s1^s2=",s1^s2)
```

```
s1|s2= set(['h', 'M', 'o', 'n', 'P', 't', 'y'])
s1&s2= set(['y', 't', 'o', 'n'])
s1-s2= set(['h', 'P'])
s1^s2= set(['h', 'M', 'P'])
```

## Mengen in Python

- ▶ Seit Python 2.6 gibt es **Mengentypen**
- ▶ Zwei Ausprägungen:
  - ▶ `set` → veränderlich
  - ▶ `frozenset` → unveränderlich, darf auch nur unveränderliche Elemente besitzen
- ▶ **Schreibweise:** Nach dem Wort<sup>13</sup> `set` oder `frozenset` folgt im Klammern eine Sequenz/Menge
- ▶ Jedes Element wird der Menge nur **einmal** eingefügt
- ▶ Es wird keine Ordnung bewahrt

```
# set.py -- set types
s1 = set([1, 2, -9, 0, 10, 2])
s2 = frozenset("Python")
print("s1=",s1)
print("s2=",s2)
```

```
s1= set([0, 1, 2, 10, -9])
s2= frozenset(['h', 'o', 'n', 'P', 't', 'y'])
```

## Python Dictionaries

- ▶ **Dictionaries** sind (ungeordnete) Mengen von Zweier-Tupeln.
  - ▶ Ungeordnet → nicht über Index zugreifbar
- ▶ Das erste Element dient als **Schlüssel** (*key*), über die ein Tupel auffindbar ist
- ▶ Dictionaries sind **veränderlich**
- ▶ Die Nutzung ist aus folgendem Beispiel ersichtlich

```
# dictionary.py -- demonstrate dictionaries
mlength = {'jan': 31, 'feb': 28,
           'mar': 31, 'apr': 30,
           'may': 31, 'jun': 30,
           'jul': 31, 'aug': 31,
           'sep': 30, 'oct': 31,
           'nov': 30, 'dec': 31}

print("March has", mlength['mar'], "days")
print("February has", mlength['feb'], "days")

mlength['feb'] = 29 # leap year
print("February has", mlength['feb'], "days")
```

## Aufgaben

### Aufgabe 3.1

Was ist der Unterschied zwischen den C-Ausdrücken `int (*x)[10];` und `int *x[10];`?  
Zeichnen Sie eine Skizze zum jeweiligen Speicherlayout!

### Aufgabe 3.2

Überlegen Sie, wie Sie die Funktionalität einer auf maximal 10 Elemente beschränkten Python-Liste mit einem C-Array realisieren können.

Berücksichtigen Sie, dass sich in der Liste Elemente unterschiedlicher Typen befinden können!

## Aufgaben (Forts.)

### Aufgabe 3.3

Für eine Abbildung  $\mathcal{A}$  gilt:

|                  |                  |                  |                  |
|------------------|------------------|------------------|------------------|
| 8809 $\mapsto$ 6 | 7111 $\mapsto$ 0 | 2172 $\mapsto$ 0 | 6666 $\mapsto$ 4 |
| 1111 $\mapsto$ 0 | 3213 $\mapsto$ 0 | 7662 $\mapsto$ 2 | 9312 $\mapsto$ 1 |
| 0000 $\mapsto$ 4 | 2222 $\mapsto$ 0 | 3333 $\mapsto$ 0 | 5555 $\mapsto$ 0 |
| 8193 $\mapsto$ 3 | 8096 $\mapsto$ 5 | 7777 $\mapsto$ 0 | 9999 $\mapsto$ 4 |
| 7746 $\mapsto$ 2 | 6855 $\mapsto$ 3 | 9881 $\mapsto$ 5 | 5531 $\mapsto$ 0 |

Welchen Wert hat  $x$  in  $2581 \mapsto x$ ?