



---

Christian Uhlig

---

## Systemnahe Programmierung WS18/19

---

- **Systemnahe Programmierung (SP)** behandelt...
  - ... die Programmiersprache C und ihre Besonderheiten
  - ... die Funktionsweise von Maschinen auf Ebene der Maschinensprache
  - ... die Abbildung von Programmen einer Hochsprache in Programme für eine Maschine am Beispiel von C
  - ... Chancen und Risiken der Hardwarenähe von C
- Fokus auf Konzepten der Informationsdarstellung und Programmierung (also der Formulierung von Programmen), relativ wenig auf Datenstrukturen und Algorithmen
- Was macht eine Programmiersprache "systemnah"?
  - Nähe der Sprachabstraktionen und des Typsystems zu den Elementen der Maschine
  - Speichermodell: Der Speicher des Rechners als großes Array
  - Schwache Typprüfungen

auch Notation genannt

In anderen Sprachen richten sich Datentypen nach der jeweiligen Sprache, in C aber nach der Compiler

↳ Typsystem: Auflösungsmechanismus / Fehlerkennung für den Programmierer

## Bedienungsanleitung (2)

- SP vs. PS1 (*Syntax allerdings unterschiedlich*)
  - Verschiedene, allerdings durchaus ähnliche, Sprachen (C / Pascal) (*insbesondere Zeiger*)
  - Pascal allerdings weniger "systemnah" und in vielerlei Hinsicht sicherer
  - PS1 als Veranstaltung legt den Fokus neben der Erlernung der Programmiersprache Pascal auf die Grundkenntnisse der Informationsverarbeitung und Konzepte der (prozeduralen) Programmierung
  - SP setzt diese Kenntnisse voraus und ordnet C und seine Elemente entsprechend ein
- SP vs. PS2 (*Syntax ähnlich, Idee anders*)
  - Verschiedene Sprachen (C / Java)
  - Syntaktisch sehr ähnliche Sprachen, konzeptionell aber fundamental unterschiedlich (objektorientiert vs. prozedural, strenge vs. weniger strenge Typprüfung, starke Abstraktion von der Maschinenausführung usw.) (*Java: virtuelle Maschine*)
  - Fokus in PS2 auf den Abstraktionen der Objektorientierung und den Konzepten der Sprache Java, aber kaum Bezüge zur Implementierung in der Maschine
  - SP stellt starke Bezüge zu den Abläufen in der Maschine her

## Bedienungsanleitung (3)

---

- SP vs. AuD
  - SP behandelt den Umgang mit einer Programmiersprache
  - SP vermittelt Kenntnisse zur Funktionsweise eines Rechners auf Ebene der Maschinensprache
  - AuD hingegen behandelt Datenstrukturen und Algorithmen zur Lösung von bestimmten Problemen, die Programmiersprache Java dient hier vorrangig als Notation und wird als bekannt vorausgesetzt
  - Eher "was programmiere ich?" (AuD) als "wie programmiere ich?" (PS1, PS2, SP)  
*Notation*

- Wozu in C programmieren können? *(Sprache sehr rasant: schwache Typprüfung)*
  - Große Code-Basis und nach wie vor Neuprojekte, insbesondere im UNIX-Umfeld
  - Verbreitete "Anschlusssprachen" wie C++ oder C#
  - starke Kontrolle über die Abläufe in der Maschine, **potenziell** hohe Laufzeit- und Speichereffizienz *(z.B. Javas Prüfung auf outofBoundException fällt in C weg)*
  - Hardwarenähe z.B. für direkten Hardwarezugriff per Inline-Assembler
  - Anwendungen wie Computergrafik, Programmierung von Controllern usw.

*Laufzeit<sup>↑</sup>effizienz*

- Wozu die Maschine verstehen?
  - Um in C programmieren zu können
  - Grundrüstzeug eines Informatikers
  - Bewusstsein für die "Kosten" von Abstraktionen in Hochsprachen
  - Besseres Verständnis für Probleme der IT-Sicherheit *(Bei sicherheitskritischen Programmen wird C nicht verwendet)*

- Herausforderungen beim Programmieren in C
  - In vielen Fällen undefiniertes Verhalten möglich (z.B. nicht initialisierter Speicher) *durch Programmierer nicht definiert*
  - Nur eingeschränkte Portabilität (Teile der Programmsemantik können vom Compiler und/oder Ausführungsplattform abhängig sein) *(z.B. Big-/Little Endian)*
  - Wenig ausdrucksstark, keine Abstraktionen wie Listen, Tupel, Mengen, usw.
  - An vielen Stellen Verzicht auf Sicherheit, der Programmierer kann die Abstraktionen der Programmiersprache (Variablen, Objekte, Kontrollstrukturen, usw.) relativ leicht verlassen
  - **Grundannahme: Programmierer weiß, was er tut**
  - Bedeutet auch: Viele Programmfehler können nicht zur Übersetzungszeit erkannt werden, fehlerhafte Programme werden häufig vom Compiler akzeptiert
  - Programmieren in C erfordert ein hohes Maß an **Disziplin und Präzision**
  - Kompilierbarkeit eines Programm beweist nicht viel
  - Fehler in C-Programmen können zur Laufzeit scheinbar unerklärliches oder zufälliges Verhalten verursachen und sehr schwer zu finden sein *(Heimrechner kann andere Ergebnisse liefern als Rechner in der FH)*

- Herausforderungen beim Programmieren in C (Forts.)
  - Also...
    - ... in C niemals auf den Compiler verlassen
    - ... Sprachelemente in Syntax und Semantik **vollständig** verstehen, bevor man sie anwendet
    - ... stets bemühen, sofort den endgültigen Programmcode hinschreiben
    - ... nicht davon blenden lassen, dass die Syntax stark der von Java ähnelt (*z.B. automatische Speicherverwaltung nicht existent!*)
  - Gefährliches Vorgehen:
    - Programm achtlos herunterschreiben ("mach ich später nochmal richtig", "Fehler suche ich später", "denke ich später zu Ende")
    - Programm kompilieren, Warnungen und Fehler oberflächlich abstellen
    - Programm ausführen
      - Programm macht nichts Sinnvolles  
=> wo anfangen?
      - Programm schon ganz gut, aber merkwürdiges Fehlverhalten  
=> auch bei Kleinigkeiten möglicherweise aufwendige Fehlersuche

## Bedienungsanleitung (7)

- In manchen Wochen zusätzliche Vorlesungseinheiten im Termin der Aufgabeneinführung am Donnerstag
  - siehe dazu auch die Webseite  
[http://intern.fh-wedel.de/~uhl/sp\\_ws1819.html](http://intern.fh-wedel.de/~uhl/sp_ws1819.html)
  - Mi 17.10. 8:00 HS2 + Do 18.10. 12:30 HS5 (**morgen!**)
  - Mi 24.10. 8:00 HS2 + Do 25.10. 12:30 HS5
  - Mi 31.10. Feiertag + Do 01.11. 12:30 HS5
- Übungsveranstaltung
  - Leitung durch Malte Heins (mhe@fh-wedel.de) und Helga Karafiat (kar@fh-wedel.de)
  - Anmeldung vom **16.10.2018 19:30** bis **17.10.2018 16:00**
  - **gemeinsame Einführung zu den AuD- und SP-Übungen am 17.10.2018 17:00 in HS2 (Achtung: Anwesenheitspflicht)**
  - Möglichst konsequent Gruppen bilden

Oktober	
<i>Mi</i>	<i>Do</i>
3	4
10	11
17 Vorlesung 1 (8:00 HS2)	18 Vorlesung 2 (12:30 HS5)
24 Vorlesung 3 (8:00 HS2)	25 Vorlesung 4 (12:30 HS5)
31 Reformationstag	

November	
<i>Mi</i>	<i>Do</i>
	1 Vorlesung 5 (12:30 HS5)
7 Vorlesung 6 (8:00 HS2)	8
14 Vorlesung 7 (8:00 HS2)	15
21 Vorlesung 8 (8:00 HS2)	22
28 Vorlesung 9 (8:00 HS2)	29 Verfügungstag (vorlesungsfrei)

Dezember	
<i>Mi</i>	<i>Do</i>
5 Vorlesung 10 (8:00 HS2)	6
12 Vorlesung 11 (8:00 HS2)	13
19 Vorlesung 12 (8:00 HS2)	20 Letzter Vorlesungstag vor Weihnachten

## Bedienungsanleitung (8)

- Noch ein paar Worte zur Übung...
  - Insbesondere für die Studiengänge Medieninformatik und CGT wichtige **inhaltliche und formale Voraussetzung** für anschließende Veranstaltungen (im Bereich Computergrafik)
  - Übung kann nur jährlich angeboten werden, d.h. wer die Übung nicht erfüllt, muss ein Jahr warten und eine ganze Reihe von zentralen Veranstaltungen aufschieben  
=> erhebliche Verwerfungen im individuellen Studienplan
  - In der Vergangenheit hohe Durchfallquote
- Daher:
  - Aufgabenbearbeitung immer möglichst sofort beginnen
  - Sorgfältig und präzise arbeiten – Sprachelemente erst verstehen, dann einsetzen
  - Aufgaben können (im Vergleich zu anderen Übungen) überschaubar wirken, **aber:** Besonderheiten und Gefahren der Sprache immer im Hinterkopf behalten
  - Bei Problemen Hilfe nicht erst am letzten Tag der Bearbeitungszeit in Anspruch nehmen

## Bezahlte Assistenzen

---

- FH Wedel bietet seit einigen Semestern regelmäßig bezahlte Assistenzen an
  - Heißt typischerweise: In der Vorlesungszeit in Übungen als Abnehmer arbeiten
  - Praktischer Nebenverdienst, der auch das eigene Profil schärft
  - Einstiegsmöglichkeit in ein Mitarbeiterverhältnis und damit eine akademische Laufbahn
- Voraussetzungen
  - Lehrassistent aus dem Curriculum erfolgreich und engagiert absolviert
  - Gute Leistungen in den relevanten Fächern
- Bewerbung über das Assistenz-System

- **Imperative Programmiersprache** wie Pascal, Java, usw.  
D.h.: Wirkung eines Programms wird per schrittweiser Transformation eines Zustandes erreicht
- Zustandsbeschreibung per **Variablen**, Änderung des Zustandes per **Zuweisungen**, Steuerung der Transformation per **Anweisungen**, Berechnung von Werten mit **Ausdrücken**
- Kontrollstrukturen **Sequenz / Selektion / Iteration**
- **Prozedurale Programmiersprache**, d.h. zentraler Modularisierungsgegenstand ist die Prozedur / Funktion => **keine Objektorientierung**
- **Statische Typisierung**, allerdings simples Typsystem und relativ schwache Typprüfung (*Typen werden zur Übersetzungszeit, zur Laufzeit überhaupt nicht*)
- Viel Flexibilität im Umgang mit Zeigern
  - *Manuelle Speicherverwaltung*

- Ursprünglich geschaffen von **Dennis Ritchie** Anfang der **70er Jahre**
- Zunächst kein Standard, erste Beschreibung in "**The C Programming Language**" von **Kernighan und Ritchie** (1978), auch als K&R-C bezeichnet
- Standardisierung durch das ANSI (American National Standards Institute) im Jahr 1989 (**ANSI C**)
- Umgewandelt in einen ISO-Standard (ISO/IEC 9899:1990), bekannt als **C89**
- **C89 (ANSI C) ist Gegenstand dieser Veranstaltung**
- Erweiterungen von Standard C im Jahr 1995 (**C95**) und 1999 (**C99**)
  - Für VL irrelevant*
- ANSI C heute häufig als "portabler Assembler" charakterisiert *(Portabel bezieht sich auf Prozessor)*
- **C++**: Modernes C-Derivat mit objektorientierten Sprachelementen, vergleichsweise komplexe Sprache
- **C#**: Microsofts Gegenstück zu Java *(Etwas moderner als Java)*

## Literaturhinweise

- Harbison, Samuel; Steele, Guy L.  
**C – A Reference Manual**  
Fifth Edition  
2002  
*Zum Nachschlagen*
- Kernighan, Brian W.; Ritchie, Dennis M.  
**The C Programming Language**  
Second Edition  
2000  
*Bessere Einführung*
- Louden, Kenneth C.  
**Programming Languages: Principles and Practice**  
2. Auflage  
Thomson Learning, 2002
- Referenzen zur C-Standardsbibliothek z.B. per Google oder Linux manpages

## Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen

# Ein erstes C-Programm (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int namelength = strlen(argv[0]);
    ↑
    if(argc > 0)           Per standard Programmname
        fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);

    if(argc <= 1) {
        fprintf(stdout, "Keine Kommandozeilenparameter!\n");
    } else if(argc == 2) {
        fprintf(stdout, "Kommandozeilenparameter: %s\n", argv[1]);
    } else {
        int idx = 1;
        while(idx < argc) {
            fprintf(stdout, "Kommandozeilenparameter %d: %s\n", idx, argv[idx]);
            ++idx;
        }
    }

    return EXIT_SUCCESS;
}
```

**Zusammenfassung:**  
*Gibt den Programmnamen und seine Länge in Anzahl Zeichen aus, außerdem die Kommandozeilenparameter (soweit vorhanden)*

Länge des Programmnamens berechnen

Programmname und seine Länge in Zeichen ausgeben

kein Parameter => Fehlermeldung

1 Parameter => Ausgabe

>1 Parameter => Auflisten der Parameter

Meldung an Betriebssystem: Alles prima

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* ... */
```

- **C-Header-Dateien (\*.h)** werden eingebunden, um Funktionen aus Bibliotheken nutzen zu können (insbesondere der Standard-Bibliothek von C)
- Header-Dateien enthalten in der Regel keine Implementierungen, sondern lediglich "Bekanntmachungen" von verfügbaren Funktionen
  - Zur Verwendung einer Funktion muss dem Compiler ihre Signatur (Rückgabetyp, Parameter und ihre Typen) bekannt sein
  - Header-Dateien enthalten entsprechende Signaturen
  - Die eigentlichen Funktionsimplementierungen liegen bereits als Binärdateien (Bibliotheken) vor und müssen mit dem Programm verbunden werden
  - Für die Standard-Bibliothek geschieht das Binden der Bibliothek automatisch

```
#include <stdio.h>
#include <stdlib.h>• Dateien d. Standardbibliothek
• Auch möglich " zu verwenden (wird genutzt um lokale Ordner zu kennzeichnen)
#include <string.h>
/* ... */
```

- **Präprozessor-Direktive #include:**  
Jeweils Angabe einer einzubindenden Header-Datei (hier also **stdio.h**, **stdlib.h** und **string.h**)
- **stdio.h:** Standard Input/Output, insbesondere Dateizugriff  
**stdlib.h:** Allgemeine Hilfsmittel  
**string.h:** String-Behandlung
- Es können eigene Bibliotheken nebst entsprechenden Header-Dateien erstellt werden

## Ein erstes C-Programm (4) – Einbinden von Header-Dateien

```

#include <stdio.h> Ausgabe (Konsole, Datei,...)
#include <stdlib.h> } Warden alle sehr oft
#include <string.h> } verwendet

int main(int argc, char* argv[]) {
  int namelength = strlen(argv[0]);

  if(argc > 0)
    fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);

  if(argc <= 1) {
    fprintf(stdout, "Keine Kommandozeilenparameter!\n");
  } else if(argc == 2) {
    fprintf(stdout, "Kommandozeilenparameter: %s\n", argv[1]);
  } else {
    int idx = 1;
    while(idx < argc) {
      fprintf(stdout, "Kommandozeilenparameter %d: %s\n", idx, argv[idx]);
      ++idx;
    }
  }

  return EXIT_SUCCESS;
}

```

## Ein erstes C-Programm (5) – Funktionsdefinitionen

```
int main(int argc, char* argv[]) {
    /* ... */
}
```

*Definition: Mit Implementierung  
Deklaration: Ohne Implementierung*

- Angabe von **Rückgabetyp** und **Funktionsbezeichner**, Deklaration von **Parametern** (jeweils Typ und Bezeichner) sowie Angabe des **Funktionsrumpfes**

```
<Rückgabetyp> <Bezeichner> ( <Parameterdeklarationen> )
{
    <Funktionsrumpf>
}
```

- Funktionsrumpf ist eine Folge von **Variablen-deklarationen** und **Anweisungen**
- **Variablen-Deklaration:** Angabe des Bezeichners der Variablen, ihres Typen sowie **optional** eines Ausdrucks zur Initialisierung mit einem Wert

```
<Typ> <Bezeichner> = <Initialisierung> ;
    ↑
    Wenn weggelassen, Wert vom Speicher
```

## Ein erstes C-Programm (6) – Funktionsdefinitionen

```

int main(int argc, char* argv[]) {
  int namelength = strlen(argv[0]);      • Zuweisung nicht dasselbe
                                             wie eine Initialisierung

  if(argc > 0)
    fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);
  /* ... */
  return EXIT_SUCCESS;
}
  
```

- **Anweisungen**, z.B.

- **Leere Anweisung ' ; '**
- **Zuweisungen**
- **Block: { <Folge von Anweisungen> }**
- **Bedingte Anweisungen (if, switch)**
- **Schleifen (for, ...)**
- **usw.**

```
int main(int argc, char* argv[]) {
    int namelength = strlen(argv[0]);

    if(argc > 0)
        fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);
    /* ... */
    return EXIT_SUCCESS;
}
```

- hier also...
  - **Rückgabetyp:** `int` (ganze Zahlen)
  - **Parameter**
    - `argc` vom Typ `int` (ganze Zahlen)
    - `argv` vom Typ `char* []` (intuitiv ein **Feld von Zeichenfolgen**)
  - **Funktionsrumpf:**  
`Deklaration einer lokalen Variable namelength,`  
`anschließend Folge von Anweisungen`

- **Kein "Mixen" von Deklarationen und Anweisungen in C89:**

In jedem Funktionrumpf / jedem Anweisungsblock erst Deklarationen, dann Anweisungen (d.h. keine Deklarationen mehr nach der ersten Anweisung)

```
int foo() {  
    int x = 23; /* OK: Deklaration (mit Initializer, keine Zuweisung!) */  
    x = 42;      /* OK: Anweisung (Zuweisung) */  
    int y = 7;   /* Fehler: Deklaration nach erster Anweisung */  
}
```

- Jede Anweisungsfolge einer Funktion endet mit einer **return**-Anweisung, die den Rückgabewert definiert *Rückgabetyp void erlaubt, allerdings mit return / ohne in Ansi C*

```
int main(int argc, char* argv[]) {  
    /* ... */  
    return EXIT_SUCCESS;  
}
```

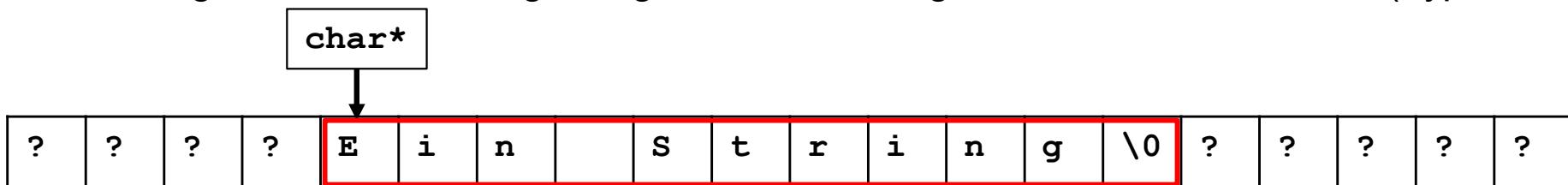
- **Prozeduren** sind in C kein eigenständiges Sprachelement, stattdessen können Funktionen den Rückgabetyp **void** aufweisen (konzeptionell ein Typ mit leerer Wertemenge)

```
void foo() {  
    fprintf(stdout, "%s", "OK: Funktion ohne Rückgabewert");  
    return;  
}
```

- **void**-Funktionen entsprechen also Prozeduren in anderen Sprachen, auf **return**-Anweisungen kann in diesem Fall verzichtet werden

```
void foo() {  
    fprintf(stdout, "%s", "OK: Funktion ohne Rückgabewert");  
}
```

- Ein paar Worte zu verfügbaren Typen (von Parametern / Variablen)
  - Ganzzahl-Typen in verschiedener Genauigkeit, es werden vorzeichenbehaftete und vorzeichenlose Varianten unterschieden (`int`, `unsigned int`, ...)
  - Aufzählungstypen (*Können sehr gut mit Ganzzahltypen kombiniert werden*)
  - Fließkomma-Typen in verschiedener Genauigkeit (`float`, ...)
  - Zeigertypen, deren Werte Verweise auf Speicherbereiche darstellen
  - Aggregatstypen (Arrays, strukturierte Typen)
  - In C89 **kein eigener Typ für boolesche Werte**
  - **Kein spezieller Typ für Strings**
    - Als String wird in C eine Sequenz von `char`-Werten im Speicher verstanden, die mit dem **NULL-Symbol** (binäre 0) abgeschlossen wird (*Länge d. Strings also um 1 Zeichen länger als d. Wort*)
    - Der Zugriff auf einen String erfolgt über einen Zeiger auf das erste Zeichen (Typ `char*`)



```
int main(int argc, char* argv[]) {  
    /* ... */  
    return EXIT_SUCCESS;  
}
```

- Besondere Funktion, die beim Programmstart aufgerufen wird
- Standardisierte Funktionsparameter zur Bereitstellung des Programmnamens und der Parameter des Programmaufrufs (in der Reihenfolge ihrer Angabe)
- **argv**
  - Kurzform für "**argument vector**"
  - Feld von Zeichenfolgen für Name und Aufrufparameter des Programms
- **argc**
  - Kurzform für "**argument count**"
  - Anzahl der Elemente (Feldgröße) in **argv**

• *argv.length wie in Java nicht möglich*

```
int main(int argc, char* argv[]) {  
    int namelength = strlen(argv[0]);  
    if(argc > 0)  
        fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);  
    /* ... */  
}
```

- Es gilt immer **argc**  $\geq 0$  (*Arraygröße positiv*)
- **argc** = 0 => keine Elemente in **argv**
- **argc** > 0 => **argv[0]** ist der **Programmname**
- **argc** > 1 => **argv[1]** bis **argv[argc-1]** sind die **Aufrufparameter**
- Der Programmname wird nach Belieben der Umgebung gesetzt und darf auch die leere Zeichenfolge sein (Länge 0) (*Es muss nicht unbedingt der richtige Programmname sein*)
- **Indizierter Zugriff** auf die Elemente: **argv[idx]** liefert die Zeichenfolge am Index **idx**, wobei der erste Index 0 ist

```
int main() {  
    /* ... */  
    return EXIT_SUCCESS;  
}
```

- Alternative Signatur, wenn keine Aufrufparameter benötigt werden:  
`int main(void)`                oder                `int main()`
- Namen der Parameter (`argc`, `argv`) sind nur Konvention, sie können beliebig anders gewählt werden, *muss aber ein int gefolgt von einem Char-Array sein*
- Rückgabewert
  - ganze Zahl, die den Aufrufer über das Ergebnis des Programmlaufs informiert ( $\text{== } 0$  für erfolgreich und  $\text{!= } 0$  für Fehlerfälle)
  - kleine Gefahr: Betriebssysteme können auch bestimmte Werte  $\text{!= } 0$  als Erfolgsmeldungen interpretieren
  - Standard-Bibliothek bietet mit `EXIT_FAILURE` eine Konstante, die garantiert einen Fehler signalisiert; symmetrisch dazu existiert `EXIT_SUCCESS` (immer 0)

## Ein erstes C-Programm (14) – main () -Funktion

```
int main(int argc, char* argv[]) {
    /* ... */
    return EXIT_SUCCESS;
}
```

- Beispiele für den Aufruf von `main()`

Aufruf an der Kommandozeile	argc	argv[1]	argv[2]
<code>ws1819.exe</code>	1	-	-
<code>ws1819.exe Test</code>	2	"Test"	-
<code>ws1819.exe Test Test</code>	3	"Test"	"Test"
<code>ws1819.exe "Test Test"</code>	2	"Test Test"	-
<code>ws1819.exe ""</code>	2	""	-

Beliebter Testfall

Bei Leerzeichen in  
Anführungszeichen setzen

undefined Zustand

```
int main(int argc, char* argv[]) {  
    int namelength = strlen(argv[0]);  
  
    fprintf(stdout, "Aufruf: %s\n", argv[0]);  
    /* ... */  
}
```

<Bezeichner> ( <tatsächliche Argumente> ) ;

- Verwendung von Funktionen erst nach Deklaration (d.h. Deklaration von Funktionen muss vorher im Quelltext erfolgen)
- Für **strlen** und **fprintf** erfolgt die Deklaration zuvor durch Einbinden der Header-Dateien
  - Funktionsaufruf ist ein Ausdruck, der einen Wert liefert
  - Ausdrücke in C sind zugleich Anweisungen, das Ergebnis der Auswertung wird in diesem Fall verworfen (**Beispiel oben: fprintf**)

```
int main(int argc, char* argv[]) {  
    /* ... */  
    fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);  
    /* ... */  
}
```

- **int `fprintf(<Stream>, <Formatstring>, ...)`;**
  - Als Stream können z.B. `stdout`, `stderr` oder geöffnete Dateien angegeben werden
  - Wie `fprintf()` sind `stdout` und `stderr` in der Header-Datei `stdio.h` deklariert
    - ↑ Kommandozeile
  - **Formatstring:** Weitgehend beliebige Zeichenfolge, in der **conversion specifications** (jeweils eingeleitet durch ein %-Zeichen) eingearbeitet sind (**hier: %s, %d**)
  - Die conversion specifications entsprechen der Reihe nach den weiteren Parametern von `fprintf()` (**hier: %s -> argv[0], %d -> namelength**)
  - Normale Zeichen werden einfach ausgegeben, **conversion specifications** hingegen führen zur formatierten Ausgabe des entsprechenden Parameterwertes von `fprintf()` *file print formatted*

```
if(argc <= 1) {  
    fprintf(stdout, "Keine Kommandozeilenparameter!\n");  
}
```

**if ( <Ausdruck> ) <Anweisung>**

- Entspricht der Selektion als Kontrollstruktur, d.h. bedingte Ausführung von Anweisungen im Programmablauf
- **Konzeptionelle Semantik**
  - Auswertung des angegebenen Ausdrucks
  - Wertet der Ausdruck zu "**wahr**" aus, wird die **Anweisung** ausgeführt
  - Wertet der Ausdruck zu "**falsch**" aus, wird nichts ausgeführt
  - Die Ausführung wird anschließend nach der **if**-Anweisung fortgesetzt

```
if(argc <= 1) {  
    fprintf(stdout, "Keine Kommandozeilenparameter!\n");  
}
```

- Vergleichsoperatoren liefern 1 → Wahr, 0 → Falsch
- Ganzzahlen als Bedingungen zugelassen

- **Aber:** C89 kennt keinen eigenen Datentypen für boolesche Werte  
=> was also ist mit "wahr" und "falsch" gemeint?
- **Semantik in C89**
  - Auswertung des angegebenen Ausdrucks zu einer ganzen Zahl
  - Wertet der Ausdruck zu einem Wert != 0 aus, wird die **Anweisung** ausgeführt
  - Wertet der Ausdruck zu 0 aus, wird nichts ausgeführt
  - Die Ausführung wird anschließend nach der **if**-Anweisung fortgesetzt

## Ein erstes C-Programm (19) – if-Anweisung

```
if(argc <= 1) {  
    fprintf(stdout, "Keine Kommandozeilenparameter!\n");  
} else {  
    fprintf(stdout, "Kommandozeilenparameter: %s\n", argv[1]);  
}
```

**if ( <Ausdruck> ) <AnweisungA> else <AnweisungB>**

### ▪ Semantik in C89

- Auswertung des angegebenen Ausdrucks
- Wertet der Ausdruck zu einem Wert != 0 aus, wird **AnweisungA** ausgeführt
- Wertet der Ausdruck zu 0 aus, wird **AnweisungB** ausgeführt
- Die Ausführung wird anschließend nach der **if**-Anweisung fortgesetzt

```
if(argc <= 1) {  
    fprintf(stdout, "Keine Kommandozeilenparameter!\n");  
} else if(argc == 2) {  
    fprintf(stdout, "Kommandozeilenparameter: %s\n", argv[1]);  
} else {  
    int idx = 1;  
    while(idx < argc) {  
        /* ... */  
    }  
}
```

**if ( <Ausdruck> ) <AnweisungA> else <AnweisungB>**

## ▪ Semantik in C89

- Auswertung des angegebenen Ausdrucks
- Wertet der Ausdruck zu einem Wert != 0 aus, wird **AnweisungA** ausgeführt
- Wertet der Ausdruck zu 0 aus, wird **AnweisungB** ausgeführt
- Die Ausführung wird anschließend nach der **if**-Anweisung fortgesetzt
- Mehr als 2 Fälle per Kaskadierung (**if - else if - ... - else**)

## Ein erstes C-Programm (20) – if-Anweisung

```

if(nameLength > 10)
if(nameLength < 20)
fprintf(stdout, "%d", 1);
else
fprintf(stdout, "%d", 2);
  
```

- Ausgabe für **nameLength == 5?**
- **else** gehört zum innersten **if**, das syntaktisch möglich ist
- Für **nameLength == 5** in diesem Fall gar keine Ausgabe

```

if(nameLength > 10)
  if(nameLength <
    fprintf(stdout, "%d", 1);
else
  fprintf(stdout, "%d", 2);
  
```



```

if(nameLength > 10)
  if(nameLength < 20)
    fprintf(stdout, "%d", 1);
else
  fprintf(stdout, "%d", 2);
  
```

- Was kommt als Anweisung in Frage?

- z.B. einzelner Funktionsaufruf

```
if(argc > 0)
    fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);
```

- z.B. leere Anweisung

```
if(argc > 0)
    ;
```

- Und wenn mehr als eine Anweisung benötigt wird?

**Anweisungsblöcke** (können überall stehen, wo eine Anweisung erwartet wird)

```
if(argc > 0) {
    fprintf(stdout, "Aufruf: %s\n", argv[0]);
    fprintf(stdout, "Anzahl Zeichen: %d\n", namelength);
}
```

- Anweisungsblock sinnlos, wenn nur eine Anweisung?

```
if(argc > 0)
    fprintf(stdout, "Aufruf: %s (%d Zeichen)\n", argv[0], namelength);
```

- "goto fail"-Bug

```
int Pruefe(...) {
    int err = 0;
    ...
    err = VorbereitungA(...);
    if (err)
        goto fail;
    err = VorbereitungB(...);
    if (err)
        goto fail;
    goto fail;
    err = VorbereitungC(...);
    if (err)
        goto fail;
    err = EigentlichePruefung(...);
    ...
fail:
    return err;
}
```

- "goto fail"-Bug
  - **err = VorbereitungB(...)**  
sei erfolgreich (soweit alles bestens)
  - Das erste **goto fail** wird also nicht ausgeführt
  - Das zweite **goto fail** hingegen wird (entgegen der irreführenden Einrückung) **immer** ausgeführt
  - **err = EigentlichePruefung(...)** wird übersprungen und die Ausführung an der Sprungmarke **fail:** fortgesetzt
  - **Aber:** **err** hat den Wert 0, **vorbereitungB()** hat keinen Fehler gemeldet
  - **Gänzlich ohne die eigentliche Prüfung ausgeführt zu haben, meldet Pruefe() nun also, dass alles ok ist**

```
int Pruefe(...) {  
    int err = 0;  
    ...  
    err = VorbereitungA(...);  
    if ( err )  
        goto fail;  
    err = VorbereitungB(...);  
    if ( err )  
        goto fail;  
    goto fail;  
    err = VorbereitungC(...);  
    if ( err )  
        goto fail;  
    err = EigentlichePruefung(...);  
    ...  
fail:  
    return err;  
}
```

⇒ „Geschweifte Klammern sind 'ne gute Sache.“

```
switch ( <Ausdruck> ) <Anweisung>
```

- Der **Ausdruck** liefert einen Ganzahl-Wert **x** (der hier aber nicht als Wahrheitswert interpretiert wird)
- Die **Anweisung** ist gewöhnlich ein Anweisungsblock, der sogenannte **case**- und optional ein **default**-Label enthält
- Ein **case**-Label definiert ein Sprungziel für eine bestimmte Zahl, ein **default**-Label nennt einfach nur ein Sprungziel

```
switch(<Ausdruck>) {  
    case <ganze Zahl>:  
        <Anweisungen>  
        break;  
    case <ganze Zahl>:  
        <Anweisungen>  
        break;  
    default:  
        <Anweisungen>  
}
```

```
switch(<Ausdruck>) {  
    case <ganze Zahl>:  
        <Anweisungen>  
        break;  
    case <ganze Zahl>:  
        <Anweisungen>  
        break;  
    default:  
        <Anweisungen>  
}
```

## ▪ Semantik

- Auswertung des angegebenen Ausdrucks zu einer Ganzzahl **x**
- Suchen eines **case**-Labels mit der Ganzzahl **x**
- **Wenn gefunden:** Sprung zu diesem **case**-Label, **ansonsten:** Suche nach einem **default**-Label
- **Wenn gefunden:** Sprung zum **default**-Label, **ansonsten** wird die Ausführung nach der **switch**-Anweisung fortgesetzt
- **break**-Anweisungen springen direkt zur nächsten Anweisung nach dem **switch**

- Syntaktisch einwandfreie und völlig sinnlose **switch**-Anweisungen:

```
switch(nameLength) ;  
switch(nameLength) { fprintf(stdout, "Shouldn't see me\n"); }  
switch(42) { case 42: fprintf(stdout, "Wichtige Information\n"); }
```

- Schon sinnvoller: Unsere geschachtelten **if**-Anweisungen als **switch**

```
switch(argc) {  
    case 0: Fall-through erlaubt  
    case 1:  
        fprintf(stdout, "Keine Kommandozeilenparameter!\n");  
        break;  
    case 2:  
        fprintf(stdout, "Kommandozeilenparameter: %s\n", argv[1]);  
        break;  
    default:  
        fprintf(stdout, "Zu viele Kommandozeilenargumente (%d) !\n", argc);  
}
```

• Wichtig: Switch ist deutlich schneller als korrespondierende if-Kaskade

**while ( <Ausdruck> ) <Anweisung>**

- Wiederholte Ausführung von **Anweisung**, solange **Ausdruck** zu "wahr" auswertet, d.h. zu einem Wert  $\neq 0$
- **Semantik in C89**
  - Auswertung des angegebenen Ausdrucks zu einer ganzen Zahl
  - Wertet der Ausdruck zu einem Wert  $\neq 0$  aus, wird die **Anweisung** ausgeführt und anschließend der vorherige Schritt wiederholt (Ausdruck auswerten usw.)
  - Wertet der Ausdruck zu 0 aus, wird die Programmausführung nach der **while**-Anweisung fortgesetzt
- **Beispiel:** Ausgabe der Programmargumente

```
int idx = 1;
while(idx < argc) {
    fprintf(stdout, "Kommandozeilenparameter %d: %s\n", idx, argv[idx]);
    ++idx;
}
```

- Syntax von **Bezeichnern**:

[A-Za-z\_] [A-Za-z0-9\_] \*

- **Beispiele:**

```
int _;                  /* merkwuerdig, aber OK */
int meinBezeichner;    /* OK */
int 99Luftballons;     /* Nicht OK */
int _99Luftballons;    /* OK */
```

- **Reservierte Worte**:

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	inline	int
long	register	return	short	signed	sizeof
static	struct	switch	typedef	union	unsigned
void	volatile	while			

### ▪ Kommentare

- mehrzeiliger Kommentar:  
"/\*" bis zum ersten "\*/"
- Auskommentierung der restlichen Zeile  
per "://" (leider erst in C99)

```
/* Negativer Preis? -> Fehlermeldung */
if(price < 0) {
    // leider nicht erlaubt in C89
    ...
}
```

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen

Typ-Kategorie	C-Typen
Integrale Typen (Ganzzahl-Typen)	<b>(signed / unsigned) char</b> <b>(signed / unsigned) short / int / long</b> <b>enum</b> (Aufzählungstyp)
Fließkomma-Typen	<b>float</b> <b>double</b> <b>long double</b> <i>besondere Genauigkeit</i>
Zeiger-Typen	<b>T *</b> , wobei T ein beliebiger anderer Typ ist
Array-Typen	<b>T [ ]</b> , wobei T ein beliebiger anderer Typ ist
Stukturierte Typen	<b>struct</b>
Vereinigungstypen	<b>union</b>
Funktionstypen	<b>T ()</b> , wobei T ein beliebiger anderer Typ ist
Void-Typ	<b>void</b>

## Typen – Überblick (2)

- Die **integralen** und die **Fließkomma-Typen** bilden zusammen die **arithmetischen Typen** (da sie regelmäßig die Operanden arithmetischer Operationen darstellen)
- Die **arithmetischen Typen** bilden zusammen mit den Zeigertypen die **skalaren Typen**
  - Skalare Typen weisen lediglich elementare Werte auf, d.h. sie besitzen keine Struktur und sind damit weder mehrwertig noch zusammengesetzt (*alleinstehender Wert*)
- Die **Array-Typen** bilden zusammen mit den **strukturierten Typen** die **Aggregats-Typen**
  - Aggregatstypen sind insofern strukturiert, als ihre Werte stets mehrere andere Werte zusammenfassen (z.B. ein Array von Integern)
- **Vereinigungs-** und **Funktionstypen** sowie der **Void-Typ** bilden eigenständige Typkategorien

## Typen – Überblick (3)

Typ-Kategorie	höhere Typ-Kategorie	höhere Typ-Kategorie
<b>Integrale Typen (Ganzzahl-Typen)</b>		
<b>Fließkomma-Typen</b>	<b>arithmetische Typen</b>	<b>Skalare Typen</b>
<b>Zeiger-Typen</b>		
<b>Array-Typen</b>		
<b>Stukturierte Typen</b>		<b>Aggregats-Typen</b>

## Typen – Ganzzahl-Typen (1)

- **unsigned** Ganzzahl-Typen:
  - n Wert-Bits, *(wie viele genau ist vom jeweiligen Compiler abhängig)*
  - Wertebereich 0 bis  $(2^n - 1)$
- **signed** Ganzzahl-Typen
  - n Wert-Bits + 1 Vorzeichenbit
  - **Vorzeichenbit nicht gesetzt:** Gleiche Werte wie der entsprechende **unsigned**-Typ mit einem Bit weniger
  - **Vorzeichenbit gesetzt:** 3 Repräsentationen für negative Zahlen in C89 zulässig
  - Darstellung per **Vorzeichen-Betrag**: Negierung des Betrages der Wert-Bits

VZ	4	2	1			
0	0	0	0	Vorzeichenbit nicht gesetzt, Betrag 0	=>	<b>Wert 0</b>
0	1	1	1	Vorzeichenbit nicht gesetzt, Betrag 7	=>	<b>Wert 7</b>
1	0	0	0	Vorzeichenbit gesetzt, Betrag 0	=>	<b>Wert -0</b>
1	1	0	1	Vorzeichenbit gesetzt, Betrag 5	=>	<b>Wert -5</b>
1	1	1	1	Vorzeichenbit gesetzt, Betrag 7	=>	<b>Wert -7</b>

## Typen – Ganzzahl-Typen (2)

- **signed** Ganzzahl-Typen (Forts.)

- Darstellung per **Vorzeichen-Betrag**: Wertebereich  $-(2^n-1)$  bis  $(2^n-1)$
- Darstellung per **Zweierkomplement**: Wert des Vorzeichenbits ist  $-(2^n)$

$-8$	$4$	$2$	$1$	<i>↪ wird von Hardware bevorzugt</i>
0	0	0	0	Vorzeichenbit nicht gesetzt, Betrag 0 => <b>Wert 0</b>
0	1	1	1	Vorzeichenbit nicht gesetzt, Betrag 7 => <b>Wert 7</b>
1	0	0	0	Vorzeichenbit gesetzt, Betrag 0 => <b>Wert -8</b>
1	1	0	1	Vorzeichenbit gesetzt, Betrag 5 => <b>Wert -3</b>
1	1	1	1	Vorzeichenbit gesetzt, Betrag 7 => <b>Wert -1</b>

Wertebereich  $-(2^n)$  bis  $(2^n-1)$       *- Normale Addition/Subtraktion möglich*

- Darstellung per **Einerkomplement**: Wert des Vorzeichenbits ist  $-(2^n-1)$

$-7$	$4$	$2$	$1$	
0	0	0	0	Vorzeichenbit nicht gesetzt, Betrag 0 => <b>Wert 0</b>
0	1	1	1	Vorzeichenbit nicht gesetzt, Betrag 7 => <b>Wert 7</b>
1	0	0	0	Vorzeichenbit gesetzt, Betrag 0 => <b>Wert -7</b>
1	1	0	1	Vorzeichenbit gesetzt, Betrag 5 => <b>Wert -2</b>
1	1	1	1	Vorzeichenbit gesetzt, Betrag 7 => <b>Wert -0</b>

Wertebereich  $-(2^n-1)$  bis  $(2^n-1)$

## Typen – Ganzzahl-Typen (3)

- **signed** Ganzzahl-Typen (Forts.)
  - **Einerkomplement / Vorzeichen-Betrag:**  
Doppelte (negative) Null, Besonderheiten bei Operationsausführung (insbesondere Addition)
  - **Negieren** von Zahlen
    - **Vorzeichen-Betrag:** Vorzeichen-Bit setzen
    - **Einerkomplement:** Bitweises Komplement *(Komplette Zahl)*
    - **Zweierkomplement:** Bitweises Komplement + 1
  - Alle positiven Werte eines **signed**-Typen auch im **unsigned**-Pendant repräsentierbar (grundsätzlich gleich viele oder weniger Wert-Bits im **signed**-Typ)

## Typen – Ganzzahl-Typen (4)

- Zentraler **Ganzzahl**-Typ: **char**
- Repräsentation des Basis-Ausführungszeichensatzes
- Basiszeichensatz muss mit einem **nicht-negativen char**-Wert repräsentierbar sein  
↳ min. 127 Werte
- Andere Zeichen können implementierungsabhängig als negative Zahlen repräsentiert werden
- **Gefahr:** Man weiß nicht, ob der Typ **char** auf einer gegebenen Plattform negative Zahlen unterstützt (also vorzeichenbehaftet ist) oder nicht  
**=>** gegebenenfalls explizit Verwendung von **signed char / unsigned char**
- "storage unit" == Byte == kleinste adressierbare Einheit == Typ **char**
- **Mindestens 8 Bit breit**, darf aber auch mehr sein
- **CHAR\_BIT** (Header-Datei **limits.h**): Anzahl Bits eines **char**-Wertes  
↳ Hinweis d. Compilers wie groß ein Byte sein soll.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	a	b	c	d	e	f	g	h	i	j
k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z		
0	1	2	3	4	5	6	7	8	9								
!	"	#	%	&	'	(	)	*	+	,	-	.	/	:			
;	<	=	>	?	[	\	]	^	_	{	}	~					
space, hori. tab, vert. tab, form feed, alert, backspace, carriage return, new line																	

- **Vorsicht mit der Nomenklatur:**

- Bytes typischerweise begrifflich fest mit einer Länge von 8 Bit verknüpft
- In C dürfen Bytes (bzw. das, was C als Bytes bezeichnet) mehr als 8 Bits haben
- Nicht gänzlich abwegig: Manche Maschinen adressieren tatsächlich kleinste Einheiten von z.B. 9 Bit
- z.B. relevant für bitweise Zugriffe auf Ganzzahl-Werte

## Typen – Ganzzahl-Typen (6)

- Größe von Werten sämtlicher Typen wird in Anzahl Bytes bzw. Anzahl **char** gemessen
- **sizeof**-Operator: Liefert die Anzahl Bytes eines Wertes bzw. Typen
  - **sizeof( <Typname> )** Größe der Werte des Typen in Anzahl **char**-Werten
  - **sizeof <Ausdruck>** Größe der Werte, die der Ausdruck liefert (bzw. der Typ d. Ausdrucks)
  - **sizeof(char)** per Definitionem == 1 → Operator zur Übersetzungszeit ausgerechnet, keine Performanceeinbußen zur Laufzeit.
  - **Beispiele**

```

int i = 0;

printf(stdout, "%ld\n", sizeof(int));

printf(stdout, "%ld\n", sizeof int); /* Syntaxfehler */

printf(stdout, "%ld\n", sizeof i);   Ausdruck, Klammern können weggelassen werden
printf(stdout, "%ld\n", sizeof(i)); /* Klammer-Ausdruck, Klammern
                                    gehören nicht zu sizeof */

```

## Typen – Ganzzahl-Typen (7)

- Weitere Ganzzahl-Typen neben **char**: **short**, **int**, **long**
- Jeweils Teilmenge des nächsten Typen (**char <= short <= int <= long**)
- Wie bei **char** jeweils **signed**- und **unsigned**-Varianten
- **unsigned**-Varianten haben immer den gleichen Speicherbedarf wie **signed**-Varianten
- **char** implementierungsabhängig entweder gleicher Wertebereich wie **unsigned char** oder wie **signed char**
- **short**, **int** und **long** entsprechen hingegen immer dem **signed**-Typen, hier ist das Schlüsselwort in der Regel redundant
- Die Typen "**signed**" bzw. "**unsigned**" entsprechen den Typen "**signed int**" bzw. "**unsigned int**" *(Sollte man nicht tun)*
- Header-Datei **limits.h** definiert Konstanten zu den tatsächlichen Wertebereichen jedes Ganzzahl-Typen für eine Implementierung, wobei der Standard für jede Schranke einen **kleinster Betrag** definiert

## Typen – Ganzzahl-Typen (11)

- Schranken in `limits.h` (Lieg bei Klausur als Beilage zur Verfügung)

Typ	untere Schranke	Wert	obere Schranke	Wert
char	<code>CHAR_MIN</code>	= 0 oder <code>SCHAR_MIN</code>	<code>CHAR_MAX</code>	= <code>SCHAR_MAX</code> oder <code>UCHAR_MAX</code>
<code>signed char</code>	<code>SCHAR_MIN</code>	$\leq -127 // -(2^7-1)$	<code>SCHAR_MAX</code>	$\geq +127 // (2^7-1)$ <i>mindestens</i>
<code>unsigned char</code>	immer 0		<code>UCHAR_MAX</code>	$(2^n-1), n = \text{CHAR\_BIT}$ $\geq +255 // (2^8-1)$
short	<code>SHRT_MIN</code>	$\leq -32.767 // -(2^{15}-1)$	<code>SHRT_MAX</code>	$\geq +32.767 // (2^{15}-1)$
<code>unsigned short</code>	immer 0		<code>USHRT_MAX</code>	$\geq +65.535 // (2^{16}-1)$
int	<code>INT_MIN</code>	$\leq -32.767 // -(2^{15}-1)$	<code>INT_MAX</code>	$\geq +32.767 // (2^{15}-1)$
<code>unsigned int</code>	immer 0		<code>UINT_MAX</code>	$\geq +65.535 // (2^{16}-1)$
long	<code>LONG_MIN</code>	$\leq -2.147.483.647 // -(2^{31}-1)$	<code>LONG_MAX</code>	$\geq +2.147.483.647 // (2^{31}-1)$
<code>unsigned long</code>	immer 0		<code>ULONG_MAX</code>	$\geq +4.294.967.295 // (2^{32}-1)$

## Typen – Ganzzahl-Typen (12)

- Mögliche Ganzzahl-Typen

<code>char</code>	<code>/* signed oder unsigned? */</code>		
<code>signed char</code>			
<code>unsigned char</code>			
<code>short</code>	<code>== short int</code>	<code>== signed short</code>	<code>== signed short int</code>
<code><u>unsigned short</u></code>	<code>== unsigned short int</code>		
<code>int</code>	<code>== signed int</code>	<code>== signed</code>	<code>== &lt;nix&gt;</code>
<code><u>unsigned int</u></code>	<code>== unsigned</code>		
<code>long</code>	<code>== long int</code>	<code>== signed long</code>	<code>== signed long int</code>
<code><u>unsigned long</u></code>	<code>== unsigned long int</code>		

## Typen – Ganzzahl-Typen (13)

- Verwendung von Ganzzahltypen für verschiedene Zwecke
  - **Ganzzahlen**
  - **einzelne Zeichen**
  - **Wahrheitswerte**: Interpretation von Ganzzahlen == 0 als **false** und != 0 als **true**
  - **Bit-Vektoren**: Folgen einzelner nutzbarer Bits *(Einzelne Bits d. Typen können spezielle Fkt. übernehmen (gesetzt vs. nicht gesetzt)).*

- **Allgemeine Syntax von Variablen-Deklarationen (vereinfacht):**

**<Speicherklasse?> <Typ-Qualifizierer\*> <Typ> <Deklarator+>**  
*(gar keiner bis beliebig viele)*

- **Deklarator:** Für Ganzzahl-Variablen lediglich Angabe des **Variablenbezeichners**, außerdem kann jeweils optional ein **Initialisierer** angegeben werden

**<Bezeichner> oder <Bezeichner> = <Initialisierer>**

- **Mehrere Variablen** (Deklaratoren) in einer Deklaration möglich

- Wie eine Anweisung wird eine Deklaration mit einem ; abgeschlossen

- **Beispiel:**      **<Speicherklasse?> <Typ-Qualifizierer\*> <Typ> <Deklarator+>**

auto

const

int a = 0, b, c = 42;

nicht initialisiert.

- **Wo** können Variablen deklariert werden?

- **Top-Level:** Außerhalb von Funktionsdeklarationen
- **Funktionsparameter:** Parameterliste einer Funktion
- **Funktionsrumpf:** Vor der ersten Anweisung
- **Block:** Vor der ersten Anweisung in einem Block

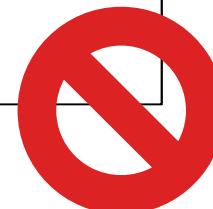
```
#include <...>

int a = 0;

void func(int b) {
    int c = 0;
    ...
    <Anweisungen>
    {
        int c = 0;
        ...
        <Anweisungen>
    }
}
```

- **Lokale Variablen:** Variablen in Funktionsrümpfen / Blöcken
- **Sichtbarkeit**
  - **Top-Level (file scope):** Deklarationsstelle bis Ende der Quellcodedatei
  - **Funktionsrumpf / Block, formale Parameter (local scope):** Deklarationsstelle bis Ende des Funktionsrumpfes / Blocks
- **Überladen**
  - Deklarationen innerhalb eines Blocks können Deklarationen außerhalb des Blocks verdecken
  - Funktionsparameter gehören zum jeweiligen Funktionsrumpf

```
void func(int v) {  
    char v = 'X';  
}
```



```
void func(int v) {  
    ...  
    {  
        char v = 'X';  
    }  
}
```



## ▪ Speicherklassen (für Variablen)

- **auto** (nur für lokale Variablen möglich): Eigener Zustand je Funktionsaufruf, zugewiesener Wert der Variable geht mit Abschluss des jeweiligen Aufrufs verloren

```
void storage_class_auto(int count) {
    int a = 1;
    fprintf(stdout, "call %d #1: int a == %d\n", count, a);
    a = a * 2;

    if(count < 2) {
        storage_class_auto(count + 1);
    }

    fprintf(stdout, "call %d #2: int a == %d\n", count, a);
    a = a * 2;
}
```

**auto** ist **Standard-Speicherklasse für lokale Variablen**, daher praktisch nie explizit im Quellcode

## ▪ Speicherklassen (für Variablen)

- **static** (für alle Variablen möglich): Nur ein Zustand mit **einmaliger Initialisierung**, Zustand bleibt nach Funktionsaufrufen erhalten (**bis Programmende**)

```
void storage_class_static(int count) {
    static int b = 1; ← Wird nur einmal initialisiert (rekursive Aufrufe, müssen entsprechend angepasst werden)
    fprintf(stdout, "call %d #1: static int b == %d\n", count, b);
    b = b * 2;

    if(count < 2) {
        storage_class_static(count + 1);
    }

    fprintf(stdout, "call %d #2: static int b == %d\n", count, b);
    b = b * 2;
}
```

**static** (bzw. eigentlich **extern**) ist **Standard-Speicherklasse von Top-Level-Variablen**

- **extern** (alle Variablen und Funktionen), **register** (lokale Variablen und Funktionsparameter): später...

## Deklaration – Ganzzahl-Variablen (5)

### ▪ Typ-Qualifizierer

- Typ-Qualifizierer sind grundsätzlich nur für Variablen auf der **linken Seite einer Zuweisung** von Relevanz
- **const**
  - Zuweisungen auf die Variable sind unzulässig (Initialisierungen hingegen natürlich nicht)

```
const int x = 42;
x = 23;           /* Fehler */
```

### ▪ **volatile**

- Zeigt an, dass sich der Wert einer Variablen ohne Zutun des Programmcodes ändern kann
- Als Folge unterlässt der Compiler mögliche Optimierungen (z.B. Ersetzen der Variablen durch einen konstanten Wert)
- **Beispiel:** Die Variable **x** könnte augenscheinlich eliminiert und durch **f(...)** ersetzt werden. Der Qualifier **volatile** zeigt jedoch an, dass sich der Wert der Variablen auch anderweitig ändern kann.

```
volatile int x;
...
x = f(...);
...
fprintf(stdout, "x = %d\n", x);
```

*↳ z.B. durch direkten Eingriff, zeigt an, dass Speicherplatz nicht wegoptimiert werden darf.*

## ▪ Initialisierung

### ▪ **statische** Variablen von Ganzzahltypen

- zur Initialisierung dürfen nur **konstante Ganzzahl-Ausdrücke** verwendet werden
- fehlt ein Initialisierer, so wird die Variable **automatisch mit 0 initialisiert**

```
static int a;                      /* OK: Default-Initialisierung mit 0 */
static int b = (21 + 21);           /* OK: Initialisierung mit konstant 42 */
static int c = strlen(...);         /* Fehler: kein konstanter Ausdruck */
```

### ▪ **automatische** Variablen von Ganzzahltypen

- zur Initialisierung dürfen **beliebige Ganzzahl-Ausdrücke** verwendet werden
- fehlt ein Initialisierer, so ist der Wert der Variablen **uninitialisiert und zufällig**

```
int a;                            /* keine Initialisierung, Wert ist zufällig */
int b = (21 + 21);                /* OK: Initialisierung mit konstant 42 */
int c = strlen(...);              /* OK: Initialisierung mit berechnetem Ausdruck */
```

## ▪ Deklaration mehrerer Variablen

- Angabe der Deklaratoren in einer einzigen Zeile

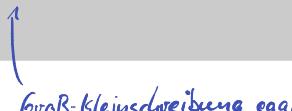
```
int i = 0, idx = 0, selected_column, x, found = 1;
```

- problematische Lesbarkeit (wieviele Variablen sind es?)
  - Initialisierung überall vorhanden?
  - Änderungen umständlich und fehlerträchtig
  - Kommentierung schlecht möglich
- 
- **Besser: Eine Variable je Zeile** (*Genauso gut, Typ in jede Zeile schreiben*)

```
int
    i                  = 0
    , idx              = 0
    , selected_column = 0 /* Kommentar */
    , x                = 0
    , found            = 1
;
```

```
int
    i                  = 0,
    idx              = 0,
    selected_column = 0, /* Kommentar */
    x                = 0,
    found            = 1
;
```

# Ganzzahl-Konstanten / -Literale

Darstellung	Syntax	Typen	Beispiele
Dezimal	<i>Erste Ziffer ist keine 0, sonst Ziffern 0-9: [1-9] [0-9]* &lt;IntSuffix?&gt;</i>	<code>int, long, unsigned long</code>	1 42
Oktal	<i>Erste Ziffer ist eine 0, Ziffern 0-7: 0 [0-7]* &lt;IntSuffix?&gt;</i>	<code>int, unsigned int, long, unsigned long</code>	0 01 052
Hexadezimal	<i>Prefix 0x bzw. 0X, Ziffern 0-f: 0 [xX] [0-9A-Fa-f]+ &lt;IntSuffix?&gt;</i>  	<code>int, unsigned int, long, unsigned long</code>	0x0 0x1 0x2a

- **Typ:** Der **erste zur Repräsentation ausreichende** aus der angegebenen Typ-Folge
- **Ganzzahl-Konstanten** sind immer positiv (außer bei Überlauf); ein eventuelles führendes '-' ist ein unärer Minus-Ausdruck und gehört nicht zur Konstanten selbst  
*Um Überläufe bei Ergebnissen bestimmter Berechnungen zu verhindern.*
- **<IntSuffix?>:** Erzwingen "größerer" Typen als zur Repräsentation erforderlich
  - **Suffix u/U:** `unsigned int, unsigned long` **Beispiel:** `42U` (-> `unsigned int`)
  - **Suffix l/L:** `long, unsigned long` **Beispiel:** `42L` (-> `long`)
  - **Suffix ul/UL:** `unsigned long` **Beispiel:** `42UL` (-> `unsigned long`)

- Angabe als Zeichen in einfachen Anführungszeichen: ' <Zeichen> '  
*↳ Muss ein Zeichen beinhalten*
- Werden vom Compiler in Ganzzahlen entsprechend dem Ausführungszeichensatz der Plattform übersetzt, z.B. 'a' == 97 für ASCII-Zeichensatz
- Steuerzeichen per **Escape-Sequenzen**, z.B. '\n', '\t', '\0' usw.
- Zeichenkonstanten sind vom Typ **int**
- Zuweisung auf / Initialisierung von **char**-Variablen: Konstantenart (Ganzzahl oder Zeichen) davon abhängig machen, wofür die Variable **konzeptionell** steht

```
char c = 42;
char c = 0x2A;
char c = 052;

char c = 97; /* Zahl oder Zeichen gemeint? */
char c = 'a'; /* lesbarer & portabel bzgl. Zeichensatz */

char c = ' '; /* Vorsicht: Zahl 32 (ASCII) */
char c = '0'; /* Vorsicht: Zahl 48 (ASCII) */

char c = 0; /* sinnvolle Standard-Initialisierung? */
char c = '\0'; /* verdeutlicht, dass c für ein Zeichen steht */
```

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen

## Typen – Aufzählungen (1)

- **Aufzählungstyp in C:** Menge von **benannten Ganzzahl-Konstanten**

```
enum <Tag-Bezeichner?> {
    <Aufzählungskonstante+>
}
```

*optional*

- Typdeklarationen an den gleichen Stellen im Programm wie eine Variablen-deklaration (und mit der gleichen Sichtbarkeit)
- Angabe mindestens einer **Aufzählungskonstante**
- Jede Aufzählungskonstante repräsentiert eine **Ganzzahl-Konstante** (Typ `int`)
- Als "Tag" und zu jeder Aufzählungskonstante ist ein im Sichtbarkeitsbereich **eindeutiger Bezeichner** anzugeben
- Für welche Ganzzahl steht eine Aufzählungskonstante?
  - Die erste Konstante erhält den Wert 0
  - Jede weitere Konstante erhält den Wert der Vorgänger-Konstante + 1

```
enum X {
    A == 0
    B == 1
    C == 2
};
```

## Typen – Aufzählungen (2)

- **Optional** jeweils explizite Angabe von Ganzzahl-Konstanten zu den Aufzählungsnamen

- Jeweils **beliebiger konstanter Ausdruck**, der einen Wert eines **integralen Typen** liefert (*und zur Übersetzungszeit ausgerechnet werden kann*)

```
enum Y {
    A = 10,
    B = 2*10,
    C = 3*10
};
```

**A == 10**  
**B == 20**  
**C == 30**

- **Keine Wertangabe zu einzelnen Konstanten:**  
Wie gehabt implizit Wert 0 bei erster Konstante bzw.  
Wert der vorherigen Konstante + 1 bei jeder anderen
- **Doppelte Werte** möglich
- **Negative Werte** möglich
- **Beliebige Reihenfolge der Werte**, jederzeit wieder kleinere Werte möglich

```
enum Z {
    A,
    B,
    C = 10,
    D,
    E = -1,
    F
};
```

**A == 0**  
**B == 1**  
**C == 10**  
**D == 11**  
**E == -1**  
**F == 0**

*fängt bei vorherigen Konstanten an zu erhöhen.*

## Typen – Aufzählungen (3)

- Ausdrücke dürfen Bezug auf **vorherige Aufzählungskonstanten** nehmen
- **Anwendungsbeispiel:** Bezeichnung des höchsten Wertes per eigener festbenannter Konstante

```

enum Priority {
    PRIO_LOW,
    PRIO_NORMAL,
    PRIO_HIGH,
    MAX_PRIO = PRIO_HIGH
}
...
/* Schleife über alle Prioritäten */
/* (0 bis MAX_PRIO) */

```

```

0 enum Priority {
1     PRIO_LOW,
2     PRIO_NORMAL,
3     PRIO_HIGH,
4     PRIO_EXTREME
5     MAX_PRIO = PRIO_EXTREME
}
...
/* Schleife über alle Prioritäten */
/* (0 bis MAX_PRIO) */

```

- Gegeben sei die Deklaration `enum Error { ERR_A, ERR_B, ERR_C };`
- Deklaration einer Variablen dieses Aufzählungstypen:  
`Error err; /* Fehler */` =>   
`enum Error err;`
- **Grund:** `Error` ist kein Typbezeichner, sondern nur ein sogenanntes Tag  
=> der eigentliche Typ heißt `enum Error`

## Deklaration – Aufzählungstyp-Variablen (1)

- Also: `enum Error { ERR_A, ERR_B, ERR_C };` deklariert...
  - ... das **Aufzählungs-Tag** `Error`
  - ... den **Aufzählungstypen** `enum Error`
  - ... die **Aufzählungskonstanten** `ERR_A, ERR_B` und `ERR_C` vom Typ `int`
- **Deklaration (zusätzlicher) Typ-Namen**
  - `typedef`-Deklaration gibt einem existierenden Typen einen neuen Namen
  - Syntax entspricht einer Variablen-deklaration (an den gleichen Stellen im Programm, gleiche Sichtbarkeit), der "Variablename" entspricht dem neuen Typnamen

`enum Error Err;`  
deklariert eine **Variable** `Err` vom Typ `enum Error`

`typedef enum Error Err;`  
deklariert den neuen **Typnamen** `Err` für den Typ `enum Error`

## Deklaration – Aufzählungstyp-Variablen (2)

- Weitere Varianten

- Deklaration von **Aufzählungs-Tag** **Error**, **Aufzählungstyp enum Error**, **Aufzählungskonstanten** und neuem **Typnamen Err** in einer Deklaration

```
typedef enum Error { ERR_A, ERR_B, ERR_C } Err;
```

- Deklaration von **Aufzählungs-Tag** **Error**, **Aufzählungstyp enum Error**, **Aufzählungskonstanten** und direkt der **neuen Variable err** in einer Deklaration

```
enum Error { ERR_A, ERR_B, ERR_C } err;
```

- Aufzählungs-Tag ist optional, es kann auch direkt ein neuer Typ-Name oder eine Variable deklariert werden. Dies kann sinnvoll sein, wenn der Aufzählungstyp direkt neu benannt wird oder nur für diese eine Variable verwendet werden soll (bzw. nur die Konstanten wiederverwendbar sein sollen).

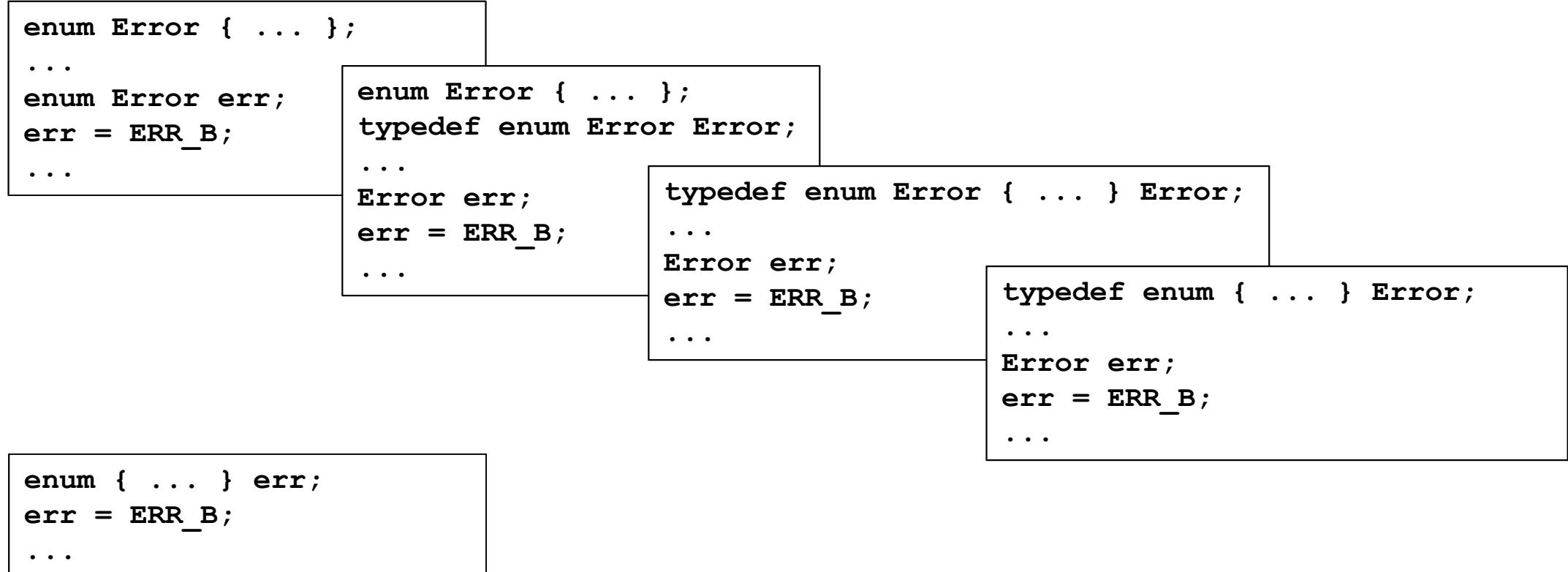
```
typedef enum { ERR_A, ERR_B, ERR_C } Err;
```

```
enum { ERR_A, ERR_B, ERR_C } err;
```

## Deklaration – Aufzählungstyp-Variablen (3)

- Weitere Varianten (Forts.)
- Typname darf dem Tag-Namen entsprechen (unterschiedliche Namensräume)

```
typedef enum Error { ERR_A, ERR_B, ERR_C } Error;
```



- Speicherbedarf einer **Variable** von einem Aufzählungstypen ist **implementationsabhängig** (gemeint ist in erster Linie der Compiler)
  - Forderung des Standards: Alle Aufzählungskonstanten müssen repräsentierbar sein
  - Darüber hinaus bestehen keine Vorgaben, insbesondere müssen solche Variablen nicht dem Speicherbedarf einer **int**-Variable entsprechen
  - Größenmessung wie üblich per **sizeof**, z.B.

```
sizeof( enum { ERR_A, ERR_B, ERR_C } )
```

- **Achtung:** `sizeof( ERR_A )` gibt **nicht** die Größe des Typen, sondern einer **int**-Konstanten an (macht bei vielen Implementierungen keinen Unterschied, darauf gibt es aber keine Garantie)

↳ Bei wenig Enumelementen wird weniger Speicherplatz genutzt, da geschaut wird wie viel Platz die Konstante die Anzahl annimmt.

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner

- **Einfache Zuweisung**

**<AusdruckL> = <AusdruckR>**

- **Semantik**

- Auswertung von **AusdruckR** zum Wert **r**
- Auswertung von **AusdruckL** zu einem Wert **1**, wobei **1** ein sogenannter **modifiable Ivalue** sein muss
- **Ivalues**
  - Ein **Ivalue** referenziert ein "Objekt" in einem festen Speicherbereich, das beobachtet und geändert werden kann
  - **non-modifiable Ivalue**: Der **Ivalue** verbietet die Änderung des referenzierten Objektes aus konzeptionellen Gründen (z.B. wenn sein Typ den Typ-Qualifier **const** aufweist)
  - **modifiable Ivalue**: Der **Ivalue** erlaubt Änderungen des referenzierten Objektes
  - Typische **modifiable Ivalues**: **Variablenwerte**
- Speicherung des Wertes **r** im Speicherbereich von **1**

## ▪ Einfache Zuweisung (Forts.)

- Zuweisungsausdruck liefert als Ergebnis den Wert zurück, der in 1 gespeichert wurde
- Zuweisungsausdruck hat den Typ von 1
- Zuweisungsausdruck liefert keinen lvalue
- Rechtsassoziative Auswertung

```
int a = 0, b = 0;  
  
a = b = 42; /* a == 42, b == 42 */  
a = (b = 42);  
  
a = 2 * (b = 42); /* a == 84, b == 42 */  
  
(a = 42) = 13; /* Fehler */
```

## ▪ Zusammengesetzte Zuweisung:

**<AusdruckL> <Op>= <AusdruckR>**

- Verkürzte Syntax für **<AusdruckL> = <AusdruckL> <Op> <AusdruckR>**
- Mögliche Varianten: **+=, -=, \*=, /=, %=, ...**
- **Beispiel:**

```
int i = 0;  
i += 10; /* entspricht i = i + 10; */
```

## ▪ Postfix Inkrement / Dekrement

<Ausdruck>++

<Ausdruck>--

- Liefert den Wert des Ausdrucks und inkrementiert bzw. dekrementiert den gespeicherten Wert **anschließend** um 1
- **Ausdruck** muss zu einem **modifiable lvalue** auswerten (z.B. Variable)
- anwendbar auf **skalare Typen** (also arithmetische und Zeiger-Typen)
- Semantik
  - wertet **Ausdruck** zu einem **modifiable lvalue x** aus
  - erhöht (++) bzw. verringert (--) den Wert **x** um 1
  - liefert den **ursprünglichen** Wert **x** als Ergebnis
- Ergebnis ist kein **lvalue**

```
int a = 0, b = 0;
a = b++;           /* a == 0, b == 1 */
b++;
/* a == 0, b == 2 */
(a++) = 42;        /* Fehler */
```

- **Prefix Inkrement / Dekrement**

**++<Ausdruck>**

**--<Ausdruck>**

- Inkrementiert bzw. dekrementiert den gespeicherten Wert um 1 und liefert den **neuen Wert** des Ausdrucks
- **Ausdruck** muss zu einem **modifiable lvalue** auswerten (z.B. Variable)
- anwendbar auf **skalare Typen** (also arithmetische und Zeiger-Typen)
- Semantik
  - wertet **Ausdruck** zu einem **modifiable lvalue x** aus
  - erhöht (++) bzw. verringert (--) den Wert **x** um 1
  - liefert den **neuen** Wert **x+1** bzw. **x-1** als Ergebnis
- Ergebnis ist kein **lvalue**

```
int a = 0, b = 0;
a = ++b;           /* a == 1, b == 1 */
++b;              /* a == 1, b == 2 */
(++a) = 42;        /* Fehler */
```

## Ausdrücke – Zuweisungen (5)

- Zuweisungskompatibilität
  - **arithmetische Typen** können anderen **arithmetischen Typen** zugewiesen werden
  - Zuweisung: **Integraler Typ => Integraler Typ**

- Wert im Zieltyp **repräsentierbar**  
**=> alles ok**

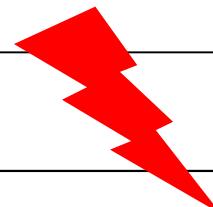
```

unsigned char ch = 123;
int           i = ch; /* i == 123 */
  
```

- Wert im Zieltyp **nicht repräsentierbar** und Zieltyp ist **signed**  
**=> Overflow, Wert undefiniert**

```

unsigned char uc = 200; /* 11001000 (200) */
signed char sc = uc;   /* undefiniert, häufig 11001000 (-56) */
  
```



- Wert im Zieltyp **nicht repräsentierbar** und Zieltyp ist **unsigned**  
**=> repräsentierbare Bits des Quellwertes bleiben erhalten,**  
**Wert entspricht Quellwert mod 2^n (n Bits des Zieltypen)**

```

int i = 1431655765;      /*          01010101 01010101 01010101 01010101 */
unsigned char ch = i;     /*          01010101 01010101 01010101 01010101 */
                           /*          01010101 */
  
```

## Ausdrücke – Zuweisungen (6)

- Zuweisungskompatibilität (Forts.)

```

typedef enum { CAR = 5, TRAIN = 10, PLANE = 15 } Vehicle;
typedef enum { NOTEBOOK = 1, DESKTOP = 2, SERVER = 3 } System;

Vehicle v = CAR;
System s = DESKTOP;

int i1 = CAR;
int i2 = TRAIN * SERVER + 2;

Vehicle v1 = NOTEBOOK;
Vehicle v2 = NOTEBOOK + PLANE;

```

- **Keinerlei Sicherheit:** Jeder arithmetische Typ auf jeden arithmetischen Typen zuweisbar
- Offensichtlich konzeptionell unsinnig, aber zulässig:

```

int i1 = CAR; /* i1 == 5 */
int i2 = TRAIN * SERVER + 2; /* i2 == 32 */
Vehicle v1 = NOTEBOOK; /* v1 == 1 */
Vehicle v2 = NOTEBOOK + PLANE; /* v2 == 16 */

```



## ▪ Literale (Konstanten)

- Auswertung zum Wert und Typ der Konstanten
- Sind mit Ausnahme von String-Konstanten keine **Ivalues**

```
int x      = 42;
x          = (5 + 6) * 3;
6          = 42;                  /* Fehler, Ganzzahl-Konstante ist kein lvalue */
```

## ▪ Bezeichner

- Auswertung zum Wert des mit dem Bezeichner verknüpften "Objektes"
  - Bezeichner von **Variablen** werten zu **Ivalues** aus
  - Bezeichner von **Aufzählungskonstanten** werten zur entsprechenden Ganzzahl-Konstanten aus und sind **keine Ivalues**

```
typedef enum { N = 5, M = 8 } Foo;
int x = 0, y = 0;
x = 10;                      /* x als lvalue in einem Zuweisungsausdruck */
y = 42 + 2 * x + N * M;     /* y als lvalue in einem Zuweisungsausdruck */
N = 1;                        /* Fehler, Aufzählungskonstante ist kein lvalue */
```

↳ "An sich nur eine Zahl die da steht..."

## Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen

## Anwendungsbeispiele für Aufzählungen (1)

- **Beispiel:** Routine mit Fehlerbehandlung (hier: Hauptprogramm)

```

int main(int argc, char *argv[]) {
  /* Variablendeclarationen */
  /* I/O, z.B. Öffnen von Dateien */
  if( ... ) {
    ...
    if( ... ) {
      fprintf(stderr, "%s", "....");
      return 1; /* Fehler: Datei nicht gefunden */
    } else if( ... ) {
      fprintf(stderr, "%s", "....");
      return 2; /* Fehler: Dateiinhalt fehlerhaft */
    }
    ...
  } else {
    fprintf(stderr, "%s", "....");
    return 3; /* Fehler: Aufrufsyntax fehlerhaft / ungültige Operation */
  }

  /* Dateien schließen */
  return 0;      /* gut gelaufen */
}

```

"längender Fehlerstring"

## Anwendungsbeispiele für Aufzählungen (2)

- **Beispiel:** Routine mit Fehlerbehandlung (hier: Hauptprogramm)

```

int main(int argc, char *argv[]) {
  /* Variablendeclarationen */
  /* I/O, z.B. Öffnen von Dateien */
  if( ... ) {
    ...
    if( ... ) {
      fprintf(stderr, "%s", "....");
      return ERR_FILE_NOT_FOUND;
    } else if( ... ) {
      fprintf(stderr, "%s", "....");
      return ERR_INVALID_FILE;
    }
    ...
  } else {
    fprintf(stderr, "%s", "....");
    return ERR_INVALID_OP;
  }

  /* Dateien schließen */
  return ERR_NONE;
}
  
```

```

enum Error {
  ERR_NONE          = 0,
  ERR_FILE_NOT_FOUND = 1,
  ERR_INVALID_FILE   = 2,
  ERR_INVALID_OP     = 3
};
  
```

## Anwendungsbeispiele für Aufzählungen (3)

- Beispiel: Routine mit Fehlerbehandlung (hier: Hauptprogramm)

```

int main(int argc, char *argv[]) {
    Error err = ERR_NONE;
    /* Variablen-deklarationen */
    /* I/O, z.B. Öffnen von Dateien */
    if( ... ) {
        ...
        if( ... ) {
            err = ERR_FILE_NOT_FOUND;
        } else if( ... ) {
            err = ERR_INVALID_FILE;
        }
        ...
    } else {
        err = ERR_INVALID_OP;
    }
    /* Dateien schließen */
    return error_handling(err);
}

```

```

typedef enum Error {
    ERR_NONE          = 0,
    ERR_FILE_NOT_FOUND = 1,
    ERR_INVALID_FILE   = 2,
    ERR_INVALID_OP      = 3
} Error;

```

```

Error error_handling(Error err) {
    switch(err) {
        case ERR_FILE_NOT_FOUND:
            fprintf(...);
            break;
        default:
            ...
    }
    return err;
}

```

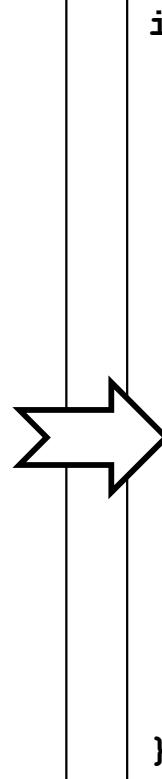
Redundant code  
vermeiden

Besonders wichtig  
 • Returns würden dies  
 erschweren (vorherige Folie)  
 • würden außerdem das  
 Debugging verkomplizieren

## Anwendungsbeispiele für Aufzählungen (4)

- Nochmal "goto fail"...

```
int Pruefe(...) {
    int err = 0;
    ...
    err = VorbereitungA(...);
    if (err)
        goto fail;
    err = VorbereitungB(...);
    if (err)
        goto fail;
    err = VorbereitungC(...);
    if (err)
        goto fail;
    err = EigentlichePruefung(...);
    ...
fail:
    return err;
}
```



```
int Pruefe(...) {
    int err = 0;
    ...
    err = VorbereitungA(...);
    if (!err) {
        err = VorbereitungB(...);
    }
    if (!err) {
        err = VorbereitungC(...);
    }
    if (!err) {
        err = EigentlichePruefung(...);
    }
    ...
    return err;
}
```

↑ können komplexe Blöcke sein  
↑ 1x return → Abbruch trotzdem zu einem richtigen Zeitpunkt.

- Strukturierte Programmierung:** Vermeidung von expliziten Sprüngen (`goto`) & systematische Bildung und Schachtelung bzw. Komposition von funktionalen Blöcken

## Anwendungsbeispiele für Aufzählungen (5)

- Immer nur ein `return` je Funktion?
  - Fallweise unterschiedlich zu bewerten
  - Spricht für ein einzelnes `return` am Ende:
    - Lange Funktionen
    - Komplexe Kontrollstrukturen / bedingte Schachtelungen
    - I/O-Lastigkeit (Ressourcenallokation am Anfang, Freigabe am Ende)
  - Spricht für mehrere `return`-Anweisungen:
    - Kurze Funktionen
    - Geringer Verschachtelungsgrad, sequenzieller Charakter
    - Keine Ressourcenverwaltung

```
CharType classify_char(char ch) {
    switch(ch) {
        case ' ':
        case '\t':
            return CT_SPACE;
        case '0':
            ...
        case '9':
            return CT_NUM;
        case 'A':
            ...
        case 'Z':
        case 'a':
            ...
        case 'z':
            return CT_ALPHA;
    default:
        return CT_MISC;
    }
}
```

"Absolut sinnvoll..."

```
typedef enum CharType {
    CT_MISC = 0,
    CT_SPACE = 1,
    CT_NUM = 2,
    CT_ALPHA = 3
} CharType;
```

## Anwendungsbeispiele für Aufzählungen (6)

- Trennung von **Modell** und **Benutzerschnittstelle**
  - **Beispiel:** Kommandozeilenaufruf `read -v my.data`
    - `-v` ist hier ein Kommandozeilenparameter zur Steuerung des Programmverhaltens (z.B. möglicherweise Kurzform von "verbose")
    - Auslagerung der Programmfunktionalität in Funktion `perform()`, einzelnes Zeichen `op` zur Ansteuerung entsprechend Kommandozeilsyntax

```
int main(int argc, char *argv[]) {
    char ch = ... argv ...;
    ...
    perform(ch, fname);
    ...
    return err;
}
```

```
void perform(char op, char *file) {
    /* ... */
}
```

- **Vermischung von Modell und Benutzerschnittstelle:**  
Änderungen an der Benutzerschnittstelle erfordern Änderungen an `perform()`, fehlende programminterne Abstraktion

## Anwendungsbeispiele für Aufzählungen (7)

- Trennung von **Modell** und **Benutzerschnittstelle**
- **Beispiel:** Kommandozeilenaufruf `read -v my.data`

```
int main(int argc, char *argv[]) {
    Operation op = OP_NONE;
    switch(input) {
        case 'v':
            op      = OP_VALIDATE;
            break;
        ...
        default:
            err     = ERR_INVALID_OP;
    }
    ...
    perform(op, fname);
    ...
    return err;
}
```

```
typedef enum Operation {
    OP_NONE      = 0,
    OP_VALIDATE  = 1,
    OP_LOAD      = 2,
    ...
} Operation;
```

```
void perform(Operation op, char *file) {
    /* ... */
}
```

Vorteile:

- Einfache/übersichtliche Wartbarkeit
  - ↳ Benutzer-Schnittstelle (Eingaben)
  - ↳ Code-Erweiterung

- Eigener Aufzählungstyp abstrahiert von den Festlegungen für die Benutzerschnittstelle, Benutzerschnittstelle und `perform()` können unabhängig voneinander variiert werden

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger

## Typen – Zeigertypen (1)

### ▪ Zeiger

- Wert repräsentiert einen Verweis auf den Anfang eines Speicherbereichs im Sinne einer Adresse
- Häufig Verweis auf den Speicher, in dem der Wert einer Variablen abgelegt ist
- **Typisiert:** Der Typ des Zeigers definiert den Typ des Wertes im referenzierten Speicherbereich (und damit die Länge dieses Speicherbereichs)
- Zu jedem Typen  $\mathbf{T}$  (einschließlich Zeigertypen selbst) existiert ein Typ "Zeiger auf  $\mathbf{T}$ "; es existieren also z.B. Zeiger auf `int`-Werte, Zeiger auf `char`-Werte usw.
- Differenzierung von Zeigertypen in **Objekt-Zeiger** (wenn  $\mathbf{T}$  ein Datentyp ist) und **Funktions-Zeiger** (wenn  $\mathbf{T}$  ein Funktionstyp ist)

## Typen – Zeigertypen (2)

### Beispiel

```
char    ch = 'a';
char *ptr = NULL;

fprintf(stdout, "  ch = %c\n", ch);
fprintf(stdout, "  ptr = %p\n\n", ptr);

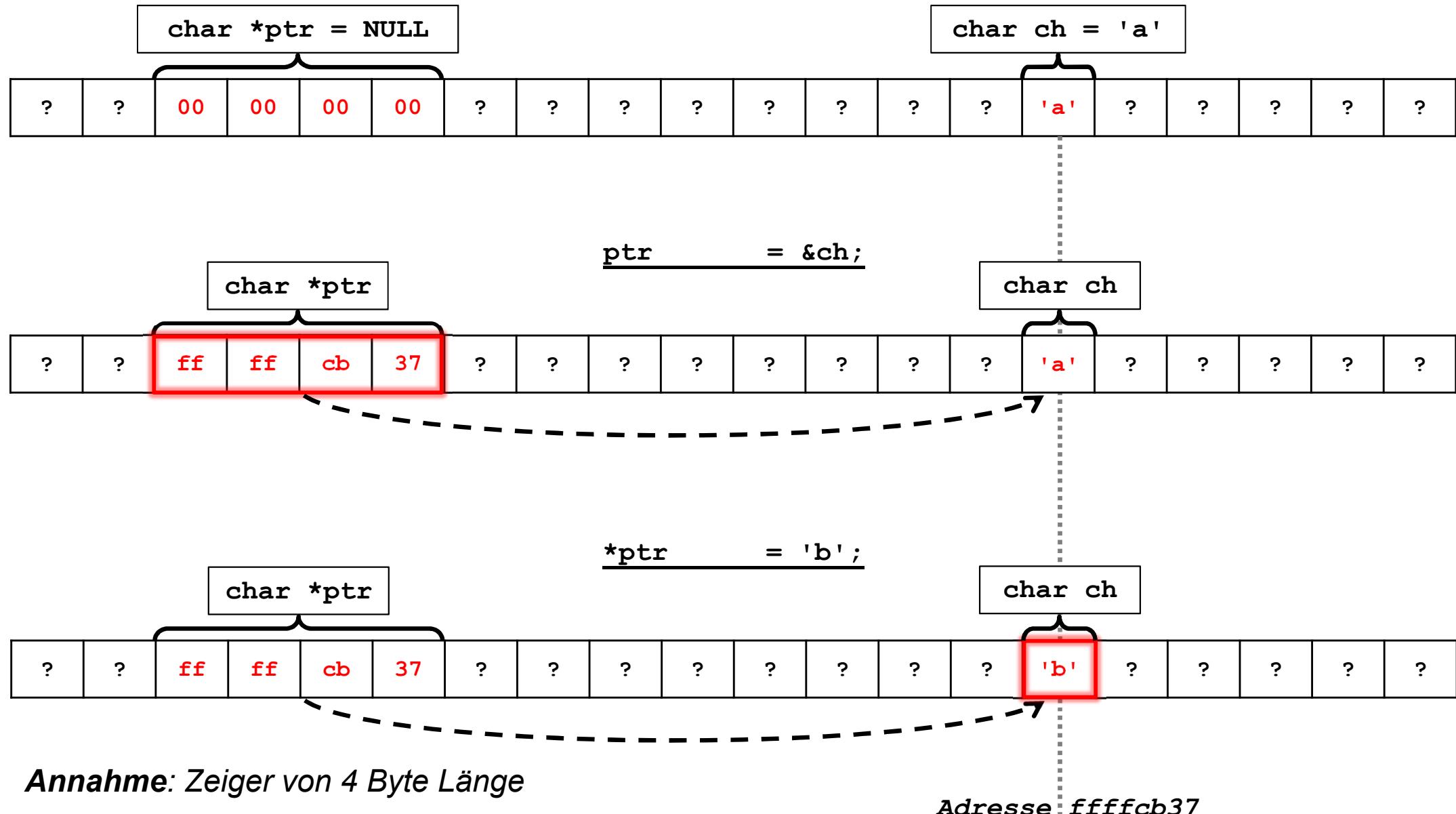
ptr      = &ch;
/* Adressoperator &, liefert einen Zeiger auf ein Objekt */

fprintf(stdout, "  ch = %c\n", ch);
fprintf(stdout, "  ptr = %p\n", ptr);
fprintf(stdout, "*ptr = %c\n\n", *ptr);
/* Dereferenzierung *: Auswertung zum referenzierten Wert */

*ptr      = 'b';

fprintf(stdout, "  ch = %c\n", ch);
fprintf(stdout, "  ptr = %p\n", ptr);
fprintf(stdout, "*ptr = %c\n\n", *ptr);
```

## Typen – Zeigertypen (3)



## Typen – Zeigertypen (4)

- **Größe eines Zeigertypen** per `sizeof`-Operator:  
Größe implementationsabhängig und potenziell unterschiedlich für verschiedene Zeigertypen
- **NULL-Zeiger**

```
char *ptr = NULL;           /* gleichwertig zu char *ptr = 0; */
```

- **spezieller Zeiger**, unterscheidet sich von **jedem gültigen Zeiger**
- zuweisungskompatibel zu **jedem Zeigertypen**
- definiert als Ganzzahl-Ausdruck mit dem Wert 0
- Konstante `NULL` in Header-Datei `stddef.h`
- möglicherweise repräsentiert als binäre 0 (alle Bits 0), aber nicht zwingend
- Vergleich `NULL == 0` wertet zu `true` aus (unabhängig davon, ob der `NULL`-Zeiger als binäre 0 repräsentiert ist)  
=> damit auch Interpretation als boolesches `false`, jeder andere Zeiger als `true`
- **Sicherheit:** Alle Zeiger-Variablen sollten den Wert `NULL` enthalten, wenn sie nicht auf einen definierten Speicherbereich zeigen

## ▪ & Adress-Operator

```
char ch = 'a';
char *ptr = &ch;
```

## Typen – Zeigertypen (6)

- \* Dereferenzierungs-Operator (Indirection)

```

char ch = 'a';
char *ptr = &ch;
*ptr      = 'b';
  
```

- Liefert zu einem Zeiger einen **Ivalue**, der dem **referenzierten Wert** entspricht
- Operand muss von einem Zeigertypen sein ("Zeiger auf **T**")
- Ergebnis ist ein **Ivalue** vom Typ **T**
- Eine Qualifizierung von **T** (insbesondere **const**) bleibt erhalten, d.h. Zuweisungen sind nur möglich, wenn der **Ivalue** auch **modifiable** ist

```

const char ch = 'a';
const char *ptr = &ch;
*ptr          = 'b'; /* Fehler: Typ von *ptr ist "const char" */
  
```

- Inverse Semantik zum **&**-Operator, d.h.: **\*(&x) == x** für jede Variable **x**

```

char ch = 'a';
*(&ch) = 'b';           /* gleichwertig zu: ch = 'b'; */
  
```

- Verhalten für ungültige Zeiger oder **NULL**-Zeiger **undefined**

## Typen – Zeigertypen (7)

- Gültigkeit von Zeigern: Zeiger auf Variable `num` mit Speicherklasse `auto`

```

void foo(...) {
    int *ptr = NULL;
    {
        int num = 42;
        ptr = &num;
        fprintf(stdout, "%d\n", *ptr);
    }
    fprintf(stdout, "%d\n", *ptr);
}
  
```

*Lebensdauer der Variable überschritten, Speicherplatz  
 kann mittlerweile neu belegt worden sein.*

```

int* foo(...) {
    int num = 42;
    return &num;
}
...
int* ptr = foo(...);
fprintf(stdout, "%d\n", *ptr);
  
```



- Gültigkeit von Zeigern: Zeiger auf Variable `num` mit Speicherklasse `static`

```

void foo(...) {
    int *ptr = NULL;
    {
        static int num = 42;
        ptr = &num;
        fprintf(stdout, "%d\n", *ptr);
    }
    fprintf(stdout, "%d\n", *ptr);
}
  
```

```

int* foo(...) {
    static int num = 42;
    return &num;
}
...
int* ptr = foo(...);
fprintf(stdout, "%d\n", *ptr);
  
```

## Typen – Zeigertypen (8)

- Typische Nutzung von Zeigern:  
Nachbildung von Referenzparametern an Funktionen

```
int getDbl(char *str, double *fp) {
    char dummy = '\0';           Parse-Ergebnis
    double lfp = 0.0;
    if( sscanf(str, "%lf%c", &lfp, &dummy) == 1 ) {
        *fp = lfp;             ←
        return 1;
    } else {                     Nötig da d. Zeiger beschrieben wird, selbst
        return 0;               bei erfolglosem Aufruf (weiter chars nach
                                Fießkommazahl)
    }
}
```

```
void foo(char *numstr, ...) {
    ...
    double num = 0.0;
    if( getDbl(numstr, &num) ) {
        ...
    }
    ...
}
```

- **Parameter `double *fp`:**  
Aufrufer kann einen Zeiger auf eine lokale Variable übergeben (hier: `num` in `foo()`)
- Im Rumpf von `getDbl()` kann der Wert der Variablen des Aufrufers geändert werden (per `*fp`)
  - Zuweisungen auf `fp` (z.B. `fp = NULL`) haben lediglich lokale Wirkung in `getDbl()`
  - Zuweisungen auf `*fp` (z.B. `*fp = 42.0`) hingegen wirken auf den Zustand im aufrufenden Programmcode (hier auf die Variable `num` in `foo()`)

# Deklaration – Zeiger-Variablen

<Speicherklasse?> <Typ-Qualifizierer\*> <Typ> <Deklarator+>

einfacher Deklarator = <Bezeichner>

auto	const	int	a = 42;
------	-------	-----	---------

Zeiger-Deklarator = \* <Typ-Qualifizierer\*> <Deklarator>

auto	const	char	*ptr = NULL;
------	-------	------	--------------

- Typ-Qualifizierer für den Typ, auf den gezeigt wird, und **zusätzlich** im Deklarator Typ-Qualifizierer (**const, volatile**) für den Zeiger selbst

<pre>const char *      ptr; /* nicht-konstanter Zeiger auf konstanten char-Wert */ char *const ptr; /* konstanter Zeiger auf nicht-konstanten char-Wert */ const char *const ptr; /* konstanter Zeiger auf konstanten char-Wert */</pre>
--

- **Achtung:** \* syntaktisch Teil des Deklarators

<pre>char* ptr1, ptr2;      /* ptr1 hat Typ "char*", ptr2 hat Typ "char" */ =&gt; char *ptr1, *ptr2;</pre>
--

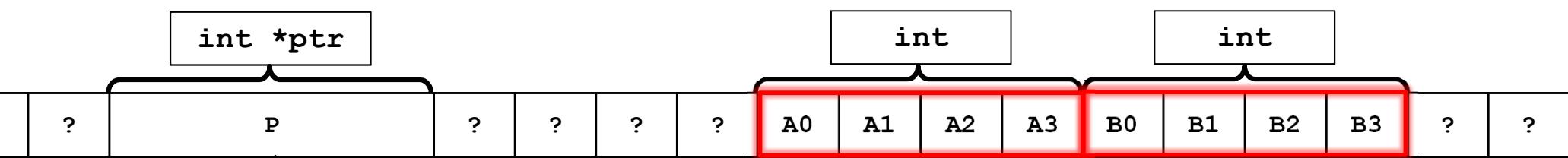
- Automatische Initialisierung von **statischen Zeiger-Variablen** mit **NULL**

## ▪ Zeigerarithmetik

- Addition (bzw. Subtraktion) von **Zeigern und Ganzzahlen** ergibt wieder einen Zeiger

### ▪ Semantik

- Zeiger **p** verweise auf das **n.** Element einer Folge von Elementen des Typs **T** im Speicher
- Zum Zeiger **p** werde eine ganze Zahl **o** addiert (**r = p + o** bzw. **o + p**)
- Das Ergebnis **r** ist ein "Zeiger auf **T**", der auf das **(n+o)**. Element des Feldes zeigt (bzw. auf die von **p** referenzierte Adresse zzgl. **o \* sizeof(T)** Bytes)



```

int *ptr = ...;
fprintf(stdout, "%d\n", *ptr);
fprintf(stdout, "%d\n", *(ptr+1));
...

```

*Einmal d. Basistyp d. Zeigers*

```

int *ptr = ...;
fprintf(stdout, "%d\n", *ptr);
ptr++;
fprintf(stdout, "%d\n", *ptr);
...

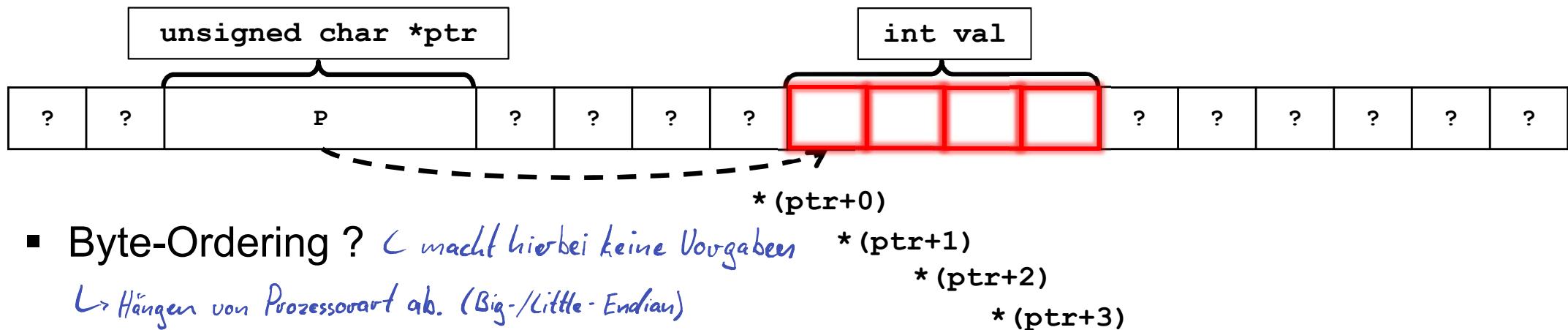
```

- **Anwendungsbeispiel:**  
Bytes eines beliebigen Datenobjektes ausgeben

```

int           val = 0x113377ff;
unsigned char *ptr = (unsigned char*) &val;

unsigned int    i = 0;
for(i = 0 ; i < sizeof(val) ; i++) {
    fprintf(stdout, "%p=%x\n", (ptr+i), *(ptr+i));
}
  
```



- Byte-Ordering ? ↳ macht hierbei keine Vorgaben  
 ↳ Hängt von Prozessorart ab. (Big-/Little-Endian)
  - $* (ptr+0)$
  - $* (ptr+1)$
  - $* (ptr+2)$
  - $* (ptr+3)$

## ▪ [] Index-Operator (Subscripting)

**<Ausdruck> [ <Ausdruck> ]**

- Ein **Ausdruck** muss zu einem "Zeiger auf T" und der andere **Ausdruck** zu einer **Ganzzahl** auswerten
- Entspricht **Zeiger-Arithmetik + Dereferenzierung**
- **Semantik:**  $a[b] == *((a) + (b))$   
d.h. **a** und **b** werden addiert und das Ergebnis dereferenziert
- Ergebnis ist ein **Ivalue** vom Typ **T**

```
int          val = 0x113377ff;
unsigned char *ptr = (unsigned char*)&val;
unsigned int    i = 0;
for(i = 0 ; i < sizeof(val) ; i++) {
    fprintf(stdout, "%p=%x\n", (ptr+i), ptr[i]);
}
```

- Typischerweise 1. Ausdruck ein Zeiger und 2. Ausdruck eine Ganzzahl, prinzipiell aber umgekehrt möglich: **a[b] == b[a]** oder **\*ptr == ptr[0] == 0[ptr]**

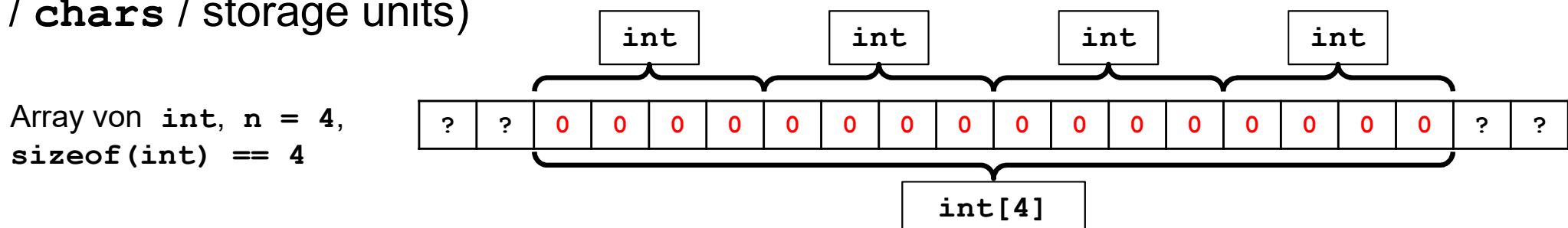
# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays

## Typen – Array-Typen (1)

- **Array:**
  - Feld (Sequenz) von Werten eines Typs  $T$
  - Indizierter Zugriff beginnend bei 0
- Zu (fast) jedem Typen  $T$  (u.a. nicht zu `void`) existiert ein Typ "Array von  $T$ ", der Felder von Werten des Typen  $T$  repräsentiert
- Größe  $n$  in Anzahl Elementen des Typs  $T$
- Ohne Größenangabe ist ein Arraytyp "**unvollständig**", es können keine Variablen eines solchen Typen deklariert werden
- **Repräsentation:** Ein Array von  $T$  der Größe  $n$  wird durch einen **sequenziellen Speicherbereich** von  $n * \text{sizeof}(T)$  repräsentiert (in Bytes / `chars` / storage units)



# Deklaration – Array-Variablen (1)

<Speicherklasse?> <Typ-Qualifizierer\*> <Typ> <Deklarator+>

einfacher Deklarator = <Bezeichner>

auto	const	int	a = 42;
------	-------	-----	---------

Zeiger-Deklarator = \* <Typ-Qualifizierer\*> <Deklarator>

auto	const	char	*ptr = NULL;
------	-------	------	--------------

Array-Deklarator = <Deklarator> [ <Größe?> ]

auto	const	char	arr[5];
------	-------	------	---------

- Größe ist ein **konstanter Ganzzahl-Ausdruck**
- **Kombination mit Zeiger-Typen**
  - Bindungsstärke von Array-Deklaratoren höher als von Zeiger-Deklaratoren
  - Abweichungen per Klammerung möglich

<pre>char      *arr[10]; /* char *(arr[10]): Array von 10 Zeigern auf char */ char    (*arr)[10]; /* Zeiger auf ein Array von 10 char-Werten */ const char arr[5]; /* Array von 5 konstanten char-Werten */</pre>
---

## Deklaration – Array-Variablen (2)

### ▪ Initialisierung

**<Array-Deklarator> = { <Feldwert+> }**

- Initializer gibt der Reihe nach Werte für die Feldelemente an
- **Array-Deklarator mit Größenangabe (wertet zur Ganzzahl n aus):**  
Anzahl der Werte des Initializers muss  $\geq 1$  und  $\leq n$  sein; fehlende Werte werden durch die **Standard-Initialisierung für statische Variablen** ergänzt

<pre>int arr[4]; int arr[4] = { 1, 2, 3, 4 }; int arr[4] = { 1 }; <i>Nur an 1. Stelle initialisiert</i> int arr[4] = { 0 }; int arr[4] = { }; int arr[4] = { 1, 2, 3, 4, 5 }; /* Fehler: zu viele Werte */ </pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>?</td><td>?</td><td>?</td><td>?</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	?	?	?	?	1	2	3	4	1	0	0	0	0	0	0	0
?	?	?	?														
1	2	3	4														
1	0	0	0														
0	0	0	0														

/\* Syntaxfehler \*/

- **Array-Deklarator ohne Größenangabe:** Initializer bestimmt die Größe des Arrays

<pre>int arr[] = { 1, 2, 3, 4 }; /* int arr[4] */</pre>	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	1	2	3	4
1	2	3	4		

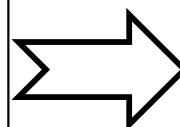
- Variablendeklarationen weder mit Größen- noch mit Initializer-Angabe sind **in der Regel unzulässig (unvollständiger Typ)**, z.B. `int arr[];`

## Typen – Arraytypen (2)

- Implizit **Zeigerarithmetik** für den Zugriff auf Array-Elemente
  - `int arr[4]`: `arr[0]` liefert **Ivalue** zum ersten Element, `arr[1]` zum zweiten usw.
  - Index-Operator `[]` aber nur für Zeiger-Werte definiert – wie funktioniert das?
  - Im Rahmen eines Ausdrucks wird ein Array-Bezeichner wie `arr` (Typ `int[4]`) **implizit** in einen Zeiger **auf das erste Array-Element** konvertiert (hier: Typ `int*`)

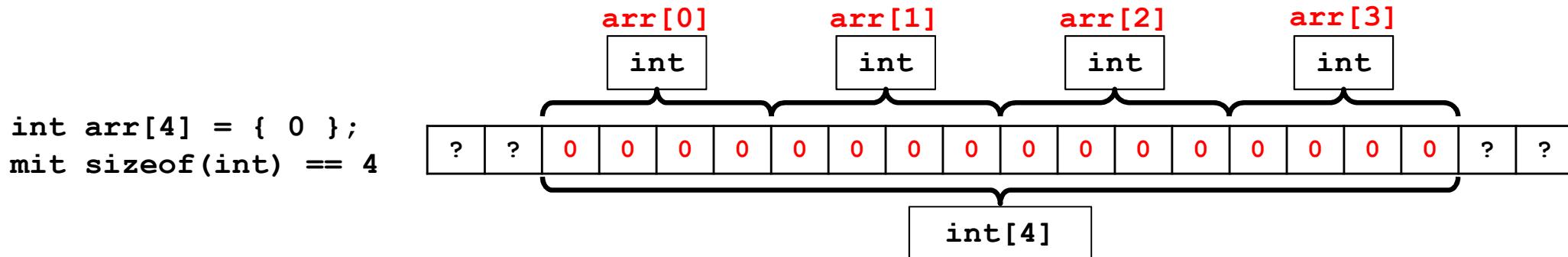
```

int arr[4] = { 0 };
for(i = 0 ; i < 4 ; i++) {
  fprintf(stdout, "%d ", arr[i]);
}
    
```



```

int arr[4] = { 0 };
for(i = 0 ; i < 4 ; i++) {
  fprintf(stdout, "%d ", *(arr+i));
/* arr': int*, Zeiger auf erstes
   Element von arr */
}
    
```



## Typen – Arraytypen (3)

- Implizit **Zeigerarithmetik** für den Zugriff auf Array-Elemente (Forts.)

```
int arr[4]
```

- **Ausnahme:** Als Parameter von `sizeof` bleibt ein Array ein Array  
=> `sizeof(arr)` liefert die echte Größe des Arrays, nicht die Größe eines Zeigers
- **Ausnahme:** Als Parameter von `&` wird `arr` nicht in einen Zeiger konvertiert  
(sondern bleibt ein Array) => `&(arr)` liefert Zeiger auf das Array
- **Kein Boundary Check:** Zugriff außerhalb der Arraygrenzen möglich

```
int arr[4] = { 0 };
int i = 0;
for(i = -10 ; i < 10 ; i++) {
    fprintf(stdout, "%d ", arr[i]);
}
```

## Typen – Arraytypen (4)

- Verwendung in **Funktionsdeklarationen**

- Wird für einen **Funktionsparameter** ein Array-Typ angegeben, so wird dieser implizit durch einen **Zeiger-Typen ersetzt**

<code>void foo(int arr[10]) { ... }</code>	=>	<code>void foo(int *arr) { ... }</code>
--	----	---

↳ Größenangabe kann weggelassen werden (Wird in Praxis auch so getan)

- Größenangabe des Arrays hierbei **bedeutungslos**, kann beliebig sein oder entfallen
- Passt zur impliziten Konvertierung des Array-Bezeichners im Aufrufer

```
void func(...) {
    int vals[4] = { 0 };
    /* sizeof(vals)? */
    foo(vals); /* implizit Zeiger auf erstes Element */
}
```

```
void foo(int arr[10]) {
    /* sizeof(arr)? */
}
```

- `char *argv[]` in `main()`-Funktion  
=> `char **argv`, d.h. Zeiger auf `char`-Zeiger (Zeiger auf den ersten String)
- Deshalb auch `argc` als Information über `argv` erforderlich
- Arraytypen als **Rückgabetyp nicht zulässig** (Zeiger auf Arrays hingegen schon)

### ▪ Strings in Arrays

- **String:** Folge von **char**-Werten, abgeschlossen per Terminierungszeichen '\0'
- Speicherbedarf für eine Zeichenfolge der Länge **n** daher **n+1**
- **Initialisierung per String-Konstante**
  - Zeichen der String-Konstante werden in das Array kopiert
  - Das Terminierungssymbol wird mitkopiert, **wenn der Platz reicht**
  - Ohne Größenangabe wird die Arraygröße auf Anzahl Zeichen + 1 festgelegt

```

char str[] = "Test";
char str[8] = "Test";
char str[4] = "Test";
char str[3] = "Test";
char *str = "Test";
    
```

T	e	s	t	\0			
T	e	s	t	\0	0	0	0
T	e	s	t				

/\* Fehler \*/

/\* OK, ist aber etwas völlig anderes... \*/

## Typen – Arraytypen (6)

- **Beispiel:** Aufzählungswerte als Index in String-Array
  - Aufzählungstyp für Fehler
  - statisches Array von Strings, Index im Array entspricht Fehlercode

```

typedef enum {
    ERR_NONE = 0,
    ERR_A,
    ERR_B
} Error;  kennzeichnet globale Variablen

static char *ERR_STRINGS[] = {
    "", ↑
    "Fehler A",  Array mit Character-Pointern
    "Fehler B",  ([] bindet stärker als *)
};
...

Error err = ERR_NONE;
...
fprintf(stdout, "%s\n", ERR_STRINGS[err]);
...

```

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)

## Anweisungen: Schleifen (while) (1)

**while ( <Ausdruck> ) <Anweisung>**

**Ausdruck** muss Wert ergeben, der mit 0 verglichen werden kann (skalare Typen)

Pointer ebenfalls erlaubt

### Semantik

1. Auswertung von **Ausdruck**, Vergleich des Ergebnisses mit 0
    - Wertet der Ausdruck zu != 0 aus => **Schritt 2**
    - Wertet der Ausdruck zu == 0 aus => **Schritt 3**
  2. Ausführung der **Anweisung**, anschließend wird mit **Schritt 1** fortgesetzt
  3. Fortsetzung der Programmausführung nach der **while**-Anweisung
- **Beispiel:** Ausgabe der Programmargumente

```

int idx = 0;
while(idx < argc) {
  fprintf(stdout, "Parameter %d: %s\n", idx, argv[idx]);
  idx++;
}
  
```

## Anweisungen: Schleifen (while) (2)

- **Beispiel:** Traversieren eines Strings

```
void toUpperCase(char *str) {
    while(*str != '\0') { /* oder: while( *str ) { ... } */
        if(*str >= 'a' && *str <= 'z') {
            *str -= ('a' - 'A');
        }
        str++;      kleinbuchstaben in Ascii-Tabelle
                    unter Großbuchstaben.
    }
}
```

- **Beispiel:** Traversierung eines Feldes von Zeigern

- Besonderheit **argv**: Array ist in Wahrheit ein Element größer als **argc** behauptet, das letztes Element muss dabei gemäß Standard den **NUL**-Zeiger enthalten

```
int main(int argc, char *argv[]) {
    char **strs = argv;
    while( *strs != NULL ) {
        fprintf(stdout, "%s\n", *strs);
        strs++;
    }
}
```

```
int main(int argc, char *argv[]) {
    char **strs = argv;
    while( *strs ) {
        fprintf(stdout, "%s\n", *strs++);
    }
}
```

## Anweisungen: Schleifen (do while)

```
do <Anweisung> while ( <Ausdruck> )
```

Variante der **while**-Schleife, die ihren Rumpf mindestens einmal ausführt

### Semantik

1. Ausführung der **Anweisung**
  2. Auswertung von **Ausdruck**, Vergleich des Ergebnisses mit 0
    - Wertet der Ausdruck zu != 0 aus => **Schritt 1**
    - Wertet der Ausdruck zu == 0 aus => **Schritt 3**
  3. Fortsetzung der Programmausführung nach der **do while**-Anweisung
- **Beispiel:** Ausgabe der Programmargumente

```
int idx = 0;
do {
    fprintf(stdout, "Parameter %d: %s\n", idx, argv[idx]);
    idx++;
} while(idx < argc);
```

## Anweisungen: Schleifen (for) (1)

```
for ( <Init?> ; <Bedingung?> ; <Update?> ) <Anweisung>
```

**Init**, **Bedingung** und **Update** sind Ausdrücke (alle **optional**).

**Bedingung** muss Wert ergeben, der mit 0 verglichen werden kann (skalare Typen)

### Semantik

1. Wenn vorhanden: Auswertung von **Init**, Vergessen des Ergebniswertes  
(z.B. **Initialisierung** von Variablen, **aber**: Keine Deklarationen möglich)
2. Wenn vorhanden: Auswertung von **Bedingung**, Vergleich des Ergebnisses mit 0
  - Wertet der Ausdruck zu != 0 aus => **Schritt 3**
  - Wertet der Ausdruck zu == 0 aus => **Schritt 6**
3. Ausführung der **Anweisung**
4. Wenn vorhanden: Auswertung von **Update**, Vergessen des Ergebniswertes
5. Weiter mit **Schritt 2**
6. Fortsetzung der Programmausführung nach der **for**-Anweisung

## Anweisungen: Schleifen (for) (2)

```
for ( <Init?> ; <Bedingung?> ; <Update?> ) <Anweisung>
```

- **Beispiel:** Ausgabe der Programmargumente

```
int idx = 0;
for( idx = 0; idx < argc ; idx++ ) {
    fprintf(stdout, "Parameter %d: %s\n", idx, argv[idx]);
}
```

- **Beispiel:** Nutzung von `sizeof` zur Traversierung der Elemente eines Arrays

```
unsigned int idx = 0;
int arr[4]      = { 0 };
for(idx = 0 ; idx < (sizeof(arr) / sizeof(*arr)) ; idx++) {
    fprintf(stdout, "%d ", arr[idx]);
}
```

- **Beispiel:** Endlosschleife

```
for( ; ; ) {
    fprintf(stdout, "%s\n", "dumdidum...");
```

## Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)
10. Strukturierte Datentypen
11. Ausdrücke 2: Konvertierungen, Vergleich / Logik / Arithmetik

## Konvertierung (1)

- Gegenstand von Konvertierungen sind die **Ergebnisse von Ausdrücken**
- **Ziel:** Abbildung eines Wertes von Typ  $Q$  (Quelle) auf einen Wert von Typ  $z$  (Ziel)
- Definition von Konvertierungen nach Quell- und Zieltyp:  
Wie wird ein Wert von Typ  $Q$  in einen Wert von Typ  $z$  überführt?
- Typ  $Q$  kann immer in Typ  $Q$  konvertiert werden (**Identitätskonvertierung**)
- Was löst eine Konvertierung aus? Ausgehend von einem Ausdruck vom Typ  $Q$ , in welchen Typ  $z$  wird konvertiert?
  - **Cast:** Angabe der Konvertierung in einen Zieltypen  $z$  explizit im Programmcode
  - **Zuweisung:** Implizite Konvertierung bei Zuweisung eines Ausdrucks vom Typ  $Q$  auf einen **Ivalue** vom Typ  $z$
  - **Implizite unäre Konvertierungen:**
    - Implizite Konvertierung der Operanden von Operatoren
    - Festlegungen, in welchen Typ  $z$  ein Ausdruckstyp  $Q$  konvertiert wird


 Bsp. *not-Operator, setzt implizit bestimmten Typ voraus*

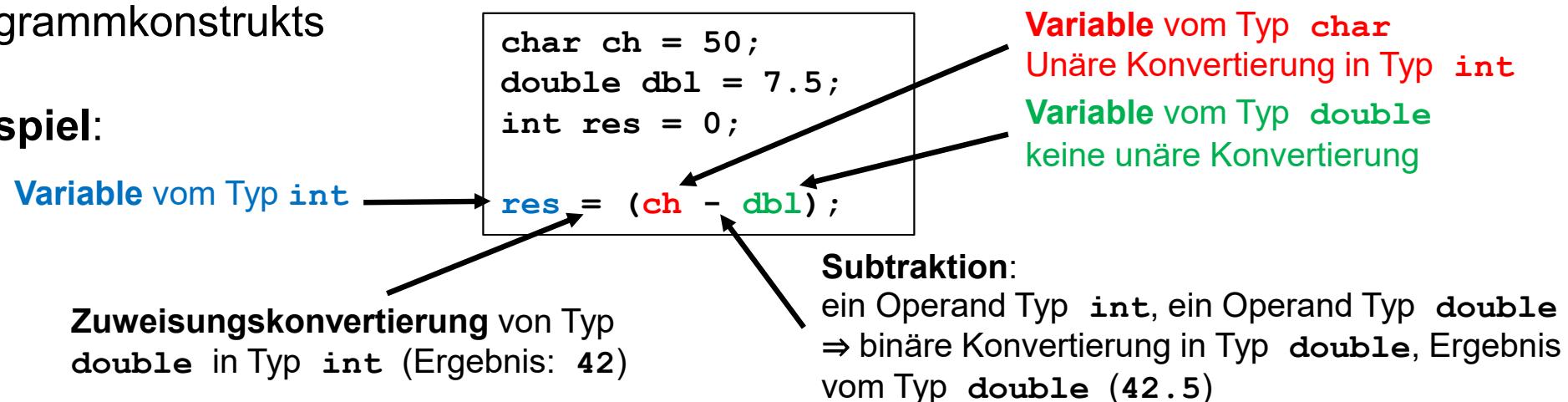
## Konvertierung (2)

- Was löst eine Konvertierung aus? Ausgehend von einem Ausdruck vom Typ **Q**, in welchen Typ **z** wird konvertiert? (Forts.)

- **Implizite binäre Konvertierungen:**

- Implizite Konvertierung der Operanden binärer Operatoren (**<AusdruckL> <BinOp> <AusdruckR>**)
- Anwendung erfolgt als Zusatzschritt nach Anwendung der impliziten unären Konvertierungen
- Festlegungen, in welchen **gemeinsamen** Typen **z** die beteiligten Ausdruckstypen  $O_L$  (**AusdruckL**) und  $O_R$  (**AusdruckR**) konvertiert werden
- Die eigentliche Operation hat anschließend **in der Regel** ein Ergebnis von Typ **z**
- Typischerweise Anwendung mehrerer Konvertierungen im Zuge eines Programmkonstrukts

### Beispiel:



# Konvertierungen zwischen Ganzzahl-Typen (1)

- Integraler Typ **Q** ⇒ Integraler Typ **Z**
  - Konvertierung immer erst zur Laufzeit
  - 1. Wert **w** ist in **Z** repräsentierbar: Wert wird verlustfrei übernommen

**Beispiel:** Konvertierung von 42

von	<b>short</b>	00000000 00101010
nach	<b>unsigned char</b>	00101010

**Beispiel:** Konvertierung von 42

von	<b>unsigned char</b>	00101010
nach	<b>short</b>	00000000 00101010

**Beispiel:** Konvertierung von -1

von	<b>short</b>	11111111 11111111
nach	<b>signed char</b>	11111111

**Beispiel:** Konvertierung von -42

von	<b>signed char</b>	11010110
nach	<b>short</b>	11111111 11010110

Annahmen  
**char** 8 bit  
**short** 16 bit  
 Zweierkomplement-Darstellung

## Konvertierungen zwischen Ganzzahl-Typen (2)

- Integraler Typ  $Q \Rightarrow$  Integraler Typ  $Z$

2. Wert  $w$  ist **negativ** und soll in einen größeren **unsigned**-Typ  $z$  konvertiert werden
  - Erweiterung von  $w$  auf die Größe von  $z$
  - Zu  $w$  wird dann die **größte repräsentierbare Zahl** von  $z + 1$  addiert  
(also  $+2^n$ , wobei  $n$  die Anzahl Wertbits von  $z$  ist)

**Beispiel:** Konvertierung von  $-42$

von <b>signed char</b>	11010110	
nach <b>unsigned short</b>	11111111 11010110	(-42) Erweiterung
	+	$2^n$ (65536 für 16bit)
	=	11111111 11010110 (65494)

3. Wert  $w$  ist **negativ** und soll in einen gleichgroßen **unsigned**-Typ  $z$  konvertiert werden
  - Zu  $w$  wird die **größte repräsentierbare Zahl** von  $z + 1$  addiert (also  $+2^n$ )

**Beispiel:** Konvertierung von  $-42$

von <b>signed char</b>	11010110 (-42)	
nach <b>unsigned char</b>	+ $2^n$ (256 für 8bit)	
	= 11010110 (214)	

Annahmen

char 8 bit	
short 16 bit	
Zweierkomplement-Darstellung	

## Konvertierungen zwischen Ganzzahl-Typen (3)

- Integraler Typ  $Q \Rightarrow$  Integraler Typ  $z$

4. Wert  $w$  soll in einen kleineren `unsigned`-Typ  $z$  konvertiert werden, in dem  $w$  nicht repräsentierbar ist
  - $w$  wird durch die **größte repräsentierbare Zahl** von  $z + 1$  dividiert (also durch  $2^n$ , wobei  $n$  die Anzahl Wertbits von  $z$  ist)
  - Konvertierungsergebnis ist dann der **positive Divisionsrest**

**Beispiel:** Konvertierung von -420

von <code>short</code>		11111110 01011100
nach <code>unsigned char</code>	%	$2^n$ (256 für 8bit)
	=	01011100 (92)

5. Wert  $w$  soll in einen `signed`-Typ  $z$  konvertiert werden, in dem  $w$  nicht repräsentierbar ist: **Überlauf, undefiniertes Ergebnis**

- **Aufzählungstyp  $Q \Leftrightarrow$  Integraler Typ  $z$**

- Aufzählungstypen werden als Ganzzahl-Typen repräsentiert, wobei die Wahl der Größe des Ganzzahl-Typen implementationsabhängig ist
- Es gelten also die dargestellten Regeln für Konvertierungen zwischen **integralen Typen**

Annahmen  
`char 8 bit`  
`short 16 bit`  
 Zweierkomplement-Darstellung

## Konvertierungen mit Zeiger-Typen (1)

- Zeigertyp  $Q \Rightarrow$  Integraler Typ  $z$ 
  - Zeiger  $w$  wird wie ein `unsigned`-Typ (in der Größe des Zeigertypen) behandelt, Konvertierung nach  $z$  dann nach den üblichen Regeln für integrale Typen
  - Konvertierung in hinreichend großen `unsigned`-Typ erhält also die binäre Repräsentation
  - **Sonderfall:** `NULL`-Zeiger muss grundsätzlich den Wert 0 ergeben
  - **Ansonsten unsicher, größter integraler Typ möglicherweise zu klein**
- Integraler Typ  $Q \Rightarrow$  Zeigertyp  $z$ 
  - Interpretation des Zeigertypen  $z$  als `unsigned`-Typ (in der Größe des Zeigertypen), Anwendung der Konvertierungsregeln für integrale Typen
  - Für den Wert 0 ergibt sich der `NULL`-Zeiger
  - **Ansonsten unsicher, nicht portabel**
- Zeigertyp  $Q \Rightarrow$  Zeigertyp  $z$ 
  - Grundsätzlich Konvertierung möglich
  - `NULL`-Zeiger darf beliebig zwischen verschiedenen Zeigertypen konvertiert werden und bleibt dabei der `NULL`-Zeiger
- Arraytyp  $Q \Rightarrow$  Zeigertyp  $z$ 
  - "Array von  $T$ " wird in "Zeiger auf  $T$ " konvertiert, Zeiger auf das erste Element des Feldes

## Sonstige Konvertierungen

---

- Beliebiger Typ  $Q \Rightarrow \text{void}$ -Typ
  - immer möglich, Ergebnis aber zu nichts zu gebrauchen
- **Unmöglich:** Konvertierung in einen Arraytypen  $z$
- **Aber:** Zeiger auf Arraytypen dürfen nach den Regeln für Zeigertypen konvertiert werden

# Überblick Konvertierungen (Erweiterung folgt...) Bei Klausur gegeben

Quelltyp Q	Zieltyp z	Anmerkungen
Integral (inkl. Aufzählungen)	Integral (inkl. Aufzählungen)	<ol style="list-style-type: none"> <li>Wert in z repräsentierbar: Konvertierung verlustfrei</li> <li>Wert in z <b>nicht</b> repräsentierbar, z ist <b>unsigned</b>, z <b>größer</b> als Q: Erweiterung auf Größe von z, Addition von <math>2^n</math></li> <li>Wert in z <b>nicht</b> repräsentierbar, z ist <b>unsigned</b>, z <b>gleich groß</b> wie Q: Addition von <math>2^n</math></li> <li>Wert in z <b>nicht</b> repräsentierbar, z ist <b>unsigned</b>, z <b>kleiner</b> als Q: pos. Divisionrest durch <math>2^n</math></li> <li>Wert in z <b>nicht</b> repräsentierbar, z ist <b>signed</b>: <b>undefiniert</b></li> </ol>
Zeiger auf A		Möglich, aber implementationsabhängig (nicht portabel)
Integral		Möglich, aber implementationsabhängig (nicht portabel)
Zeiger auf A	Zeiger auf B	
Array von A		Konvertierung in Zeiger auf erstes Element (B = A)
beliebig	void	Langweilig

- **Bis hierhin:** Welche Typen können prinzipiell in welche Typen konvertiert werden und was ist jeweils das Ergebnis?
  
- **Jetzt:** Wo werden Konvertierungen ausgelöst?
- **Implizite unäre Konvertierungen** für die Operanden bestimmter Operatoren
  - **Arrays:** Werte eines Typs "Array von **A**" werden implizit in einen "Zeiger auf **A**" konvertiert, der auf das erste Element des Arrays zeigt
  - **Integer Promotions:** Implizite Erweiterung von Werten integraler Typen
    - Integrale Typen haben eine **Rangfolge** (aufsteigend sortiert):

<code>char</code>	/ <code>unsigned char</code>	/ <code>signed char</code>
<code>short</code>	/ <code>unsigned short</code>	
<code>int</code>	/ <code>unsigned int</code>	
<code>long</code>	/ <code>unsigned long</code>	
    - Rang eines **Aufzählungstypen** entspricht dem Ganzahltypen, den die Implementierung zur Repräsentation nutzt

- **Integer Promotions (Forts.):**

- **Leitgedanken**

- Nach Konvertierung sollen Operanden nur noch den Rang von `int` oder `long` haben
    - Kleinere Typen werden dazu in den Typ `int` oder `unsigned int` erweitert
    - Informationsverlust wird immer vermieden, d.h. der mathematische Wert bleibt erhalten

Operanden-rang	...und...	Konvertierung in	Beispiel
<code>&gt;= int</code>		<b>keine Konvert.</b>	<code>long</code>
<code>&lt; int</code>	<code>ist signed</code>	<code>int</code>	<code>short</code>
<code>&lt; int</code>	<code>ist unsigned &amp; vollständig in int enthalten</code>	<code>int</code>	<code>unsigned short</code> (wenn <code>short</code> z.B. 16 Bit und <code>int</code> z.B. 32 Bit)
<code>&lt; int</code>	<code>ist unsigned &amp; nicht vollständig in int enthalten</code>	<code>unsigned int</code>	<code>unsigned short</code> (wenn <code>short</code> und <code>int</code> z.B. 32 Bit)

- Implizite Konvertierungen für die Operanden bestimmter **binärer** Operatoren
- **Zunächst** Anwendung der **unären Konvertierungen je Operand**, integrale Operanden haben dann min. den Rang `int` (also `int`, `unsigned int`, `long` oder `unsigned long`)
- Die Operanden werden dann in einen **gemeinsamen Typ** konvertiert, der typischerweise auch den Ergebnistyp der Operation darstellt
- **Leitgedanken**
  - Der gemeinsame Typ hat den **Rang** des größten beteiligten Operanden
  - **Ziel:** Informationsverluste **minimieren**, undefinierte Ergebnisse **ausschließen**
  - **Kein Problem:** Zwei `unsigned`- oder zwei `signed`-Operanden, der größere wird jeweils gemeinsamer Typ
  - **Problem:** Ein `signed`- und ein `unsigned`-Operand, einer muss konvertiert werden
    - Ohne Informationsverlust möglich, wenn der `unsigned`-Typ in den `signed`-Typ passt
    - Ansonsten `signed`-Typ in `unsigned`-Typ konvertieren (**Informationsverlust möglich**), da Konvertierung `unsigned` nach `signed` für manche Werte undefiniert

Reihenfolge der Regeln signifikant, erste passende Regel gilt:

ein Operand	anderer Operand	Gemeinsamer Typ	Beispiel
unsigned-Typ	unsigned-Typ	größerer der beiden unsigned-Typen	unsigned long + unsigned int => unsigned long
signed-Typ	signed-Typ	größerer der beiden signed-Typen	long + int => long
unsigned-Typ	signed-Typ von gleichem oder geringerem Rang	der unsigned-Typ	int unsigned int => unsigned int
unsigned-Typ	signed-Typ höheren Ranges, der den unsigned-Typen enthält	der signed-Typ	32bit 64bit => long
unsigned-Typ	signed-Typ höheren Ranges, der den unsigned-Typen nicht enthält	die unsigned-Variante des signed-Typen	32bit 32bit => unsigned long

wenn keine Regel zutrifft: keine Konvertierung

- Auswertung zu Wert und Typ des Literals

- **Zeichen-Literal:** Typ `int`, **kein lvalue**
- **Ganzzahl-Literal:** mindestens Typ `int`, **kein lvalue**

```
char ch = 'x';
int   i = 42;
```

- **String-Literal:**

- Wird durch ein statisches "Array von `char`" repräsentiert, das in Ausdrücken wie jedes Array in einen "Zeiger auf `char`" auf das erste Element konvertiert wird
- **Ausnahmen:** Wird als Operand des `&`- oder `sizeof`-Operators oder als rechte Seite einer Array-Initialisierung **nicht** in einen Zeiger konvertiert
- Dereferenzierung des entstehenden "Zeiger auf `char`" führt zwar zu einem **lvalue**, Schreiben in den Speicher des Arrays ist aber **unsicher**

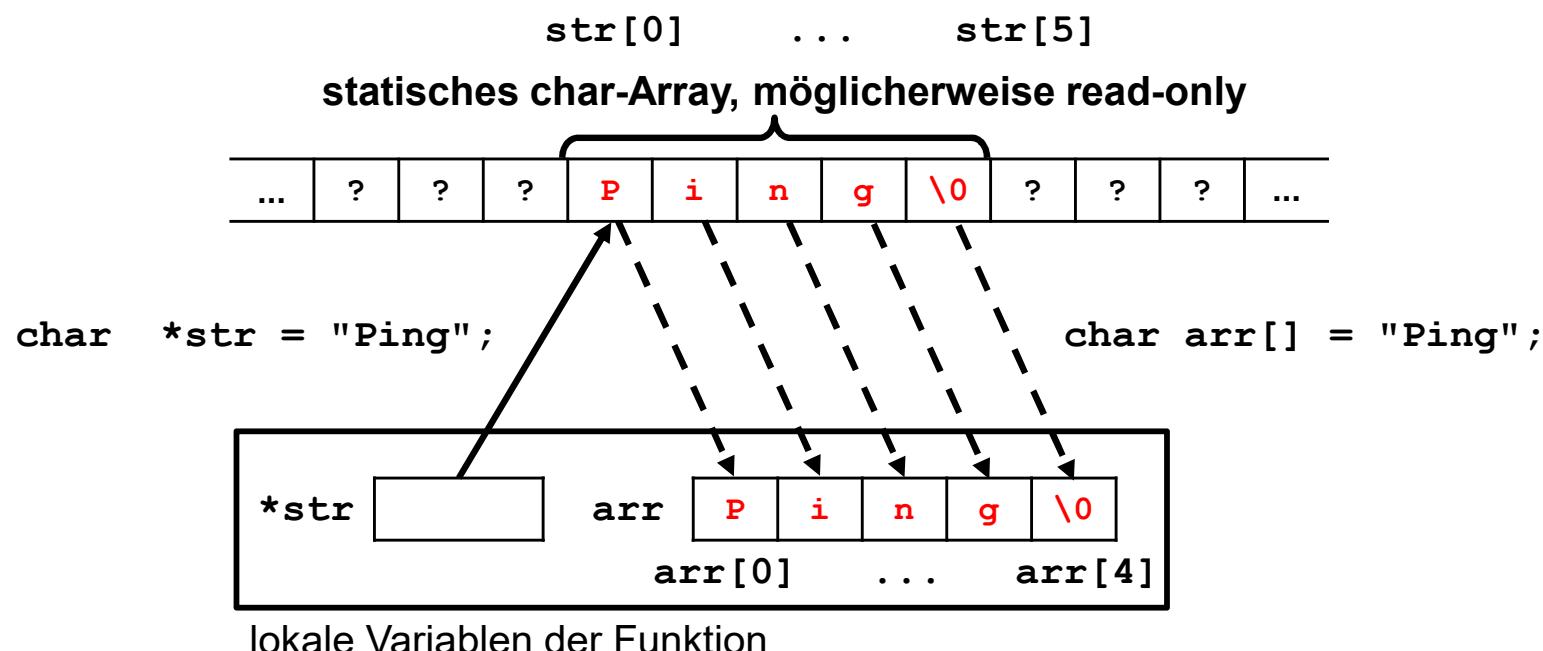
## Ausdrücke – Literale (Konstanten) (2)

- String-Literal (Forts.):

```

sizeof("Ping");      /* liefert 5, 4 Zeichen + 0-Symbol */
char arr[] = "Ping"; /* Initialisierung von arr (Kopie des statischen Arrays) */
char *str = "Ping";  /* Zuweisung Zeiger auf Speicherbereich der Konstante */
arr[1] = 'o';        /* Prima, arr enthält jetzt Zeichenfolge "Pong"
str[1] = 'o';        /* unsicher: Speicher möglicherweise read-only */
str[4] = '!';
str[5] = '!';
"Ping"[1] = 'o';    /* dieselbe gefährliche Semantik wie str[1] = 'o' */

```



- Auswertung zum Wert des mit dem Bezeichner verknüpften "Objektes" (**Variablen** oder **Aufzählungskonstanten**)
  - Bezeichner von **Variablen** werten zu **Ivalues** aus (im Falle eines Arrays **unmodifiable**)
  - Bezeichner von **Aufzählungskonstanten** werten zur entsprechenden Ganzahl-Konstanten vom Typ **int** aus und sind **keine Ivalues**

```
int x    = 42;  
int y    = x;
```

## Ausdrücke – Einfache Zuweisung (1)

$$\langle \text{AusdruckL} \rangle = \langle \text{AusdruckR} \rangle$$

- Speicherung des Wertes, zu dem **AusdruckR** auswertet, in dem von **AusdruckL** bezeichneten Objekt
- **Mögliche Zuweisungen:**
  - **AusdruckL** arithmetischer Typ     $\leq$    **AusdruckR** arithmetischer Typ
  - **AusdruckL** Zeiger auf **T**                 $\leq$    **AusdruckR** Zeiger auf **T**
  - **AusdruckL** Zeiger auf **T**                 $\leq$    **AusdruckR** NULL-Zeiger
  - ...

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung		Implizite Zuweisungskonvertierungen	
Typen (nach Konvertierung)	Beliebiger Typ <b>T</b> , <b>modifiable lvalue</b> (nicht: Qualifier <b>const</b> , Arrays)	Zuweisungskompatibler Typ <b>Q</b>	<b>T</b> , ist <b>kein lvalue</b>
Anmerkungen	<b>Auswertungsreihenfolge undefined</b>		

## Ausdrücke – Einfache Zuweisung (2)

- Implizite Zuweisungskonvertierungen (Erweiterung folgt...):

Quelltyp Q	Zieltyp Z	Anmerkungen
Integral (inkl. Aufzählungen)	Integral (inkl. Aufzählungen)	möglicherweise Informationsverlust oder undefiniertes Ergebnis
Zeiger auf A		Nicht zulässig
Integral	Zeiger auf B	Ausschließlich für die Zahl 0 ( <b>NULL</b> -Zeiger) zulässig
Zeiger auf A		Nur zulässig für <b>NULL</b> -Zeigerkonstante oder wenn A zu B kompatibel (typischerweise also identischer Zeigertyp)
Array von A		Konvertierung in Zeiger auf erstes Element (A = B)
beliebig	void	Nicht möglich, da Ausdruck L nie den Typ void hat

```
int          sint = ...;
unsigned char uchar = ...;
int          *iptr = ...;
unsigned char *ucptr = ...;
int          arr[10] = ...;
```

```
sint  = uchar;
uchar = sint;
sint  = iptr; /* Fehler */
iptr  = sint; /* Fehler */
iptr  = 0;
```

```
iptr  = &sint;
iptr  = &ucptr; /* Fehler */
iptr  = arr;
ucptr = arr;    /* Fehler */
```

- Nicht vorgesehene Konvertierungen können per **cast** erzwungen werden

( <Zieltyp> ) <Ausdruck>

- **Explizite** Konvertierung des Ergebnisses von **Ausdruck** in einen **skalaren Typen Zieltyp**
- Erlaubt die Anwendung jeder prinzipiell möglichen Konvertierung, einschließlich Änderung der Type-Qualifier (insbesondere Wegfall / Hinzufügen von **const**)

```

int                  i = 42;
unsigned int *uint_ptr = (unsigned int *)&i; /* nur mit cast zulässig */
const char          cchar = 'a';
char                *char_ptr = (char *)&cchar;      /* nur mit cast zulässig */
*char_ptr = 'b';                                /* const ausgehebelt */
  
```

	Zieltyp	Operand AusdruckR	Ergebnis
Konvertierung		Arraytyp: Implizite Konvertierung in Zeiger auf das erste Element	
Typen (nach Konvertierung)	Beliebiger skalarer Typ T	Beliebiger skalarer Typ Q	T, ist <b>kein lvalue</b>
	void	Beliebiger Typ Q	void

**<AusdruckL> <Op> <AusdruckR>**  
 mit <, <=, > oder >= als <Op>

- Relationaler Vergleich von **zwei Zahlen** oder von **zwei Zeigern**
- **Zeigervergleich:**  
Position zueinander im Speicher (z.B. innerhalb eines Arrays)

```

unsigned int val_null = 0;
unsigned int a = ..., b = ..., c = ...;
if( a < 42 ) { ... }
if( (a + b) >= (b + c) ) { ... }
if( -1 < val_null ) { ... }
  
```

```

char arr[] = "Gurkensalat";
char *ptr1 = &arr[0];
char *ptr2 = &arr[5];
if( ptr1 < ptr2 ) { ... }
  
```

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung	Implizite binäre Konvertierungen		
Typen (nach Konvertierung)	oder	Beliebiger arithmetischer Typ $\mathbf{T}$ Beliebiger Typ "Zeiger auf $\mathbf{T}$ "	$\mathbf{int}$ , ist <b>kein lvalue</b>
Anmerkungen	Auswertungsreihenfolge undefined		
			liefert Wert 0 ( <b>false</b> ) oder 1 ( <b>true</b> )

# Ausdrücke – Gleichheitstest (1)

$$\begin{aligned}<\text{AusdruckL}\rangle &== <\text{AusdruckR}\rangle \\<\text{AusdruckL}\rangle &!= <\text{AusdruckR}\rangle\end{aligned}$$

- Gleichheits- bzw. Ungleichheitstest von zwei **Zahlen oder** von zwei **Zeigern**

```
int a = ..., b = ..., c = ...;
if( a != 42 ) { ... }
if( (a + b) == (b + c) ) { ... }
```

```
int left = -1431655766;
unsigned int right = 2863311530;
if( left == right ) { ... };
```

```
char arr[] = "Gurkensalat";
char *ptr = &arr[0];

if( ptr == arr ) { ... }

if( ptr != NULL ) { ... }

↳ Zeiger darf mit jedem Zeiger verglichen werden
```

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung	Implizite binäre Konvertierungen		
Typen (nach Konvertierung)	oder	Beliebiger arithmetischer Typ T	int, ist <b>kein</b> lvalue
	oder	Beliebiger Typ "Zeiger auf T"	
Ein Operand von beliebigem Typ "Zeiger auf T", der andere ist die <b>NULL</b> -Zeigerkonstante			
Anmerkungen	Auswertungsreihenfolge undefined		
	liefert Wert 0 ( <b>false</b> ) oder 1 ( <b>true</b> )		

So 189%-Operationen allg. mit int-Werten

durchgeführt, deswegen

=> unsigned char cc = (42 << 8) >> 8 wieder 42.

- Linksshift unsicher: Auf VZ-Bit geschieben, Zahl

also negativ. Beim anschließendem Rechtsshift

werden Einsen "eingeschoben".

## Ausdrücke – Gleichheitstest (2)

- Klassische Fehlerquelle: Verarbeitung von **EOF**
  - **EOF (End of File): Negative Ganzzahl-Konstante (häufig -1) aus stdio.h**
  - Rückgabewert von Bibliotheksfunktionen, wenn das Dateiende **überschritten** wurde z.B. **int fgetc (FILE \*stream)** aus stdio.h

```
FILE      *file = fopen(..., "rb");
signed char ch = '\0';
while( (ch = fgetc(file)) != EOF ) { ... }
```

$\underbrace{(ch = fgetc(file))}_{-1 \text{ (signed char)}} \neq \underbrace{EOF}_{-1 \text{ (int)}}$

implizite unäre Konvertierung:  
 $-1 \text{ (int)}$        $-1 \text{ (int)}$

implizite binäre Konvertierung:  
 $-1 \text{ (int)} \neq -1 \text{ (int)}$

```
FILE      *file = fopen(..., "rb");
unsigned char ch = '\0';
while( (ch = fgetc(file)) != EOF ) { ... }
```

$\underbrace{(ch = fgetc(file))}_{255 \text{ (unsigned char)}} \neq \underbrace{EOF}_{-1 \text{ (int)}}$

implizite unäre Konvertierung:  
 $255 \text{ (int)}$        $-1 \text{ (int)}$

implizite binäre Konvertierung:  
 $255 \text{ (int)} \neq -1 \text{ (int)}$

- **Weiteres Problem:** Für Binärdateien wird der gesamte Wertebereich eines 8bit-**char** benötigt, **EOF** entspräche hier also einem "echten" Symbol
- **Lösung:** Zuweisung auf **int**-Variable

OR:

**<AusdruckL> || <AusdruckR>**

- Liefert 1 (**true**), wenn **mindestens ein Operand** zu  $\neq 0$  auswertet
- **Verkürzte Auswertung von links nach rechts:** Wertet **AusdruckL** zu  $\neq 0$  aus, entfällt die Auswertung von **AusdruckR** und das Ergebnis ist direkt 1 (**true**)

AND:

**<AusdruckL> && <AusdruckR>**

- Liefert 1 (**true**), wenn **beide Operanden** zu  $\neq 0$  auswertet
- **Verkürzte Auswertung von links nach rechts:** Wertet **AusdruckL** zu  $= 0$  aus, entfällt die Auswertung von **AusdruckR** und das Ergebnis ist direkt 0 (**false**)

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung	Implizite <b>unäre Konvertierungen je Operand</b>		
Typen (nach Konvertierung)	Beliebiger <b>skalarer</b> Typ T		<b>int</b> , ist <b>kein lvalue</b>
Anmerkungen	<b>Auswertung von links nach rechts</b> (AusdruckL vor AusdruckR), AusdruckR wird möglicherweise gar nicht ausgewertet		liefert Wert 0 ( <b>false</b> ) oder 1 ( <b>true</b> )

## ! <Ausdruck>

- `!x` ist äquivalent zu `x == 0`, liefert also gerade dann `1 (true)`, wenn der Operand zu `0` auswertet

```

int a = ..., b = ..., c = ...;
int idx = ...;
Error      err = ...;
if( (a >= 0) || (b >= 0) ) { ... }
if( (a >= 0) && (a <= 42) ) { ... }
if( !err && (idx < argc) ) { ... }
  
```

```

char *ptr = ...;

if( (ptr != NULL) && (*ptr != '\0') ) { ... }
/* äquivalent zu: */
if( ptr && *ptr ) { ... }

if( ptr == NULL ) { ... }
/* äquivalent zu: */
if( !ptr ) { ... }
  
```

	Operand Ausdruck	Ergebnis
Konvertierung	Implizite unäre Konvertierungen	
Typen (nach Konvertierung)	Beliebiger <b>skalarer</b> Typ <b>T</b>	<b>int</b> , ist <b>kein lvalue</b>
Anmerkungen		liefert Wert <b>0 (false)</b> oder <b>1 (true)</b>

# Ausdrücke – Additive binäre Operatoren (1)

Addition:

<AusdruckL> + <AusdruckR>

- Addition von Zahlen (Ganz- oder Fließkommazahlen)

```
int a = ..., b = ..., c = ...;
a = (2 * b) + c;
```

- Addition von Ganzzahlen zu Zeigern (**Zeigerarithmetik**)

```
int arr[5] = { 10, 20, 30, 40, 50 };
int *ptr = arr;
ptr = ptr + 2; /* *ptr == arr[2] == 30 */
ptr = 2 + ptr; /* *ptr == arr[4] == 50 */
```

	Operand AusdruckL	Operand AusdruckR	Ergebnis
<b>Konvertierung</b>	<b>Implizite binäre Konvertierungen</b>		
<b>Typen</b> (nach Konvertierung)	Beliebiger arithmetischer Typ <b>T</b>		<b>T</b> , ist <b>kein lvalue</b>
<u>oder</u>	Ein Operand von beliebigem Typ "Zeiger auf <b>T</b> ", der andere von <b>integarem Typ</b>		<b>Zeiger auf T</b> , ist <b>kein lvalue</b>
<b>Anmerkungen</b>	<b>Auswertungsreihenfolge undefined</b>		

Subtraktion:      **<AusdruckL>** – **<AusdruckR>**

- Subtraktion von Zahlen (Ganz- oder Fließkommazahlen)
- Subtraktion ganzer Zahlen von Zeigern (Zeigerarithmetik), Zahl immer **AusdruckR**
- Differenzberechnung zweier Zeiger  $d = p2 - p1$ , so dass  $p1 + d = p2$

```

int      arr[5] = { 10, 20, 30, 40, 50 };
int      *ptr = arr + 3;      /* *ptr = 40 */
ptrdiff_t d = ptr - arr;    /* d == 3, arr + d == ptr, arr[d] == *ptr */
  
```

*Aktuelles Arrayelement (Typ wichtig)*

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung	Implizite binäre Konvertierungen		
	Beliebiger arithmetischer Typ $T$		$T$ , ist <b>kein lvalue</b>
<u>Typen</u> (nach Konvertierung)	<u>oder</u> Beliebiger Typ "Zeiger auf $T$ "	Beliebiger integraler Typ	Zeiger auf $T$ , ist <b>kein lvalue</b>
	<u>oder</u> Beliebiger Typ "Zeiger auf $T$ "	ptrdiff_t (stddef.h), signed, ist <b>kein lvalue</b>	
Anmerkungen	Auswertungsreihenfolge undefined		

## Ausdrücke – Additive binäre Operatoren (3)

- Weiteres Beispiel

```

long l = 2000000000 + 2000000000; /* l ? */
fprintf(stdout, "l = %ld\n", l); /* l = -294967296 */

```

- Die Literale (Operanden der Addition) haben beide Typ `int`,  
Addition wird daher mit Ergebnistyp `int` durchgeführt  
=> **Überlauf, negativer Wert**
- Zuweisung auf `long` erweitert die Darstellung, die (falsche) Zahl bleibt unverändert
- **Korrektur:**

```

long l = 2000000000L + 2000000000; /* l ? */
fprintf(stdout, "l = %ld\n", l); /* l = 4000000000 */

```

- Das Suffix `L` zum Ganzzahl-Literal ändert seinen Typ auf `long`
- **Also:** Linker Operand Typ `long`, rechter Operand Typ `int`
- Entsprechend den Regeln für binäre Konvertierungen wird die Operation im größeren  
der beiden `signed`-Typen ausgeführt, also `long`  
=> **kein Überlauf**

## Ausdrücke – Multiplikative binäre Operatoren

Multiplikation:      **<AusdruckL> \* <AusdruckR>**

Division:                **<AusdruckL> / <AusdruckR>**

Modulo:                **<AusdruckL> % <AusdruckR>**

- **Multiplikation** und **Division** unterstützen die Ausführung sowohl für Ganzzahl- als auch für Fließkomma-Operanden, **Modulo** ist nur für die Ganzzahl-Division vorgesehen

```
int a = ...;
if( a % 2 == 0 ) { ... }
```

	Operand AusdruckL	Operand AusdruckR	Ergebnis
<b>Konvertierung</b>	<b>Implizite binäre Konvertierungen</b>		
<b>Typen (nach Konvertierung)</b>	* , / : Beliebiger arithmetischer Typ T % : Beliebiger integraler Typ T		T, ist <b>kein lvalue</b>
<b>Anmerkungen</b>	<b>Auswertungsreihenfolge undefined</b>		

- <Ausdruck>
- + <Ausdruck>

- + <Ausdruck> liefert einfach den Wert von Ausdruck
- - <Ausdruck> äquivalent zu 0 - <Ausdruck>

```
int i = - 42;      /* unäres -, angewandt auf Ganzzahl-Konstante */  
int j = - (i*i);  
int k = + 42;      /* unäres +, langweilig */
```

	Operand Ausdruck	Ergebnis
Konvertierung	Implizite unäre Konvertierungen	
Typen (nach Konvertierung)	Beliebiger arithmetischer Typ T	T, ist kein lvalue

**<Bezeichner> ( <tatsächliche Argumente\*> )**

- Aufruf der **Funktion**, zu der **Bezeichner** auswertet, mit den **Argumenten**, zu denen **tatsächliche Argumente** (Liste von Ausdrücken) auswerten
- Ergebnis des Ausdrucks ist der Rückgabewert des Funktionsaufrufs

```
int      add(int p, int q)          { ... }
char * concat(char *str1, char *str2) { ... }
...
int      a = add( 21, 21 );
int      b = add( a + 2 * 3, add(21, 21) );
char *ptr1 = concat( "Eisen", "bahn" );
char *ptr2 = concat( ptr1, "wagon" );
```

## Ausdrücke – Funktionsaufruf (2)

- Weitere Beispiele...

```
FILE *f = ...;
char *str = ...;
int i = 0;
int j = add(i++, i++);
```

```
i = 0;
j = add(++i, ++i);
i = 0;
if (++i < ++i) { ; }
```

```
str = concat( readLine(f), readLine(f) );
```

```
int add(int p, int q) { ... }
char * concat(char *str1, char *str2) { ... }
char * readLine(FILE *stream) { ... }
```

/\* p? q? i? j? \*/

p=1 q=0 i=2 j=1

/\* p? q? i? j? \*/

p=2 q=2 i=2 j=4

/\* true oder false? \*/

false

str1= Zeile3

str2= Zeile2

str= Zeile3 Zeile2

• Auswertungsreihenfolge  
der Argumente nicht definiert

	Argumente	Ergebnis
Konvertierung	Zuweisungskonvertierung der Argumente auf die Parameter der Funktion	Zuweisungskonvertierung von <code>return</code> -Operanden auf den Rückgabetyp der Funktion
Typen	Typ des Arguments muss zuweisbar auf Typ des Parameters sein	Rückgabetyp der Funktion, ist <b>kein lvalue</b>
Anmerkungen	<b>Auswertungsreihenfolge der Argumente undefined</b>	

## & <Ausdruck>

- Wertet zu einem Zeiger auf das Objekt aus, zu dem der **Ausdruck** auswertet

```

int      i = 42;
char     ch = 'a';
char *cptr = &ch; /* Zeiger auf char */
int *iptr = &i;   /* Zeiger auf int */
int **diptr = &iptr; /* Zeiger auf einen Zeiger auf int */
  
```

• Adressoperat auf einem Array liefert nicht d. Zeiger auf d. erste Element sondern auf Array selbst.

	Operand Ausdruck	Ergebnis
Konvertierung	Keine impliziten unären Konvertierungen, insbesondere werden Arrays <b>nicht</b> in Zeiger auf das erste Element konvertiert	
Typen	Beliebiger Typ $T$ , muss ein lvalue sein	Zeiger auf $T$ , ist <b>kein</b> lvalue

## \* <Ausdruck>

- Wertet zu dem **Ivalue** aus, auf den der Zeiger verweist, zu dem der **Ausdruck** auswertet

```
int *iptr = ...;      /* Zeiger auf int */
int **diptr = ...;    /* Zeiger auf einen Zeiger auf int */
int     i1 = *iptr;
int     i2 = *(diptr);
```

	Operand Ausdruck	Ergebnis
Konvertierung	Implizite unäre Konvertierungen, insbesondere werden Arrays in Zeiger auf das erste Element konvertiert	
Typen (nach Konvertierung)	Beliebiger Typ "Zeiger auf T"	T, ist ein Ivalue

## <AusdruckL> [ <AusdruckR> ]

- Berechnet einen Zeiger auf ein bestimmtes Element eines Feldes (angegeben per Index) und dereferenziert diesen, erlaubt also indizierten Zugriff auf Felder
- Äquivalent zu `* ( (<AusdruckL>) + (<AusdruckR>) )`

```

int arr[10] = { ... }; /* Array von int */
int *iptr = arr;      /* Zeiger auf int */
int i1 = arr[5];     /* Zugriff auf 6. Elementes von arr */
int i2 = iptr[5];    /* dto. */
int i3 = *(arr+5);  /* dto. */
int i4 = *(iptr+5); /* dto. */
arr[5] = 42;         /* Schreiben des 6. Elementes von arr */
*(arr+5) = 42;       /* dto., ebenso per iptr[5], *(iptr+5) */
  
```

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung	Implizite binäre Konvertierungen (siehe +)		
Typen (nach Konvertierung)	Ein Operand von beliebigem Typ "Zeiger auf T", der andere von integralem Typ		T, ist ein lvalue
Anmerkungen	Auswertungsreihenfolge undefined		

```
sizeof <Ausdruck>
sizeof ( <Typ> )
```

- Wertet zur Größe des Typen von **Ausdruck** bzw. des angegebenen Typen **Typ** aus (in Anzahl **char** bzw. Byte)

```
int i = 0;
sizeof (i)
sizeof (42)
sizeof ("Ping")
```

```
sizeof( int )
sizeof( enum { ERR_A, ERR_B, ERR_C } )
sizeof( int* )
sizeof( int[10] )
```

	<b>Operand Ausdruck</b>	<b>Ergebnis</b>
<b>Konvertierung</b>	<b>Keine impliziten Konvertierungen</b> , insbesondere werden Arrays nicht in Zeiger auf das erste Element konvertiert	In d. Lage jede Arraygröße zu repräsentieren
<b>Typen</b>	Beliebiger Typ <b>T</b>	<b>size_t</b> (unsigned-Typ, Größe implementationsabhängig), ist kein lvalue
<b>Anmerkungen</b>	Übersetzungszeit-Berechnung, <b>wird nicht ausgewertet</b>	<b>Konstant</b> , wird zur Übersetzungszeit berechnet

## Postfix Inkrement / Dekrement

<Ausdruck>++

<Ausdruck>--

## Prefix Inkrement / Dekrement

++<Ausdruck>

--<Ausdruck>

- **Postfix:** Liefert den Wert des Ausdrucks und inkrementiert bzw. dekrementiert den gespeicherten Wert **anschließend** um 1

```
int a = 0, b = 3, c = 6;
a = (c++) - (b--); /* a == 3, b == 2, c == 7 */
```

- **Prefix:** Inkrementiert bzw. dekrementiert den gespeicherten Wert um 1 und liefert den **neuen Wert** des Ausdrucks; äquivalent zu <Ausdruck> += 1 bzw. -= 1

```
int a = 0, b = 3, c = 6;
a = (++c) - (--b); /* a == 5, b == 2, c == 7 */
```

	Operand Ausdruck	impliziter konstanter Operand mit Wert 1	Ergebnis
Konvertierung	Implizite binäre Konvertierungen (siehe + und -), Zuweisungskonvertierung auf <Ausdruck>		
Typen (nach Konvertierung)	Beliebiger skalarer Typ T, modifiable lvalue		T, ist kein lvalue

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)
10. Strukturierte Datentypen
11. Konvertierungen
12. Ausdrücke 2: Vergleich / Logik / Arithmetik
13. Ausdrücke 3: Bit-Operationen

## Ausdrücke – Links-Shift (1)

**<AusdruckL> << <AusdruckR>**

- Shiftet die Bitrepräsentation der Zahl, zu der **AusdruckL** auswertet (*i*), um **AusdruckR** Bits (*n*) nach **links**
  - Verwirft die *n* **höchstwertigen** Bits (shiftet die höchstwertigen Bits nach links heraus)
  - Fügt *n* 0-Bits als **niedrigstwertige** Bits an (shiftet 0-Bits von rechts herein)
  - Entspricht einer **Multiplikation** von *i* \*  $2^n$

Variable vom Typ <b>unsigned char</b> (8bit)	
60 << 1	60 << 3
00111100 << 1	00111100 << 3
01111000 (120)	11100000 (224)

	60	<< 1	60	<< 3
	00111100	<< 1	00111100	<< 3
	<b>00111100</b>	<b>001</b>	<b>11100000</b>	<b>11100000</b>

10 << 1 == 20, 10 << 3 == 80, ...
-----------------------------------

Operand AusdruckL	Operand AusdruckR	Ergebnis
Konvertierung	Implizite <b>unäre</b> Konvertierungen	
Typen (nach Konvertierung)	Beliebiger <b>integraler</b> Typ <i>TL</i>	<i>TL</i> , ist <b>kein lvalue</b>
Anmerkungen	Auswertungsreihenfolge undefined	

## Ausdrücke – Links-Shift (2)

- **Undefiniert für  $n < 0$  (Shiften um eine negative Anzahl Bits)**

```
int shifted = 1 << -1; /* Undefiniert, nicht notwendigerweise Rechts-Shift */
```

- **Undefiniert für  $n \geq$  Bitbreite des Typs von AusdruckL (nach Konvertierung)**

```
int shifted = 1 << (sizeof(1 << 0) * CHAR_BIT); /* Undefiniert */
```

- **Beispiele: Bit auf bestimmte Position shiften**

```
unsigned char uc = 0;
signed char sc = 0;
int sint = 0;

uc = 1 << 0;           /* 00000001      (1) */
uc = 1 << 5;           /* 00100000      (32) */
uc = 1 << 7;           /* 10000000      (128) */
sc = 1 << 7;           /* 10000000 (-128) */
uc = 1 << 12;          /* 00000000      (0) */
sint = 1 << 12;         /* 00000000 00000000 00010000 00000000 (4096) */

/* Zusammengesetzte Zuweisung: i <= n ⇔ i = i << n */
/* Wiederholte Anwendung, um sukzessive eine Maske für jedes Bit zu erhalten */
uc = 1;
while( uc ) { ... uc <= 1; }
```

Annahmen  
 char 8bit  
 Zweierkomplement-Darstellung

# Ausdrücke – Rechts-Shift (1)

**<AusdruckL> >> <AusdruckR>**

- Shiftet die Bitrepräsentation der Zahl, zu der **AusdruckL** auswertet (*i*), um **AusdruckR** Bits (*n*) nach **rechts**
  - Verwirft die *n* **niedrigstwertigen** Bits (shiftet die niedrigstwertigen Bits nach rechts heraus)
  - **Höchstes bzw. Vorzeichen-Bit nicht gesetzt:** Fügt *n* 0-Bits als **höchstwertige** Bits vorne an (shiftet 0-Bits von links herein)
  - **signed-Typ und Vorzeichen-Bit gesetzt:** Fügt **implementationsabhängig** entweder *n* 0-Bits oder *n* 1-Bits als **höchstwertige** Bits vorne an (shiftet 0-Bits oder 1-Bits von links herein)

Variable vom Typ <b>signed char (8bit)</b>	
60 >> 1	60 >> 3
00111100 >> 1	00111100 >> 3
00011110 (30)	00000111 (7)
60 >> 1	60 >> 3
00111100 >> 1	00111100 >> 3
00011110 0	00000111 100
-68 >> 1 <b>oder</b>	-68 >> 1
10111100 >> 1	10111100 >> 1
01011110 0 (94)	11011110 0 (-34)

	Operand AusdruckL	Operand AusdruckR	Ergebnis
<b>Konvertierung</b>	<b>Implizite unäre Konvertierungen</b>		
<b>Typen (nach Konvertierung)</b>	Beliebiger <b>integraler Typ TL</b>	Beliebiger <b>integraler Typ TR</b>	<b>TL,</b> <b>ist kein lvalue</b>
<b>Anmerkungen</b>	<b>Auswertungsreihenfolge undefined</b>		

## Ausdrücke – Rechts-Shift (2)

- Entspricht einer **Division** von *i* durch  $2^n$

```
60 >> 1 == 30, 60 >> 2 == 15, 60 >> 3 == 7...
```

- Siehe **Links-Shift: Undefiniert für  $n < 0$  (Shiften um eine negative Anzahl Bits) und für  $n \geq$  Bitbreite des Typs von AusdruckL (nach Konvertierung)**
- Beispiele: Bit auf bestimmte Position shiften**

<pre> unsigned char uc = 0; signed char sc = 0; int      sint = 0;  uc = 128 &gt;&gt; 0;          /* 10000000  (128) */ uc = 128 &gt;&gt; 2;          /* 00100000  (32) */ uc = 128 &gt;&gt; 12;         /* 00000000  (0) */  sint = 128 &gt;&gt; 2;         /* 00000000 00000000 00000000 00100000  (32) */ sint = -128 &gt;&gt; 0;        /* 11111111 11111111 11111111 10000000 (-128) */ sint = -128 &gt;&gt; 2;        /* 11111111 11111111 11111111 11100000 (-32) */ /* oder */ /* 00111111 11111111 11111111 11100000 (1073741792) */  uc = (177 &lt;&lt; 1);        /* 10110001 &lt;&lt; 1 =&gt; 01100010 */ uc &gt;&gt;= 4;                 /* 01100010 &gt;&gt; 4 =&gt; 00000110 */ </pre>	<u>Annahmen</u> char 8bit Zweierkomplement-Darstellung
---	--

## Ausdrücke – Rechts-Shift (3)

- Rechtsshift bei **signed**-Typen mit Konvertierungen

```

int
sint = -128 >> 2;

signed char
sc = -128 >> 2;

unsigned char
uc = -128 >> 2;
  
```

	$-128 \gg 2$ (Ergebnistyp int) : <b>11111111 11111111 11111111 11100000</b> (-32) oder <b>00111111 11111111 11111111 11100000</b> (1073741792)			
OK	<b>11111111 11111111 11111111 11100000</b>			
OK	<b>00111111 11111111 11111111 11100000</b>			
OK	<b>11111111 11111111 11111111 <u>11100000</u></b>			
Überlauf	<b>00111111 11111111 11111111 <u>11100000</u></b>			
OK	<b>11111111 11111111 11111111 <u>11100000</u></b>			
OK	<b>00111111 11111111 11111111 <u>11100000</u></b>			

**<AusdruckL> & <AusdruckR>**

- **AND:** Ein Ergebnis-Bit ist gerade dann gesetzt, wenn **beide** entsprechenden Bits der Operanden-Werte gesetzt sind

**<AusdruckL> ^ <AusdruckR>**

- **XOR:** Ein Ergebnis-Bit ist gerade dann gesetzt, wenn **genau eines** der entsprechenden Bits der Operanden-Werte gesetzt ist

**<AusdruckL> | <AusdruckR>**

- **OR:** Ein Ergebnis-Bit ist gerade dann gesetzt, wenn **mindestens eines** der entsprechenden Bits der Operanden-Werte gesetzt ist

&	1010
	1100
	1000

^	1010
	1100
	0110

	1010
	1100
	1110

	Operand AusdruckL	Operand AusdruckR	Ergebnis
<b>Konvertierung</b>	<b>Implizite binäre Konvertierungen</b>		
<b>Typen (nach Konvertierung)</b>	Beliebiger <b>integraler Typ T</b>	Beliebiger <b>integraler Typ T</b>	T, ist <b>kein lvalue</b>
<b>Anmerkungen</b>	<b>Auswertungsreihenfolge undefined</b>		

# Ausdrücke – Bitweise Negation

## ~ <Ausdruck>

- Bitweise Negation des Operanden

Variable vom Typ <code>unsigned char</code> (8bit)	
~1	~60
~00000001	~00111100
11111110 (254)	11000011 (195)

Variable vom Typ <code>signed char</code> (8bit)	
~1	~(-1)
~00000001	~11111111
11111110 (-2)	00000000 (0)

	Operand Ausdruck	Ergebnis
Konvertierung	Implizite unäre Konvertierungen	
Typen (nach Konvertierung)	Beliebiger <b>integraler</b> Typ T	T, ist <b>kein lvalue</b>

- Aufgrund implementationsabhängiger Repräsentation negativer Zahlen (Zweierkomplement, Einerkomplement, Vorzeichen-Betrag)  
**für `signed`-Typen nicht portabel**
- Gilt für alle Bitoperationen (Shiften, bitweise Verknüpfungen und Negation)
- Beispiel mit Negation:

Variable vom Typ <code>signed char</code> (8bit)
<code>~1</code>
<code>~00000001</code>
<code>11111110</code> (-2 oder -1 oder -126?)

- **Empfehlung:** Nur für `unsigned`-Typen nutzen

- **Anwendungsbeispiel:** Bitweise Verarbeitungen per Bitmasken

- Konstruktion einer **Bitmaske** per Shift-Operator, in der ein bestimmtes Bit gesetzt ist:

`1 << (Index des Bits)`

```
unsigned char mask4 = 1 << 4; /* Bitmaske für Bit 4: 00010000 */  
unsigned char mask7 = 1 << 7; /* Bitmaske für Bit 7: 10000000 */
```

- **Auslesen** eines Bits: **AND**-Verknüpfung

```
unsigned char data = 202;  
if( data & mask7 ) { ... };  
if( data & mask4 ) { ... };
```

**11001010**  
**& 10000000**  
**10000000 (true)**

**11001010**  
**& 00010000**  
**00000000 (false)**

- **Setzen** eines Bits: **OR**-Verknüpfung

```
unsigned char data = 202;  
data |= mask7;  
data |= mask4;
```

**11001010**  
**| 10000000**  
**11001010**

**11001010**  
**| 00010000**  
**11011010**

- **Löschen** eines Bits: **AND**-Verknüpfung mit der **negierten** Bitmaske

```
unsigned char data = 202;  
data &= ~mask4;  
data &= ~mask7;
```

**11001010**  
**& 11101111**  
**11001010**

**11001010**  
**& 01111111**  
**01001010**

- **Anwendungsbeispiel:** Bitmasken mit mehreren Bits

```
/* Konstruktion einer 3bit-Maske: */  
unsigned char mask = 255 >> (8 - 3);  
unsigned char mask = UCHAR_MAX >> (CHAR_BIT - 3);
```

```
/* Selektion bestimmter Bits eines Datenwortes: */  
unsigned char data = (data >> 2) & mask;
```

11111111 >> 5  
00000**111**

???**xxx**?? >> 2 110**101**01 >> 2  
?????**xxx** 00110**101**  
& 00000**111** & 00000**111**  
00000**xxx** 00000**101**

- **Vorsicht bei boolescher Interpretation**, es muss nicht 1 herauskommen

```
int is_set(unsigned int value, unsigned int idx) {  
    return value & (1 << idx);  
}  
...  
if( is_set(..., 1) == 1 ) { ... }
```

- **Beispiel:** Textuelle Ausgabe einer Bitfolge

```
void printIntBinary(unsigned int val) {
    int idx = 0;
    for( idx = sizeof(val) * CHAR_BIT - 1 ; idx >= 0 ; idx-- ) {
        fprintf(stdout, "%s", (val & (1 << idx)) ? "1" : "0");
        fprintf(stdout, "%s", (idx % CHAR_BIT) ? "" : " ");
    }
}

void printIntBinary(unsigned int val) {
    unsigned int mask = 1 << (sizeof(val) * CHAR_BIT - 1);
    unsigned int space = 1 << ((sizeof(val) - 1) * CHAR_BIT - 1);
    for( ; mask ; mask >>= 1 ) {
        fprintf(stdout, "%s", (mask & space) ? (space >>= CHAR_BIT, " ") : "");
        fprintf(stdout, "%s", (val & mask) ? "1" : "0");
    }
}
```

## Beispiel: Bitmengen (1)

- **Idee:** Mengen ganzer Zahlen per Bitfolgen
  - Implementierung von Mengen, die ganze Zahlen aus einem (kleinen) Intervall der ganzen Zahlen enthalten können (z.B. { 0, 3, 8 } )
  - Jede Menge wird mit einem Datenwort repräsentiert, wobei jedes Bit des Datenwortes einer Zahl des Intervalls bzw. der Anwesenheit dieser Zahl in der Menge entspricht
  - Zuordnung in unserem Beispiel:  
Die ganze Zahl 0 entspricht dem Bit 0, die ganze Zahl 11 dem Bit 11, usw.
  - Speichereffizient: Für eine Menge über den ganzen Zahlen von 0 bis 31 genügt ein einzelnes 32bit-Wort (also typischerweise Typ `int`)
  - Die Erzeugung der Mengen, ihre Manipulation sowie Verknüpfung mit anderen Mengen werden auf Bitoperationen zurückgeführt
  - Definition eines Ganzzahlen-Typen zur Repräsentation der Mengen  
(Wertebereich 0 bis `sizeof(T) * CHAR_BIT - 1`)

```
typedef unsigned int Set;
typedef unsigned char Element;
```

## Beispiel: Bitmengen (2)

- Definition der **leeren Menge** als Ganzzahl 0 (kein Bit gesetzt)

```
Set emptyset() {
    return 0;
}
```

```
Set se = emptyset();
/* se == ... 0000000 */
```

- Definition der **einelementigen Menge** als Bitmaske

```
Set single(Element e) {
    return 1 << e;
}
```

```
Set s0 = single(0);
/* s0 == ... 00000001 */
Set s3 = single(3);
/* s3 == ... 00001000 */
```

- **Einfügen** eines Elementes in eine Menge

- Erzeugung einer einelementigen Menge (Bitmaske) für die einzufügende Zahl
- OR-Verknüpfung mit der bestehenden Menge (keine Änderung, wenn bereits enthalten)

```
Set insert(Set l, Element e) {
    return l | (1 << e);
}
```

```
Set s36 = insert( s3, 6 );
/*           ... 00001000 */
/*           | ... 01000000 */
/* s36      ... 01001000 */
```

## Beispiel: Bitmengen (3)

- Test auf **Vorhandensein** eines Elementes in einer Menge
  - Konstruktion einer korrespondierenden Bitmaske und AND-Verknüpfung

```
int contains(Set s, Element e) {
    return ((1 << e) & s) != 0;
}
```

```
if( contains(s36, 5) ) { ... };
/*     ... 01001000 */
/*     & ... 00100000 */
/*     ... 00000000 */
```

- Bildung der **Vereinigungsmenge**
  - OR-Verknüpfung

```
Set unite(Set l, Set r) {
    return l | r;
}
```

```
Set s35 = insert( s3, 5 );
Set s356 = unite( s36, s35 );
/* s36     ... 01001000 */
/* s35     | ... 00101000 */
/* s356    ... 01101000 */
```

- Bildung der **Schnittmenge**
  - AND-Verknüpfung

```
Set intersect(Set l, Set r) {
    return l & r;
}
```

```
Set is = intersect( s36, s35 );
/* s36     ... 01001000 */
/* s35     & ... 00101000 */
/* s356    ... 00001000 */
```

## Beispiel: Bitmengen (4)

- **Nutzungsbeispiel von Bitmengen:** Fehlercodes akkumulieren

```
typedef enum { ERR_1, ERR_2, ERR_3, ERR_4 } Error;
Set errors = emptyset();

...
    errors = insert(errors, ERR_2);
...
    errors = insert(errors, ERR_4);
...

if( errors ) {
    fprintf(stdout, "Fehler aufgetreten!\n");
}

...
if( contains(errors, ERR_2) ) {
    fprintf(stdout, "Fehler ERR_2 aufgetreten!\n");
}

...
if( intersect(errors, unite( single(ERR_2), single(ERR3) ) ) {
    fprintf(stdout, "Fehler ERR_2 oder ERR_3 aufgetreten!\n");
}

...
```

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)
10. Strukturierte Datentypen
11. Ausdrücke 2: Konvertierungen, Vergleich / Logik / Arithmetik
12. Ausdrücke 3: Bit-Operationen
13. C-Präprozessor

- Kompilieren:** Quellcode-Dateien werden jeweils in Maschinencode und davon ausgehend in binäre Objektdateien übersetzt

```

Set emptyset() {
  ...
}

Set insert(Set s, Element e) {
  ...
}
...
  
```

Set.c

*Objektdatei: Maschinenbefehle*

```

Set emptyset():
100401293: push    %rbp
100401294: mov     %rsp,%rbp
100401297: ...

Set insert(Set s, Element e):
1004012f9: push    %rbp
1004012fa: mov     %rsp,%rbp
1004012fd: ...
  
```

Set.o

```

Set emptyset();
Set insert(Set, Element);
static int foo() { ... };
int main(...){
  ...
  foo(); ~ Im Hauptprogramm selbst implementiert
  ...
  insert(s, e);
  ...
}
  
```

MyProg.c

*int foo(): Keine Adresse*

```

int main(...):
1004013a5: push    %rbp
1004013a6: push    %rbx
1004013a7: ...
10040140a: callq   0x100401360
10040140f:
100401429: callq   <insert>
  
```

MyProg.o

## ▪ **Funktionsprototypen** (*Nur Signatur*)

- Anstelle einer Funktions**definition** können auch lediglich Funktions**deklarationen** ohne Rumpf angegeben werden (**Prototypen**)
- Ein Prototyp deklariert, dass **im Kompilat** eine Funktion des angegebenen Namens **existieren wird**, die die angegebene Signatur (Rückgabetyp, Parametertypen) aufweist
- Ermöglicht die Verwendung von Funktionen, die in anderen Quelldateien definiert werden oder bereits in kompilierter Form (Bibliotheken) vorliegen
- **Außerdem:** Ermöglicht die Verwendung von Funktionen (insbesondere für Funktionsaufrufe), **bevor die eigentliche Definition im Quelltext erscheint**
- Prototypen müssen keine Parameterbezeichner angeben, Typen genügen
- **Übliche Praxis:** Prototypen zu allen Funktionen, die in einem Modul genutzt werden, am Anfang der Quellcode-Datei

```
Set emptyset();  
Set insert(Set, Element);  
static int foo() { ... };  
int main(...) {  
    ...  
    foo();  
    ...  
    insert(s, e);  
    ...  
}
```

MyProg.c

## Übersetzungsprozess (3)

- Werden Funktionen genutzt, die **in derselben Quellcodedatei** definiert werden, so werden die entsprechenden Adressen direkt berechnet
- Beispiel:** `foo()` wurde in der Quellcode-Datei `MyProg.c` definiert, die Adresse für den Sprung zu `foo()` kann bereits aufgelöst werden
- Werden Funktionen genutzt, die **nicht in der derselben Quellcodedatei** definiert werden, so enthält die entsprechende Objektdatei nur symbolische Referenzen
- Beispiel:** `insert()` wurde in der Quellcode-Datei `MyProg.c` nicht definiert, die Adresse für den Sprung zu `insert()` kann noch nicht aufgelöst werden

```

Set emptyset();
Set insert(Set, Element);
static int foo() { ... };
int main(...) {
  ...
  foo();
  ...
  insert(s, e);
  ...
}
  
```

MyProg.c

```

int foo():
100401360: push    %rbp
100401363: ...
      int main(...):
1004013a5: push    %rbp
1004013a6: push    %rbx
1004013a7: ...
10040140a: callq   0x100401360
10040140f:
100401429: callq   <insert>
  
```

MyProg.o

## Bindung und Speicherklassen

- Speicherklasse **extern**: Statische Lebensdauer mit externer Bindung
- Speicherklasse **static**: Statische Lebensdauer mit interner Bindung

*↳ Nur bei lokalen Fkt. verw. -> äquivalent zu private in Java.*

- **Extern** gelinkte Funktionen **mit Definition** (d.h. **mit Funktionsrumpf**) werden anderen Übersetzungseinheiten bereitgestellt

z.B. **insert (Set, Element)** in **Set.c**

- **Extern** gelinkte Funktionen **ohne Definition** (d.h. **ohne Funktionsrumpf**) erfordern, dass eine andere Übersetzungseinheit die Definition bereitstellt

z.B. **insert (Set, Element)** in **MyProg.c**

- **Intern** gebundene Funktionen müssen **innerhalb ihrer Übersetzungseinheit** definiert werden, außerdem werden sie anderen Übersetzungseinheiten **nicht bereitgestellt**

z.B. **foo ()** in **MyProg.c**

*Ganze Programm laufzeit*

*Standard Speicherklasse*

```
extern
Set insert(Set s, Element e) {
    ...
}
```

**Set.c**

```
extern
Set insert(Set, Element);
```

```
static
int foo() { ... };
```

```
int main(...) {
    ...
    foo();
    ...
    insert(s, e);
    ...
}
```

**MyProg.c**

### ▪ Bindung und Speicherklassen (Forts.)

- Differenzierung externer / interner Bindung auch für **Top-Level (globale) Variablen**
- **Extern** gelinkte Variablen **mit Initializer** werden anderen Übersetzungseinheiten bereitgestellt  
**z.B. counter in Set.c**
- **Extern** gelinkte Variablen **ohne Initializer** erfordern, dass eine andere Übersetzungseinheit die Variable nebst Initialisierung bereitstellt  
**z.B. counter in MyProg.c**
- **Intern** gebundene Variablen müssen **innerhalb ihrer Übersetzungseinheit** initialisiert werden (gegebenenfalls per **Default-Initialisierung**), außerdem werden sie anderen Übersetzungseinheiten nicht bereitgestellt  
**z.B. flag in MyProg.c**

```
extern  
int counter = 42;  
...
```

Set.c

```
extern  
int counter;
```

} Var. existiert in anderer Datei, wird nur referenziert.

```
static ← interne Bindung  
int flag = 1;
```

```
int main(...) {  
    ...  
    counter++;  
    ...  
    flag = !flag;  
    ...  
}
```

MyProg.c

- Bindung und Speicherklassen (Forts.)
  - Zusammenfassung der möglichen Import / Export-Varianten

```
/* Definition in der vorliegenden Übersetzungseinheit, kein Import/Export */
static int v;
static int foo();           /* Prototyp, Definition später in der Datei */
static int foo() { ... }

/* Definition in der vorliegenden Übersetzungseinheit, Export */
extern int v = 0;
extern int foo();           /* Prototyp, Definition später in der Datei */
extern int foo() { ... }

/* Nur Deklaration in der vorliegenden Übersetzungseinheit, Import */
extern int v;
extern int foo();           /* Prototyp, keine Definition später in der Datei */
```

- **extern** ist Standard-Speicherklasse für **Funktionen** und **Top-Level-Variablen**:

int foo();	entspricht (weitgehend)	extern int foo();
------------	-------------------------	-------------------

Zwischen **extern** und dem Weglassen der Speicherklasse (**extern int v** / **int v**) gibt es allerdings feine Unterschiede, die wir hier nicht thematisieren

## Übersetzungsprozess (7)

- **Import-Deklarationen** können auch direkt in Blöcken erfolgen (eingeschränkte Sichtbarkeit), die Standard-Speicherklasse für Variablen ist dann allerdings **auto**  
=> Angabe von **extern** zwingend notwendig
- **Linken:** Mehrere Objektdateien werden zu einer Datei zusammengebunden
  - Zusammenfassung aller Inhalte in einer Datei
  - Auflösung von Symbolen, d.h. Ersetzen durch feste Adressen
  - Für eine ausführbare Datei muss genau eine **main()**-Funktion vorhanden sein
  - alternativ Erzeugung von Bibliotheken (also ohne **main()**-Funktion)

```
int main(...) {
    extern int v;
    extern int foo();
    ...
}
```

• Kann nur in außerer Fkt. verwendet werden.  
• Am besten vermeiden & Header einbinden

### MyProg.exe

```
int foo():
100401360: push    %rbp
100401363: ...
int main(...):
1004013a5: push    %rbp
1004013a6: push    %rbx
1004013a7: ...
10040140a: callq   0x100401360
10040140f:
100401429: callq   0x1004012f9
    Set emptyset():
100401293: push    %rbp
100401294: ...
    Set insert(Set s, Element e):
1004012f9: push    %rbp
1004012fa: ...
```

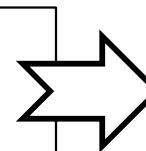
- Verarbeitet zu Beginn der Übersetzung die Quelldateien (\*.c) und führt definierte Textersetzung vor der eigentlichen Kompilation aus
  - Einkopieren von Header-Dateien (Funktionsprototypen usw.)
  - Nutzung von Namen für konstante Werte
  - Bedingte Kompilation, d.h. Entfernen / Einschluß von Quellcode-Teilen aufgrund von Bedingungen (z.B. Debugging ein/aus)
  - usw.
- Anweisungen an den Präprozessor per **Präprozessor-Direktiven**
  - Einleitung einer Direktive per #
  - **#define:** Definition von sogenannten Makros
  - **#include:** Einbinden von anderen (Quellcode-)Dateien
  - **#if, #else, #endif, #ifdef, #ifndef:**  
Bedingtes Einschließen von Codeabschnitten
  - ...

- **#define**: Definition eines Makros, dessen Vorkommen im Quelltext jeweils durch eine definierte Zeichenfolge ersetzt werden sollen (**Makro-Expansion**)

- **#define <Name> <Zeichen\*>**

Zuordnung eines Namens zu einer Zeichenfolge (**Rumpf des Makros**)

```
#define ARRAY_SIZE 5
...
int arr[ARRAY_SIZE];
```



```
...  
int arr[5];
```

- **#define <Name>(<Parameterbezeichner\*>) <Zeichen\*>**

Nutzung von Parametern im Rahmen der Ersetzung, jedes Vorkommen eines Parameterbezeichners im Rumpf wird durch den tatsächlichen Parameter ersetzt

```
#define IS_MULTIPLE_OF(NUMBER, FACTOR) ((NUMBER) % (FACTOR) == 0)
...
int i = ...;
if( IS_MULTIPLE_OF(i, 2) ) { ... }
```



```
...
int i = ...;
if( ((i) % (2) == 0) ) { ... }
```

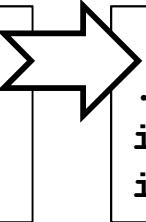
- Der Rumpf einer Makrodefinition darf andere Makros enthalten, nicht jedoch das definierende Makro selbst (keine Rekursion)

```
#define FAC(i) ((i) == 0 ? 1 : FAC((i)-1) * (i)) /* Fehler */
```

- Achtung: Makros sind reine Textersetzung

- Operatorenbindung:** Nach Expansion möglicherweise geänderte Verhältnisse

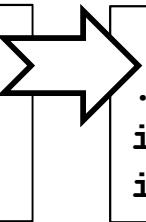
```
#define SQUARE(i) i * i
...
int a = 5;
int b = 2 * SQUARE(a + 1);
```



```
...
int a = 5;
int b = 2 * a + 1 * a + 1; /* 16 statt 38 */
```

- Daher wichtig:** Scheinbar unnötige Klammerungen, die unbeabsichtigte Wechselwirkungen mit dem Ausdruck, in dem das Makro erscheint, vermeiden

```
#define SQUARE(i) ((i) * (i))
...
int a = 5;
int b = 2 * SQUARE(a + 1);
```



```
...
int a = 5;
int b = 2 * ((a + 1) * (a + 1));
```

- **Achtung:** Makros sind reine Textersetzung (Forts.)
  - **Mehrfachauswertung:** Parameter werden nicht vor der Expansion ausgewertet

```
#define SQUARE(i) ((i) * (i))  
...  
int a = 5;  
int b = 2 * SQUARE(a + 1);
```

```
...  
int a = 5;  
int b = 2 * ((a + 1) * (a + 1));  
/* a+1 wird zweimal ausgewertet */
```

```
int square(int i) {  
    return i * i;  
}  
...  
int a = 5;  
int b = 2 * square(a + 1);
```

```
...  
int a = 5;  
int b = 2 * square(6);  
/* a+1 wird nur einmal ausgewertet */
```

- Vorsicht mit **intuitiver Syntax:** Präprozessor-Direktiven sind **kein C-Code**

```
#define ARRAY_SIZE = 5;  
...  
int arr[ARRAY_SIZE];
```

```
...  
int arr[= 5;];
```

# Exkurs: Ablauf von Funktionsaufrufen (1)

```

static int c = 0;
...
int add(int l, int r) {
    return l + r;
}

int sum(int f, int t) {
    c++;
    if(f == t) {
        return f;
    } else {
        int s = sum(f, t-1);
        return add(s, t);
    }
}
...

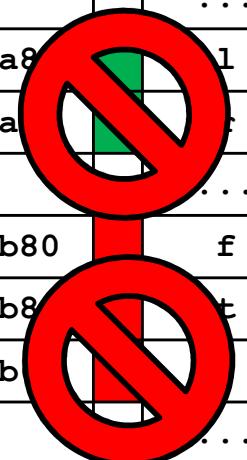
```

- Speicherorganisation zur Laufzeit (1. Versuch)
  - Kompilierter Programmcode von `add()` und `sum()`
  - Statische Variablen
  - Parameter und lokale Variablen von `add()` und `sum()`

Instruktionen für <code>add()</code>	0x15a5	add()
		...
Instruktionen zu <code>sum()</code>	0x160b	sum()
		...
statische Variablen	0x7000	c
		...
Variablen für <code>add()</code>	0xca80	l
	0xca88	r
		...
Variablen für <code>sum()</code>	0xcb80	f
	0xcb88	t
	0xcb6c	s
		...

## Exkurs: Ablauf von Funktionsaufrufen (2)

- **Problem:** Funktionen sollen sich selbst aufrufen können (Rekursion)
  - Zu allen Parametern und lokalen Variablen ist daher **je Aufruf** eigener Speicherplatz erforderlich, d.h. mehrere "Instanzen" können nebeneinander im Speicher existieren
- **Lösung:** Zur Laufzeit **Stapelspeicher** für die Zustände von Funktionsaufrufen
  - Jedes Element auf dem Aufruf-Stapel ist ein **Stack-Frame** zu einer bestimmten Funktion
  - Ein Stack-Frame bietet Platz für alle Parameter und lokalen Variablen der entsprechenden Funktion
  - Ein **Funktionsaufruf** erzeugt einen neuen Stack-Frame auf dem Stapel, der oberste Stack-Frame gehört also zum jüngsten begonnenen Funktionsaufruf
  - **return**-Anweisungen entfernen den jeweils obersten Stack-Frame
  - Implementiert Speicherklasse **auto** von Variablen (im Gegensatz zu **static**)



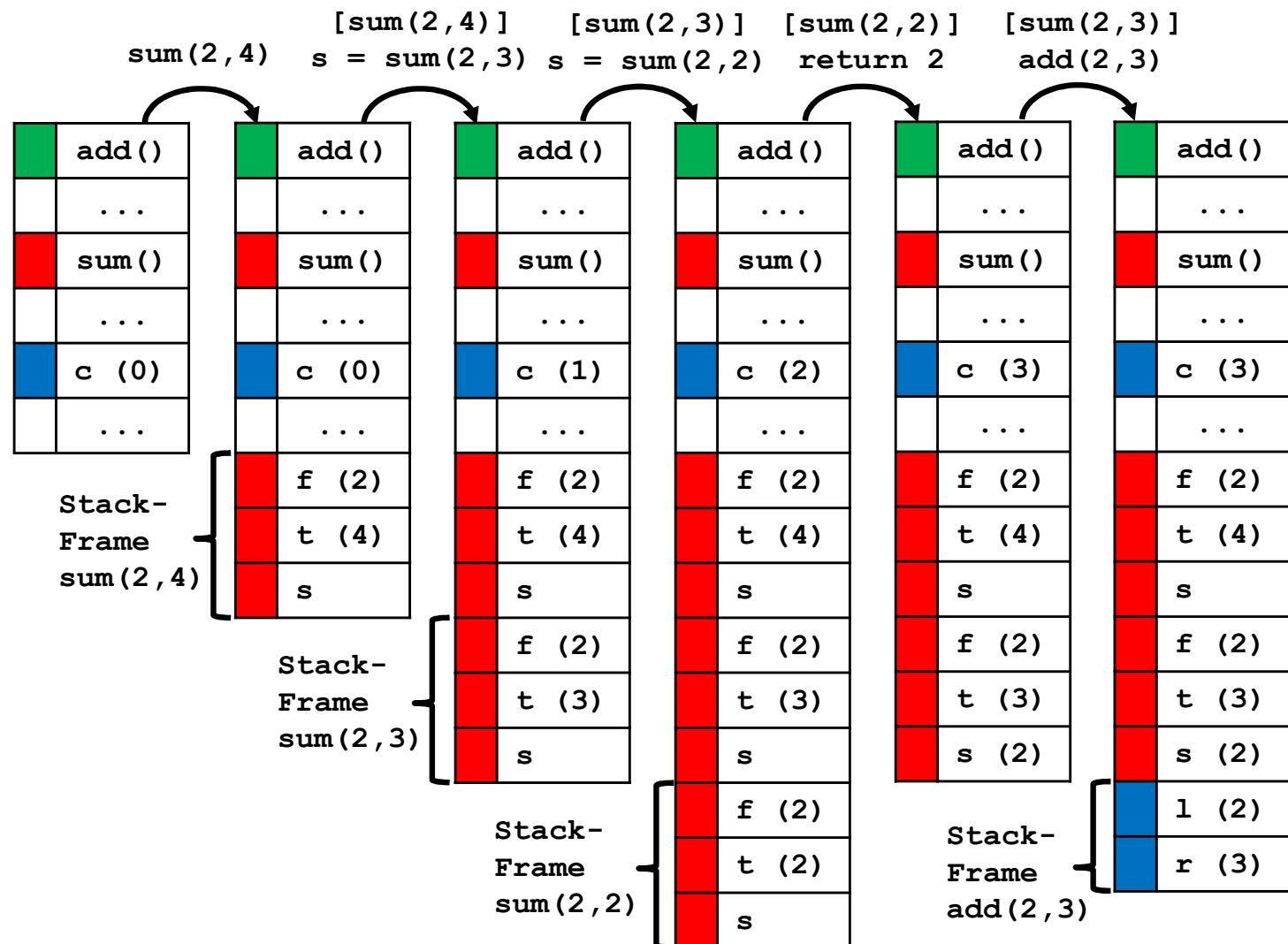
0x15a5	green	add()
		...
0x160b	red	sum()
		...
0x7000	blue	c
		...
0xca8	green	l
0xca	green	t
		...
0xcb80	green	f
0xcb8	green	t
0xcb	green	...

## Exkurs: Ablauf von Funktionsaufrufen (3)

```

static int c = 0;
...
int add(int l, int r) {
  return l + r;
}

int sum(int f, int t) {
  c++;
  if(f == t) {
    return f;
  } else {
    int s = sum(f, t-1);
    return add(s, t);
  }
}
...
  
```

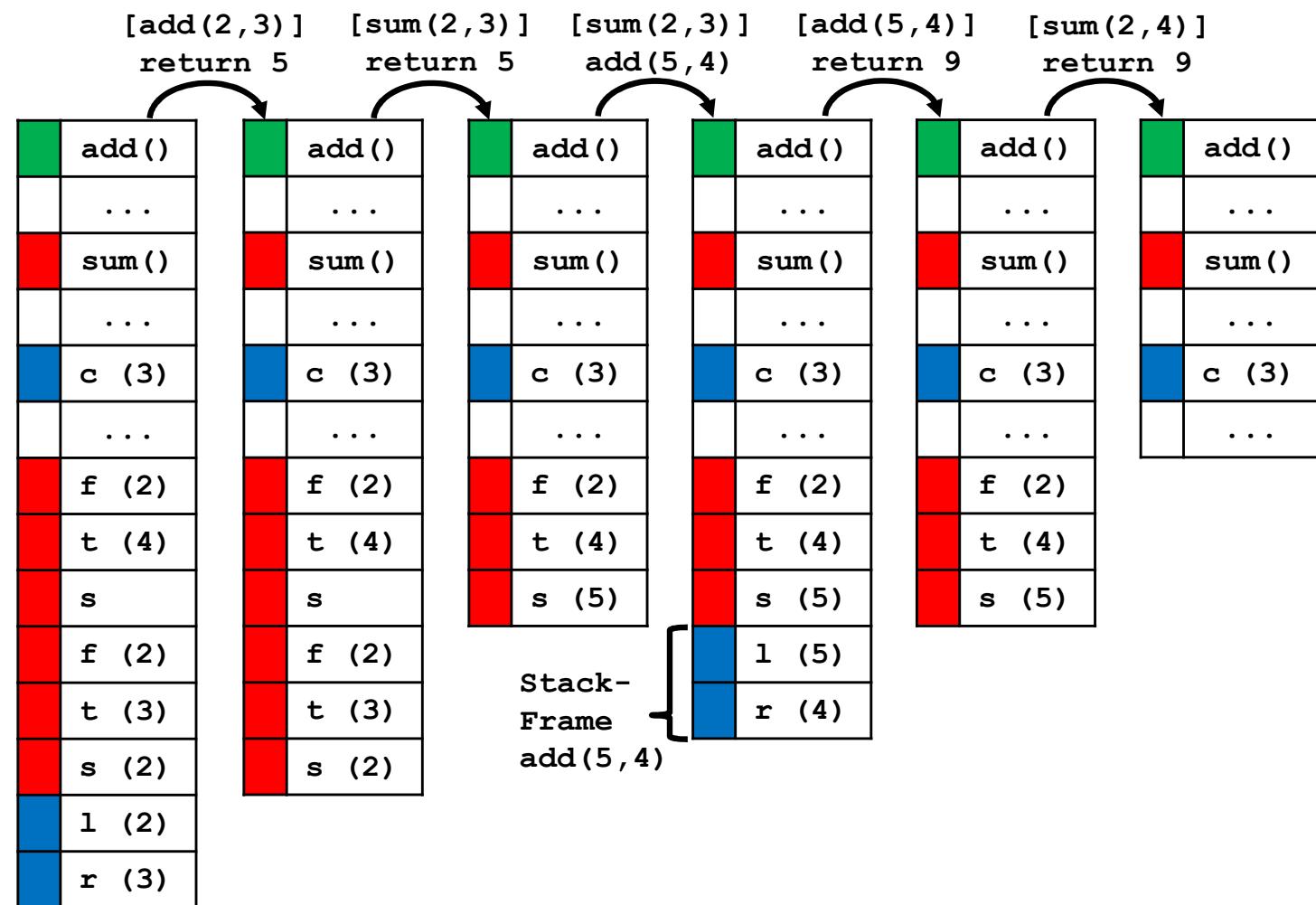


## Exkurs: Ablauf von Funktionsaufrufen (4)

```

static int c = 0;
...
int add(int l, int r) {
  return l + r;
}

int sum(int f, int t) {
  c++;
  if(f == t) {
    return f;
  } else {
    int s = sum(f, t-1);
    return add(s, t);
  }
}
...
  
```



- Ein wichtiger Treiber hinter Makros:  
**Vermeidung von Funktionsaufrufen**
- Variante der Bitmengen als Makros (**Set.h**)

```
typedef unsigned int Set;
typedef unsigned char Element;

#define emptyset()          ( 0 )
#define single(e)           ( 1 << (e) )
#define insert(s,e)         ( (s) | single(e) )
#define contains(s,e)       ( (s) & single(e) )
#define union(l,r)          ( (l) | (r) )
#define intersect(l,r)      ( (l) & (r) )
#define diff(l,r)           ( (l) & (~(r)) )
```

- Funktionsaufruf ohne Optimierung  
do Compilers relativ teuer → daher Präprozessorkonstanten / Makros
- Rekursion mit Makrodefinitionen nicht möglich.

- **#include:** Einbinden einer Text-Datei (Direktive durch Dateiinhalt ersetzen)

- **#include < <Dateiname> >**

Einbinden der bezeichneten Datei, die in implementationsspezifischen Pfaden gesucht wird (z.B. meistens im **Standardverzeichnis für Bibliotheksheader**) (*kann aber auch andere Dateien enthalten*)

```
#include <stdio.h>
...
int main(...) {
    ...
    fprintf(...);
    ...
}
```

Konvention: Suche nach Standardbibliotheken im normalen Systempfad

- **Typische Verwendung:** Systembibliotheken bzw. deren Funktionsprototypen, Variablen und Konstanten / Präprozessor-Direktiven usw. einfügen

- **#include " <Dateiname> "**

Einbinden der bezeichneten Datei, die in implementationsspezifischen Pfaden gesucht wird (z.B. meistens **im aktuellen Verzeichnis**)

```
#include "set.h"
...
int main(...) {
    ...
    insert(...);
    ...
}
```

Konvention: lokales Verzeichnis für eigene Header Dateien

- **Typische Verwendung:** Einbinden der Funktionsprototypen, Variablen und Konstanten / Präprozessor-Direktiven usw. eigener Module

- **#ifdef, #ifndef:** Bedingtes Einbinden von Quellcode
  - Prüfung (ausgehend vom Namen), ob ein Makro bereits zuvor im Quelltext definiert (**#ifdef**) bzw. nicht definiert wurde (**#ifndef**)
  - Bei erfolgreicher Prüfung wird der Text bis zum folgenden **#endif** in die Übersetzung eingeschlossen
  - **Typische Verwendung:** Konstante mit einem Standardwert definieren, wenn sie nicht zuvor durch den Compiler definiert wurde (Redefinitionen sind unzulässig)

```
#ifndef MAX_ARRAY_LENGTH
#define MAX_ARRAY_LENGTH 10
#endif
```

- **Typische Verwendung:** Doppelte Einbindungen verhindern, Header-Dateien können mittelbar mehrfach eingefügt werden und führen dann zu Konflikten  
Z.B. könnten sowohl **set.h** als auch **list.h** eine **helper.h** einbinden:

```
#include "set.h"
#include "list.h"
...
```

```
#ifndef __HELPER_H__
#define __HELPER_H__
...
#endif
```

- **#ifdef, #ifndef**: Bedingtes Einbinden von Quellcode (Forts.)
  - **Typische Verwendung**: Ausschaltbare Debug-Ausgaben

```
#ifdef DEBUG
    fprintf(stderr, "DEBUG set.insert() #1 ...", ...);
#endif
```

- **#if, #else, #endif**: Bedingtes Einbinden von Quellcode
  - Prüfung, ob ein gegebener konstanter arithmetischer Ausdruck zu `!= 0` auswertet
  - Bei erfolgreicher Prüfung wird der Text bis zum folgenden `#else` bzw. `#endif` in die Übersetzung eingeschlossen

```
#if (ARRAY_SIZE > 5)
...
#else
...
#endif
```

• *Leerer String für undefinierte Präprozessorkonstanten*

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)
10. Strukturierte Datentypen
11. Ausdrücke 2: Konvertierungen, Vergleich / Logik / Arithmetik
12. Ausdrücke 3: Bit-Operationen
13. C-Präprozessor
- 14. Dynamische Speicherverwaltung**

- Bis hierhin nur statische Speicherverwaltung (nicht zu verwechseln mit statischer Lebensdauer von Objekten)
  - Insbesondere Speicher für Variablen (Skalare, Arrays, usw.)
  - Größen aller Variablen (Objekte im Speicher) **stehen zur Übersetzungszeit fest** bzw. sind zur Laufzeit unveränderlich
  - Für **Programmcode** und **statische Variablen** steht Speicher fester Größe bereit (Adressberechnung zur Übersetzungszeit)
  - Für **Parameter und lokale Variablen** ist ein **Aufrufstapel** vorgesehen, dessen Größe mit Funktionsaufrufen variiert, typischerweise jedoch vom Betriebssystem stark beschränkt wird
- Problematisch, wenn der Speicherbedarf erst zur Laufzeit feststeht
  - **Beispiel:**  
Einlesen einer Datei in den Hauptspeicher, Dateigröße im Voraus nicht bekannt
  - **Mögliche Lösung:** Große statisch allokierte Puffer (Arrays)

- **Anwendungsbeispiel:** Einlesen einer Zeile aus einer Datei
  - per Array (Bereitstellung durch Aufrufer)

```

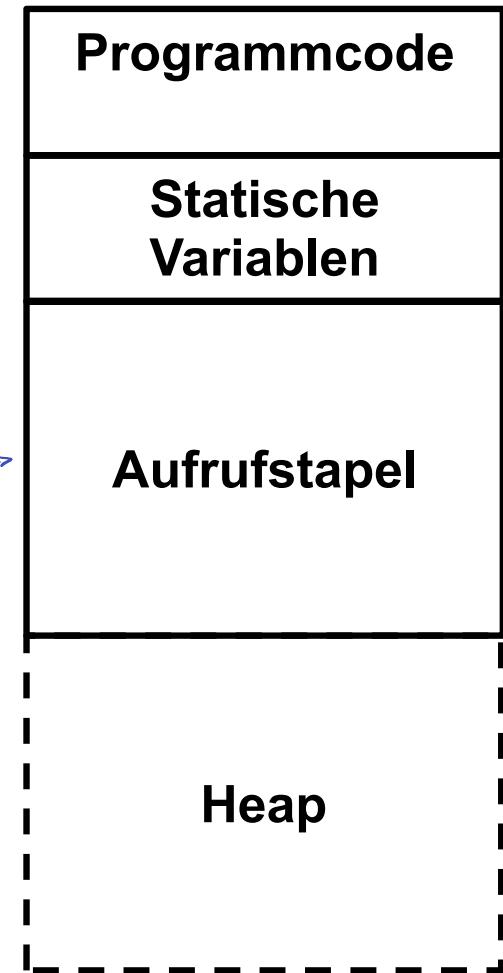
int readLine(FILE *f, char *buffer, unsigned int bsize) {
  unsigned int length = 0;
  int Muss EOF representieren können ch = '\0';
    ↗ Bereits am Ende vom File?
  if( feof(f) ) { return 0; }   ↗ Datei am Ende
                                ↗ Zeile am Ende
                                ↗ Puffer am Ende
  while( ((ch = fgetc(f)) != EOF) && (ch != '\n') && (length < bsize-1) ) {
    buffer[length++] = ch;
  }
  buffer[length] = '\0';
  return 1;
}
    ↗ Irgendwas eingelesen, inkl. leerem String
...
char line[32] = { '\0' };
if( readLine(..., line, sizeof(line)) ) { ... }
    ↗ Zeiger auf d. erste Element
  
```

## Dynamische Speicherverwaltung (3)

- Problematisch, wenn der Speicherbedarf erst zur Laufzeit feststeht (Forts.)
  - Puffer als lokale Variable, deren Größe durch den Aufrufstapel beschränkt wird
  - oder
  - als globale statische Variable
  - Puffer immer zu klein oder zu groß
- **Bessere Lösung:** Anforderung von beliebigen kontinuierlichen Speicherbereichen **zur Laufzeit**
- Bereitstellung durch die Laufzeitumgebung (C-Bibliothek, Betriebssystem) auf dem sogenannten **Heap**

*Besitzt max. Größe  
zur Laufzeit*

*Nur virtuell angelegt*
- Der **Heap** nimmt initial praktisch keinen Speicher in Anspruch (die Bereitstellung durch das Betriebssystem erfolgt bedarfsweise) und kann im Rahmen definierter Schranken beliebig wachsen und wieder schrumpfen



- **void\*** ist ein besonderer generischer Zeigertyp; er meint nicht, dass auf "nichts" (**void**) gezeigt würde, sondern **dass der Typ, auf den gezeigt wird, undefiniert (generisch) ist**
- Zeiger jedes (Objekt-)Zeigertypen können **implizit im Rahmen der Zuweisungskonvertierung** in **void\*-Zeiger** konvertiert werden
- **void\*-Zeiger** können **implizit im Rahmen der Zuweisungskonvertierung** in Zeiger beliebiger anderer Zeigertypen konvertiert werden
- **Zweck:** Funktionen zum Speichermanagement, die unabhängig vom Elementtyp des verwalteten Speichers definiert sein sollen
- Operationen über **void\*-Zeigern** sind eingeschränkt (da die Größe der Objekte, auf die gezeigt wird, nicht bekannt ist)
  - keine Dereferenzierung (per **\*** oder per **[ ]**) *← Man weiss schließlich nicht welche Werte referenziert werden.*
  - keine Zeigerarithmetik

Quelltyp Q	Zieltyp Z	Anmerkungen
Integral (inkl. Aufzählungen)	Integral (inkl. Aufzählungen)	möglicherweise Informationsverlust oder undefiniertes Ergebnis
Zeiger auf A		Nicht zulässig
Integral		Ausschließlich für die Zahl 0 ( <b>NULL</b> -Zeiger) zulässig
Zeiger auf A	Zeiger auf B	Nur zulässig für <b>NULL</b> -Zeigerkonstante oder wenn A zu B kompatibel (typischerweise also identischer Zeigertyp); <b>außerdem zulässig, wenn Q oder z der Typ void* ist</b>
Array von A		Konvertierung in Zeiger auf erstes Element (A = B)
Strukturtyp	Strukturtyp	Nur möglich für einen kompatiblen Strukturtypen
beliebig	<b>void</b>	Nicht möglich, da Ausdruck L nie den Typ <b>void</b> hat

```
int i = ...;
int *i_ptr;
char *c_ptr;
void *v_ptr;
```

```
i_ptr = &i;
c_ptr = i_ptr; /* Fehler */
v_ptr = i_ptr; /* OK: Zuweisung auf void* */
c_ptr = v_ptr; /* OK: Zuweisung von void* */
```

- **void \*malloc(size\_t size)** • memory allocation  $\Rightarrow$  malloc
  - Reserviert einen kontinuierlichen Speicherbereich von **size** Bytes und liefert einen **Zeiger auf das erste Byte** zurück

```

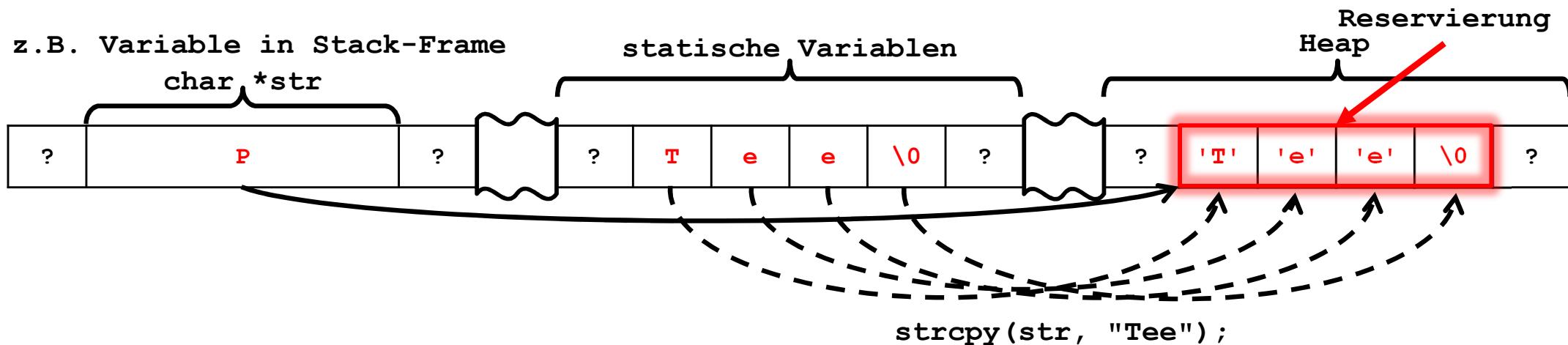
char *str = malloc( strlen("Tee") * sizeof(char) );           /* malloc( 3 ) */
strcpy(str, "Tee");
/* OK? */
  
```

*[ Hier überflüssig, geht nur darum zu zeigen, dass man stets mit der Größe eines einzelnen Elements multipliziert. ]*

```

char *str = malloc( (strlen("Tee") + 1) * sizeof(char) ); /* malloc( 3 + 1 ) */
strcpy(str, "Tee");
  
```

*[Um Nullsymbol mit einzubeziehen]*



## Funktionen zur dynamischen Speicherverwaltung (stdlib.h) (2)

- **void \*malloc(size\_t size)**

- Ist die Allokation nicht möglich, wird der **NULL**-Zeiger zurückgegeben  
=> typischerweise in diesem Fall Aufruf von **exit()** (**berechtigte Panik**)

```
char *str = malloc( ... );
if( str == NULL ) {                                /* oder: if( !str ) { */
    exit(...);
}
strcpy(str, ...);
...
```

↑  
Halb A. Allokieren geklappt, ansonsten  
grundlegend problematische Programmausführung → exit

- Ergebnis einer Anforderung von 0 Bytes ist **implementationsabhängig**:
  - **NULL**-Zeiger
  - **oder**
  - Ein eindeutiger Zeiger, der nicht dereferenziert werden darf

```
char *str = malloc( 0 );
/* str im Anschluss nicht unbedingt NULL */
```

↳ Falls Zeiger ins Nichts zeigt → darf NICHT dereferenziert werden.

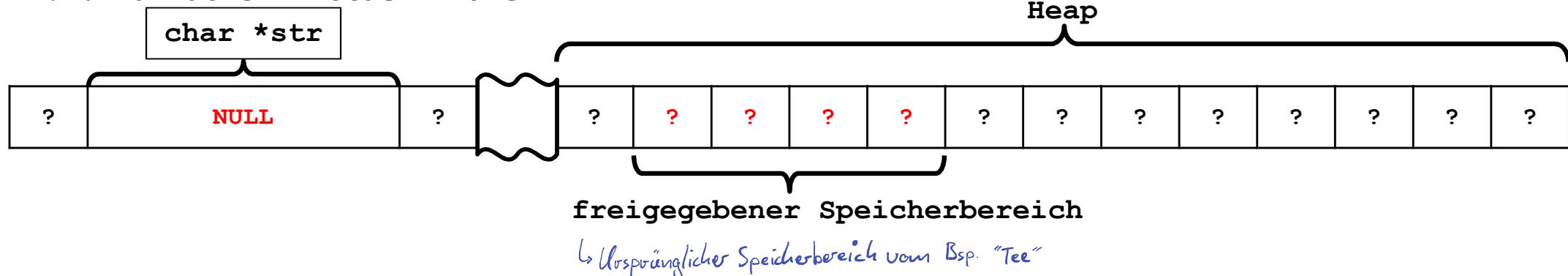
## Funktionen zur dynamischen Speicherverwaltung (stdlib.h) (3)

- **void free(void \*ptr)**
  - Hebt die Reservierung eines dynamisch reservierten Speicherbereichs auf
  - Angabe des Speicherbereichs per Parameter **ptr** (Zeiger auf erstes Byte des Bereichs)
  - **free(NULL)** ist zulässig und hat keine Wirkung
  - Zeiger **ptr** nach Freigabe **ungültig** (Semantik von Dereferenzierungen undefined)  
=> sinnvollerweise auf **NULL**-Zeiger setzen

```

char *str = malloc( strlen("Tee") + 1 ) * sizeof(char) ;
strcpy(str, "Tee");
...
free(str);
str = NULL;
  
```

z.B. Variable in Stack-Frame



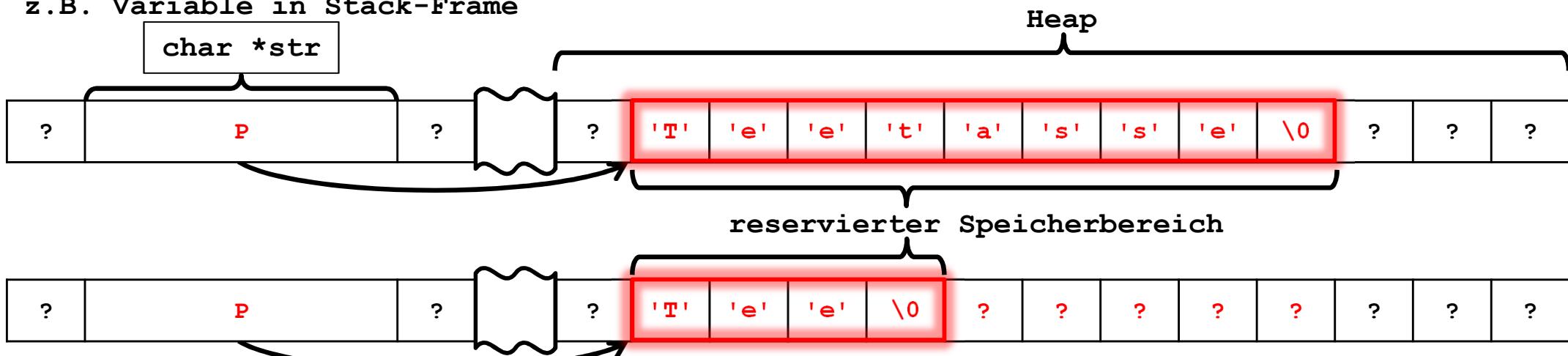
- **void \*realloc(void \*ptr, size\_t size);**

- Verkleinert oder vergrößert eine **bestehende dynamische Reservierung** der Größe `n`, auf deren erstes Byte der Parameter `ptr` zeigt
- Der Parameter `size` gibt die **neue Größe** des reservierten Speicherbereichs in Byte an
- Neue Größe `size <= n`: Anfang und Inhalt (bis zur neuen Größe) des Speicherbereichs bleiben **unverändert**, der Zeiger bleibt gültig

```

char *str = malloc( strlen("Teetasse") + 1 ) * sizeof(char) );
strcpy(str, "Teetasse");
...
str = realloc( str, (strlen("Tee") + 1) * sizeof(char) );
str[3] = '\0';
  
```

z.B. Variable in Stack-Frame

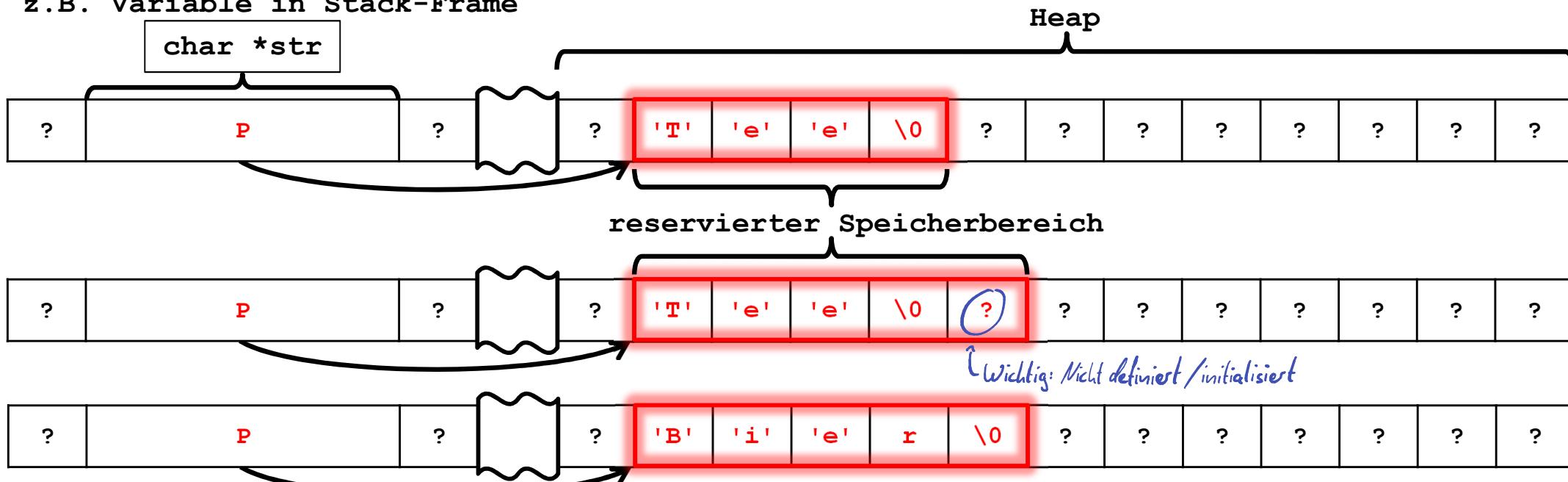


- **void \*realloc(void \*ptr, size\_t size);** (Forts.)

- Neue Größe **size > n**: Inhalt des Speichers bis zur bisherigen Größe **unverändert** (Rest undefined)

```
char *str = malloc( strlen("Tee")+1 ) * sizeof(char) );
strcpy(str, "Tee");
...
str = realloc( str, (strlen("Bier")+1) * sizeof(char) );
strcpy(str, "Bier");
```

z.B. Variable in Stack-Frame



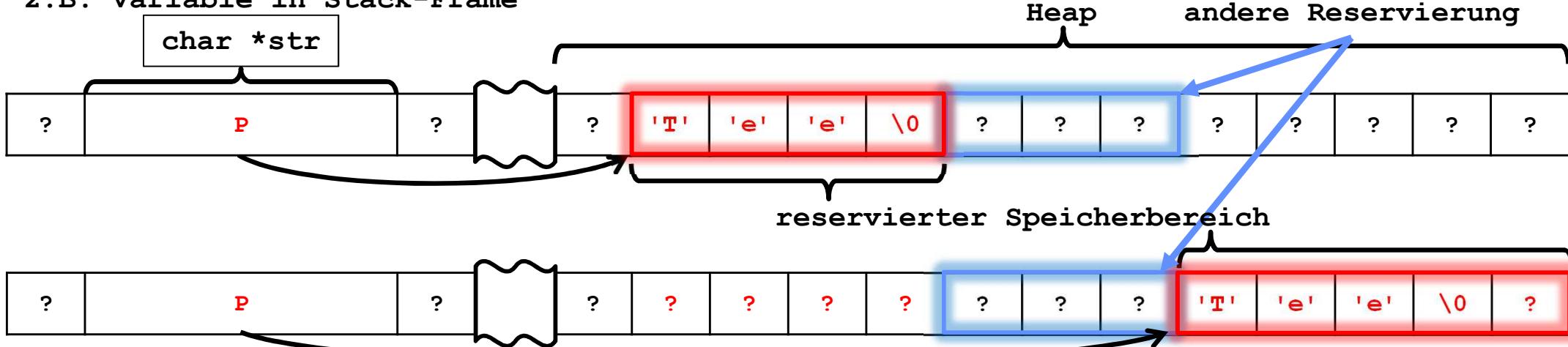
- **void \*realloc(void \*ptr, size\_t size);** (Forts.)

- Void pointer* ↗
- Neue Größe **size > n**: Inhalt des Speichers bis zur bisherigen Größe **unverändert** (Rest undefiniert), möglicherweise Verschiebung des Speicherbereichs im Adressraum und damit **neuer Anfangszeiger** (Rückgabewert), **bisheriger Zeiger wird damit ungültig**

```

char *str = malloc( strlen("Tee")+1 ) * sizeof(char) );
strcpy(str, "Tee");
...
str = realloc( str, (strlen("Bier")+1) * sizeof(char) );
  
```

z.B. Variable in Stack-Frame



## Funktionen zur dynamischen Speicherverwaltung (stdlib.h) (7)

- **void \*realloc(void \*ptr, size\_t size);** (Forts.)
  - Reservierung schlägt fehl: **NULL**-Zeiger als Rückgabewert, **ptr** bleibt gültig (mit bisheriger Größe)
  - Neue Größe **size == 0**: Entspricht **malloc(0)**, gibt Speicher gegebenenfalls frei

```

char *str = malloc( (strlen("Tee")+1) * sizeof(char) );
...
str = realloc( str, 0 );
/* str im Anschluss nicht unbedingt NULL, daher: */
str = NULL;

```
- **ptr == NULL**: Entspricht Aufruf von **malloc(size)** (Erstreservierung)
- **void \*calloc (size\_t nmemb, size\_t size)**
  - Äquivalent zu **malloc()**, aber
    - Statt **size** Bytes wird die Anzahl Objekte **nmemb** und die Größe **size** eines einzelnen Objektes angegeben
    - Speicherbereich wird mit 0-Bits initialisiert

## Anwendungsbeispiel

- **Anwendungsbeispiel:** Einlesen einer Zeile aus einer Datei
  - per dynamischer Speicherverwaltung (Speicherreservierung durch Einlesefunktion)

```

char *readLine(FILE *f) {
    unsigned int length = 0;
    char *buffer = NULL;
    int ch = '\0';

    if( feof(f) ) { return NULL; }

    while( ((ch = fgetc(f)) != EOF) && (ch != '\n') ) {
        buffer = realloc( buffer, sizeof(char) * (length + 1) );
        if( buffer == NULL) { exit(1); }
        buffer[length++] = ch;
    }
    buffer = realloc( buffer, sizeof(char) * (length + 1) );
    if( buffer == NULL) { exit(1); }
    buffer[length] = '\0';
    return buffer;
}
...
char *line = NULL;
if( (line = readLine(...)) != NULL ) { ... }

```

Siehe vorherige Version, hier keine  
Pufferprüfung

- Puffersgröße normalerweise in Blöcken reallokiert & nicht pro neues Byte.

## Verkettete Listen (1)

- Inhärent dynamische Datenstruktur zur Verwaltung einer Folge von Elementen
- je verwaltetem Element ein Listenknoten, der neben dem Element einen Verweis (Zeiger) auf den jeweils nächsten Listenknoten umfasst
- Dynamische Reservierung des Speichers für einen Listenknoten beim Einfügen eines Elementes, Freigabe beim Entfernen eines Elementes

```
struct ListNode {
    ??? payload;
    struct ListNode next;
}; Hier irrelevant
```



*/\* struct ListNode ist noch unvollständig  
=> Größe von next nicht berechenbar \*/*

```
struct ListNode {
    ??? payload;
    struct ListNode *next;
};
```

*/\* OK: Größe von next berechenbar, da  
Zeigertyp \*/*



```
struct ListNode {
    ??? payload;
    struct ListNode *next;
};

/* eigener Typname für Listen */
typedef struct ListNode *List;
```



```
/* forward reference */
typedef struct ListNode *List;

struct ListNode {
    ??? payload;
    List next;
};
```

*Vorwärtsreferenz gültig  
weil Zeigertyp*



## Verkettete Listen (2)

```

typedef char           *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
  
```

- Repräsentation der leeren Liste als **NULL-Zeiger**

### Leere Liste

```
List emptyList() {
    return NULL;
}
```

### Test auf leere Liste

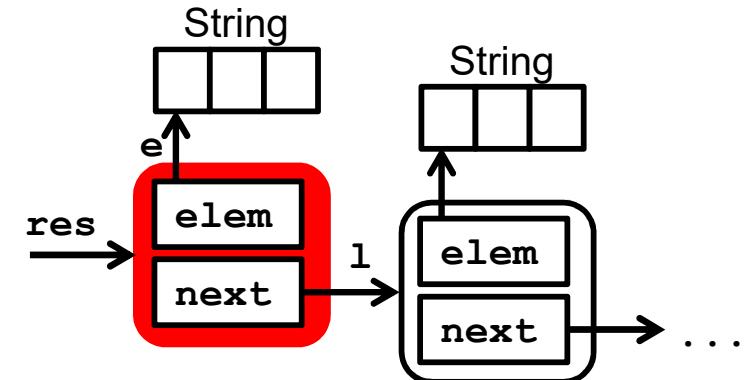
```
int isEmpty(List l) {
    return l == emptyList();
}
```

- Erzeugen eines Listenknotens

```

List cons(List l, Element e) {
    List res = malloc( sizeof(struct ListNode) );
    /* oder: malloc( sizeof(*l) ); */
    if(res == NULL) {
        exit(1);
    }
    res->elem = e;
    res->next = l;
    return res;
}
  
```

Standardfehler `sizeof(List)` würde nur die Größe vom Listenpointers liefern  
 • Typen vor & nach `malloc` MÜSSEN sich unterscheiden



## Verkettete Listen (3)

```

typedef char           *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
  
```

### ▪ Kopfelement

```

Element head(List l) {
  assert( !isEmpty(l) );
  return l->elem;
}
  
```

### Restliste

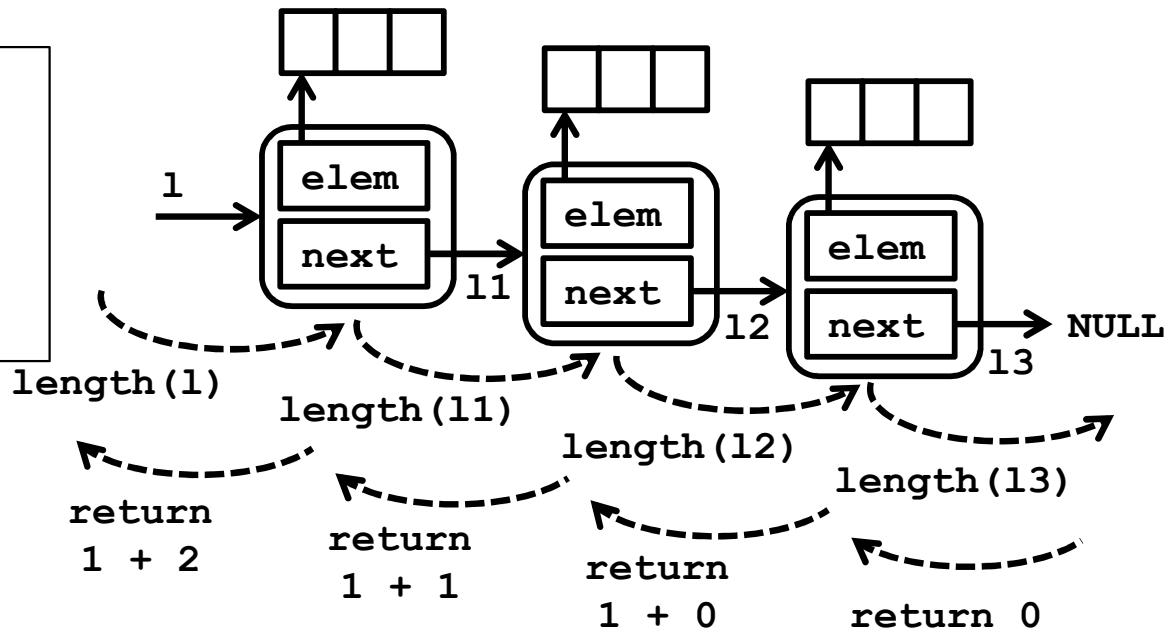
```

List tail(List l) {
  assert( !isEmpty(l) );
  return l->next;
}
  
```

### ▪ Listenlänge

```

unsigned int length(List l) {
  return isEmpty(l)
    ? 0
    : 1 + length( tail(l) )
;
}
  
```



## Verkettete Listen (4)

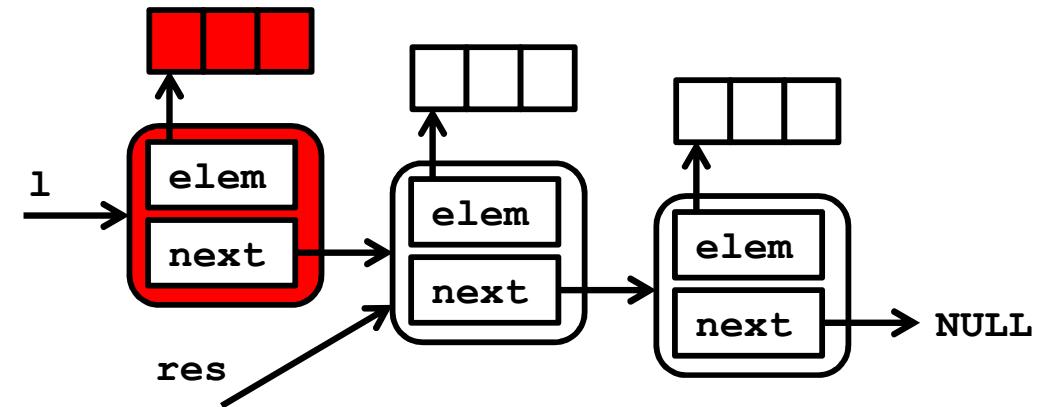
```

typedef char           *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
  
```

- Löschen des Listenkopfes

```

List deleteFirst(List l) {
  assert( !isEmpty(l) );
  List res = tail(l);
  /* free( head(l) ); ??? */
  free( l );
  return res; eigentlich muss d. Zeiger noch mit einem Nullpointer versehen werden, durch anschl. Funktionsabbruch irrelevant
}
  
```



- Löschen einer Liste

```

List deleteAll(List l) {
  while( !isEmpty(l) ) {
    l = deleteFirst(l);
  }
  return emptyList();
}
  
```

## Verkettete Listen (5)

```

typedef char           *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
  
```

- **Zugriff auf ein Listenelement**

```

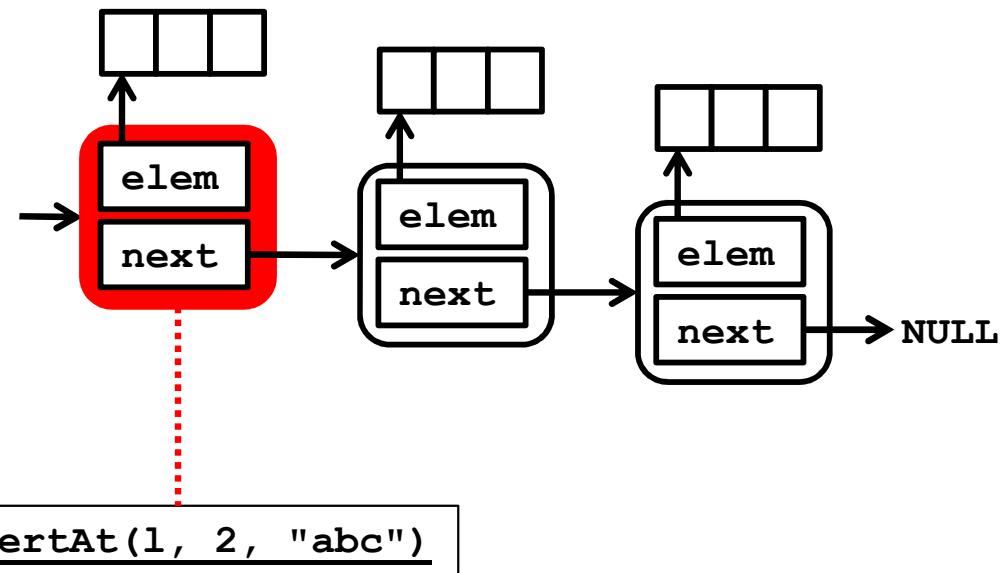
Element getAt(List l, unsigned int idx) {
  assert( !isEmpty(l) || (idx == 0) );
  return idx == 0
    ? head(l)
    : getAt( tail(l), idx-1 )
    ;
}
  
```

- **Einfügen in eine Liste**

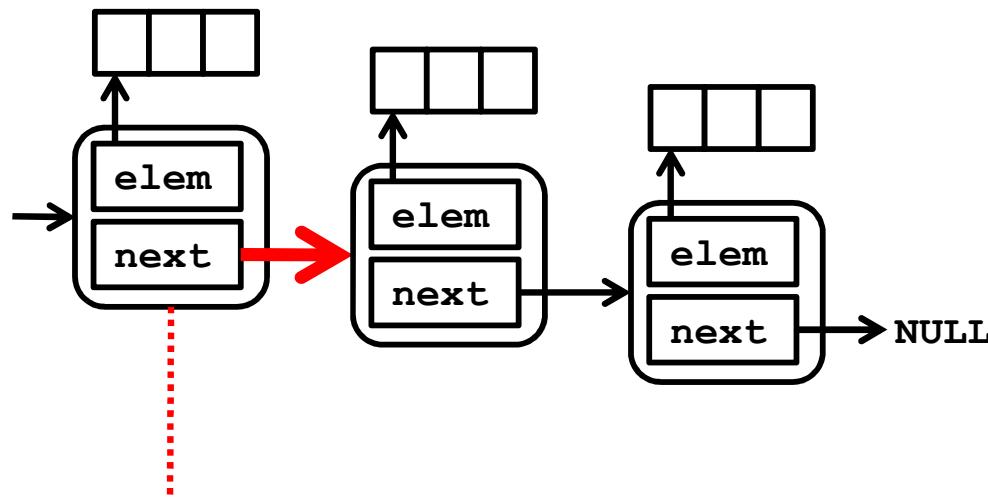
```

List insertAt(List l, unsigned int idx, Element e) {
  assert( !isEmpty(l) || (idx == 0) );
  return idx == 0
    ? cons(l, e)
    : ( l->next = insertAt( tail(l), idx-1, e ), l )
    ;
}
  
```

## Verkettete Listen (5) – Einfügen in eine Liste



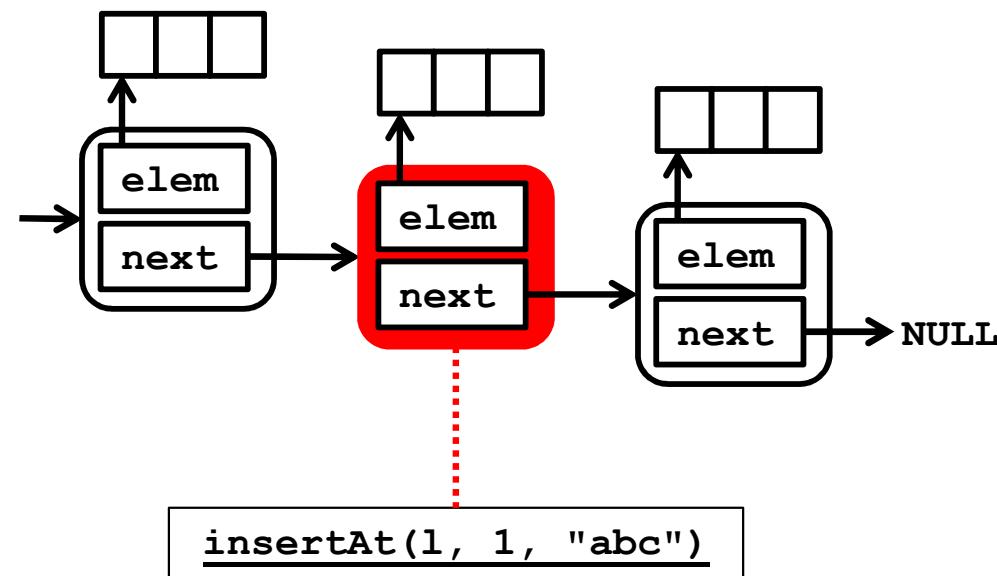
## Verkettete Listen (5) – Einfügen in eine Liste



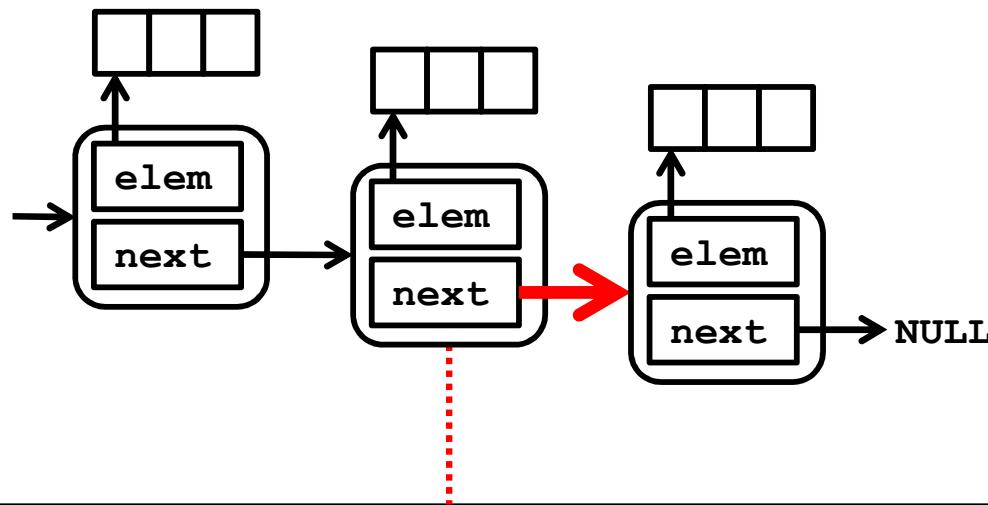
```

insertAt(1, 2, "abc")
return idx == 0
?
: ( l->next = insertAt( tail(l), idx-1, e ), l )
;
  
```

## Verkettete Listen (5) – Einfügen in eine Liste



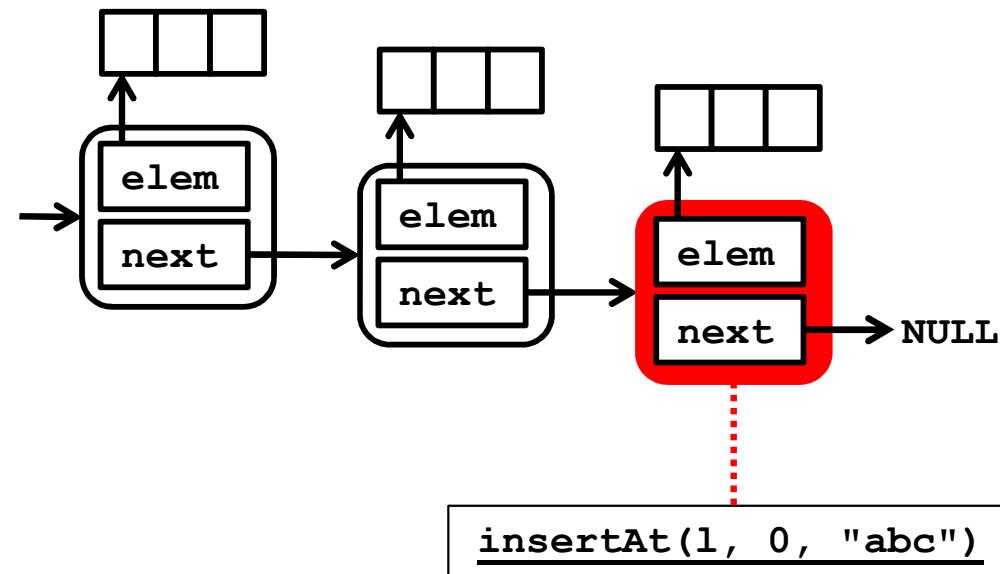
## Verkettete Listen (5) – Einfügen in eine Liste



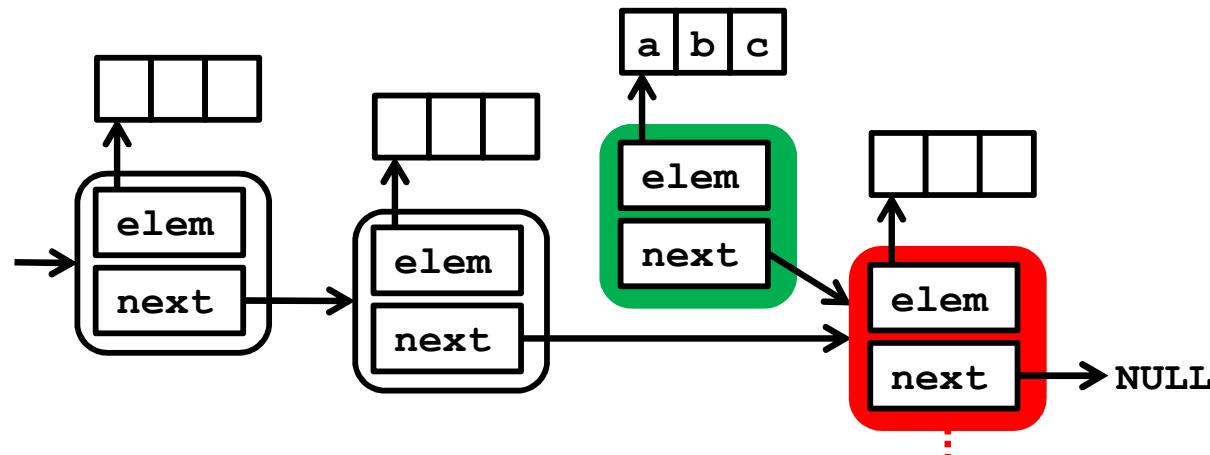
```

insertAt(l, 1, "abc")
return idx == 0
?
: ( l->next = insertAt( tail(l), idx-1, e ), l )
;
  
```

## Verkettete Listen (5) – Einfügen in eine Liste



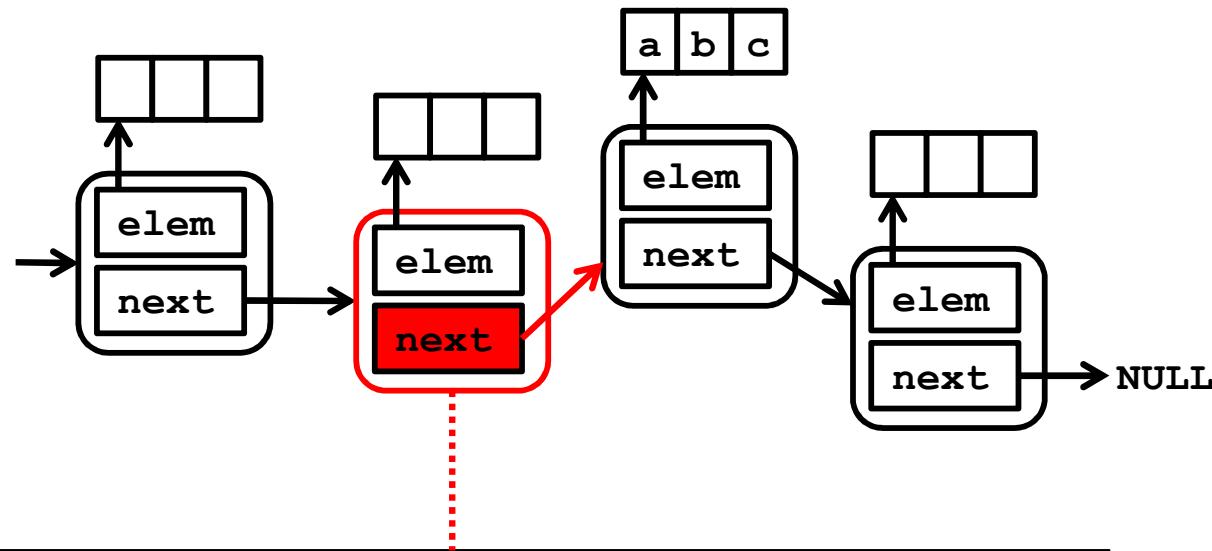
## Verkettete Listen (5) – Einfügen in eine Liste



```

insertAt(l, 0, "abc")
return idx == 0
? cons(l, e)
: ...
;
  
```

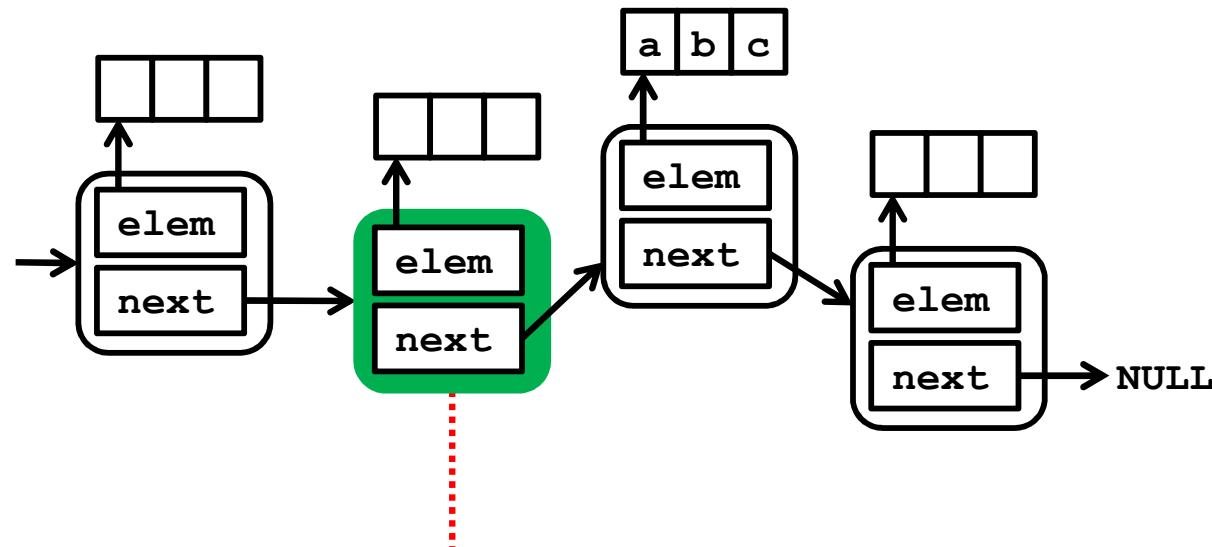
## Verkettete Listen (5) – Einfügen in eine Liste



```

insertAt(l, 1, "abc")
return ...
? ...
: ( l->next = insertAt( tail(l), idx-1, e ), l )
;
  
```

## Verkettete Listen (5) – Einfügen in eine Liste

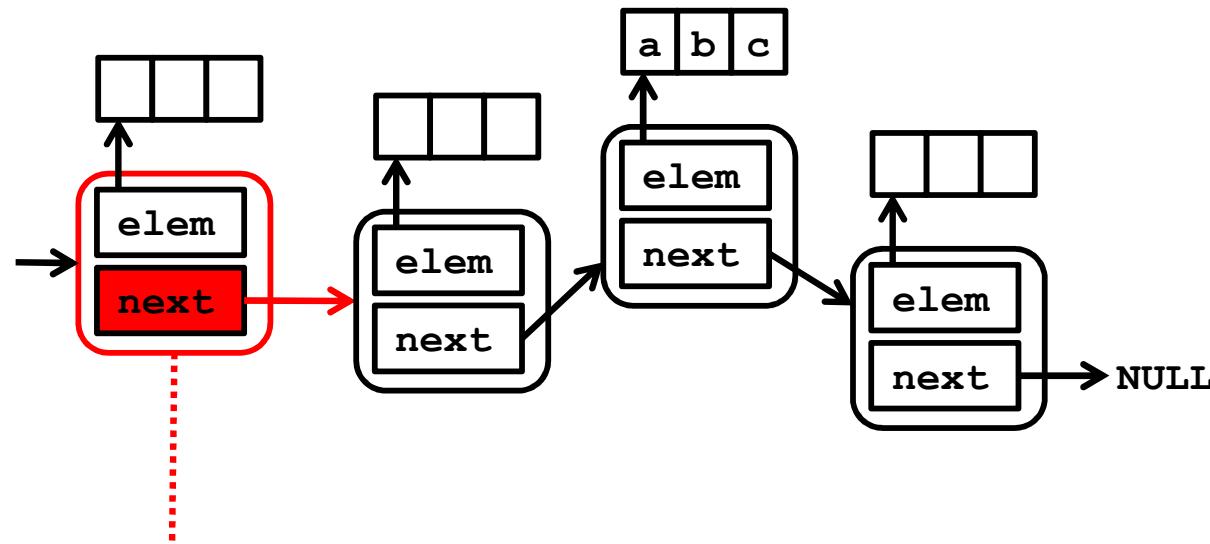


```

insertAt(1, 1, "abc")
return ...
? ...
: ( ..., 1 )
;

```

## Verkettete Listen (5) – Einfügen in eine Liste

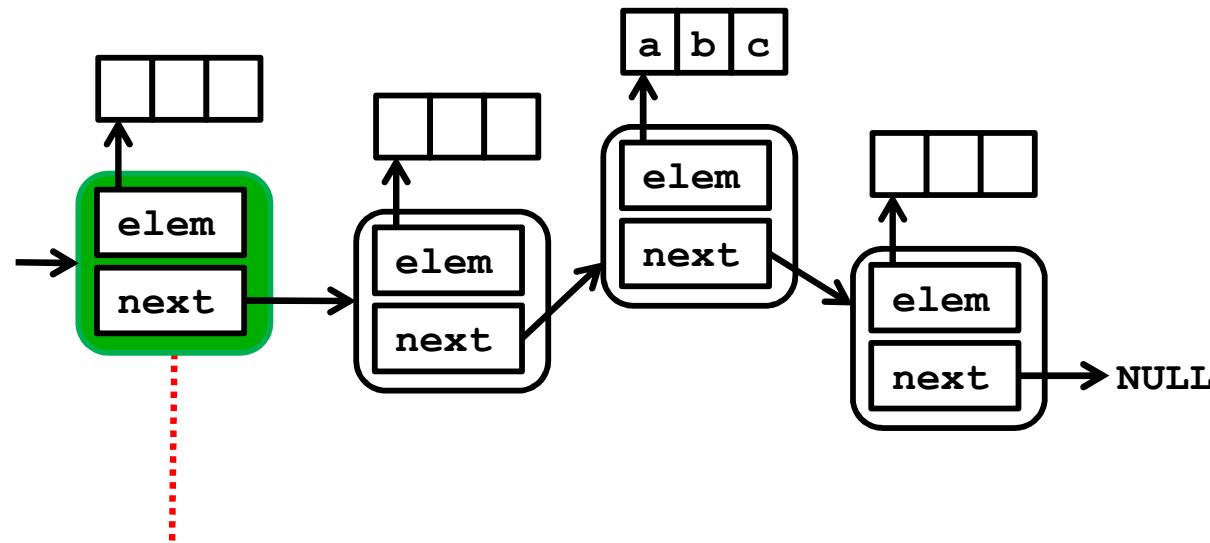


```

insertAt(1, 2, "abc")

return ...
?
: ( 1->next = insertAt( tail(1), idx-1, e ), 1 )
;
  
```

## Verkettete Listen (5) – Einfügen in eine Liste



```

insertAt(1, 2, "abc")

return ...
?
... 
: ( ..., 1 )
;

```

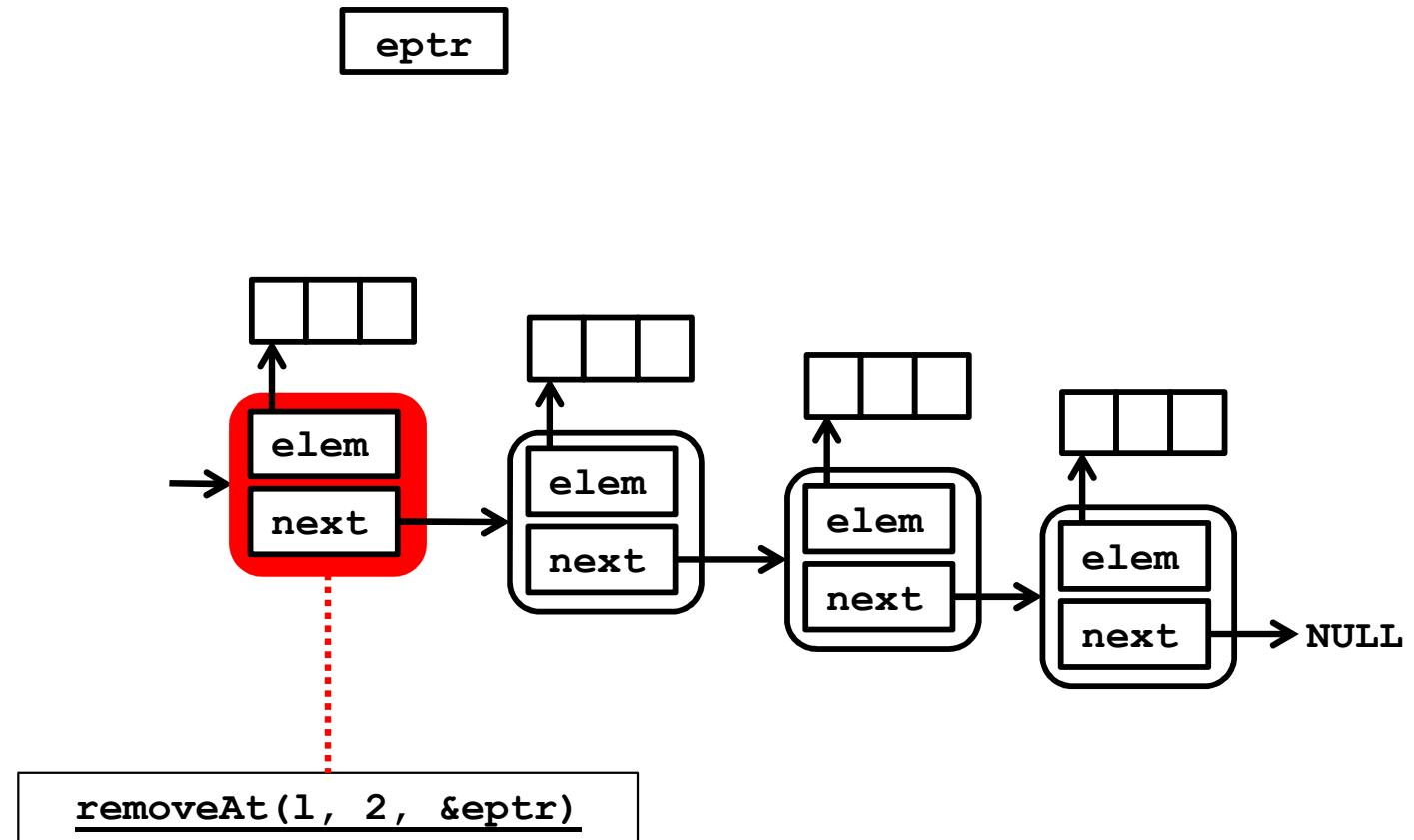
## Verkettete Listen (6)

```
typedef char           *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
```

- **Löschen aus einer Liste**

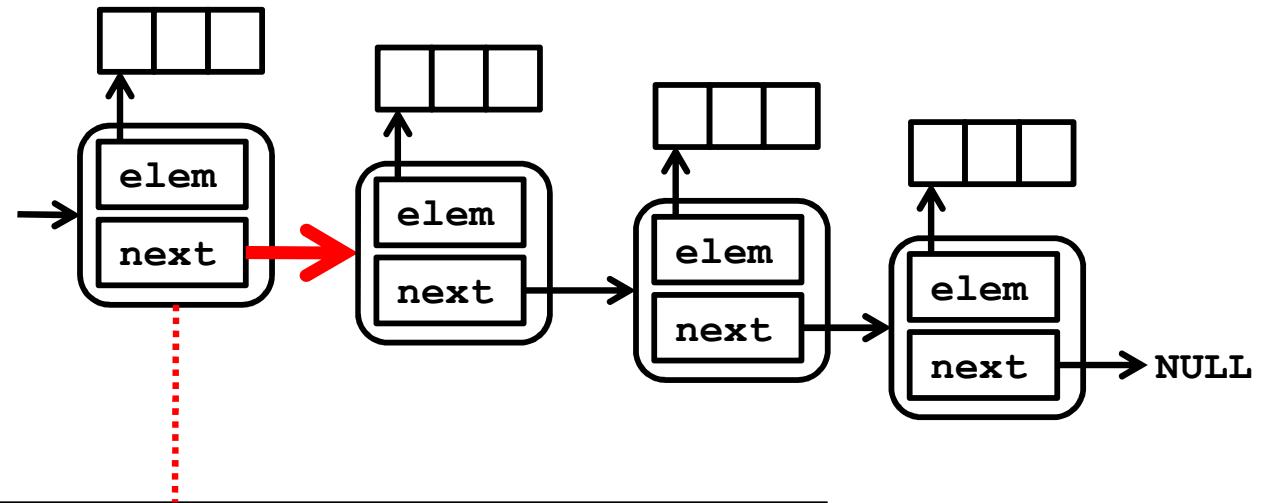
```
List removeAt(List l, unsigned int idx, Element *e) {
    assert( !isEmpty(l) || (idx == 0) );
    return idx == 0
        ? ( *e = head(l), deleteFirst(l) )
        : ( l->next = removeAt( tail(l), idx-1, e ), l )
        ;
}
```

## Verkettete Listen (6) – Löschen aus einer Liste



## Verkettete Listen (6) – Löschen aus einer Liste

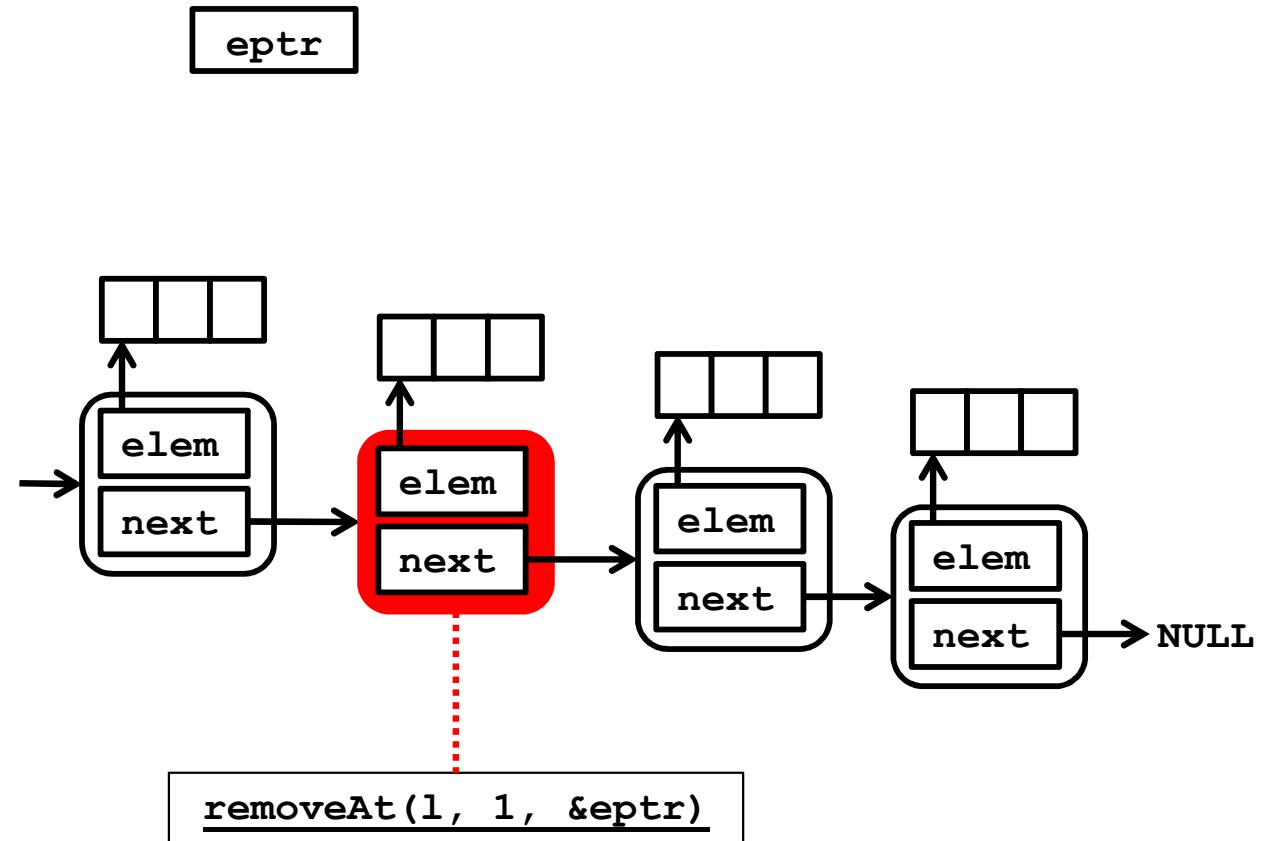
eptr



```

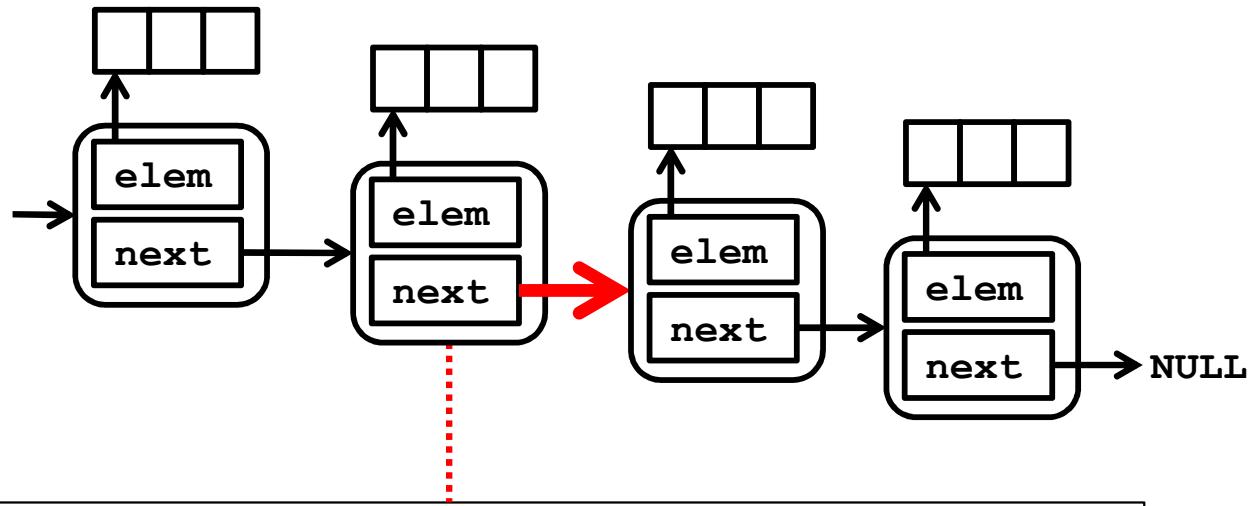
removeAt(1, 2, &eptr)
return idx == 0
?
: ( l->next = removeAt( tail(l), idx-1, e ), l )
;
  
```

## Verkettete Listen (6) – Löschen aus einer Liste



## Verkettete Listen (6) – Löschen aus einer Liste

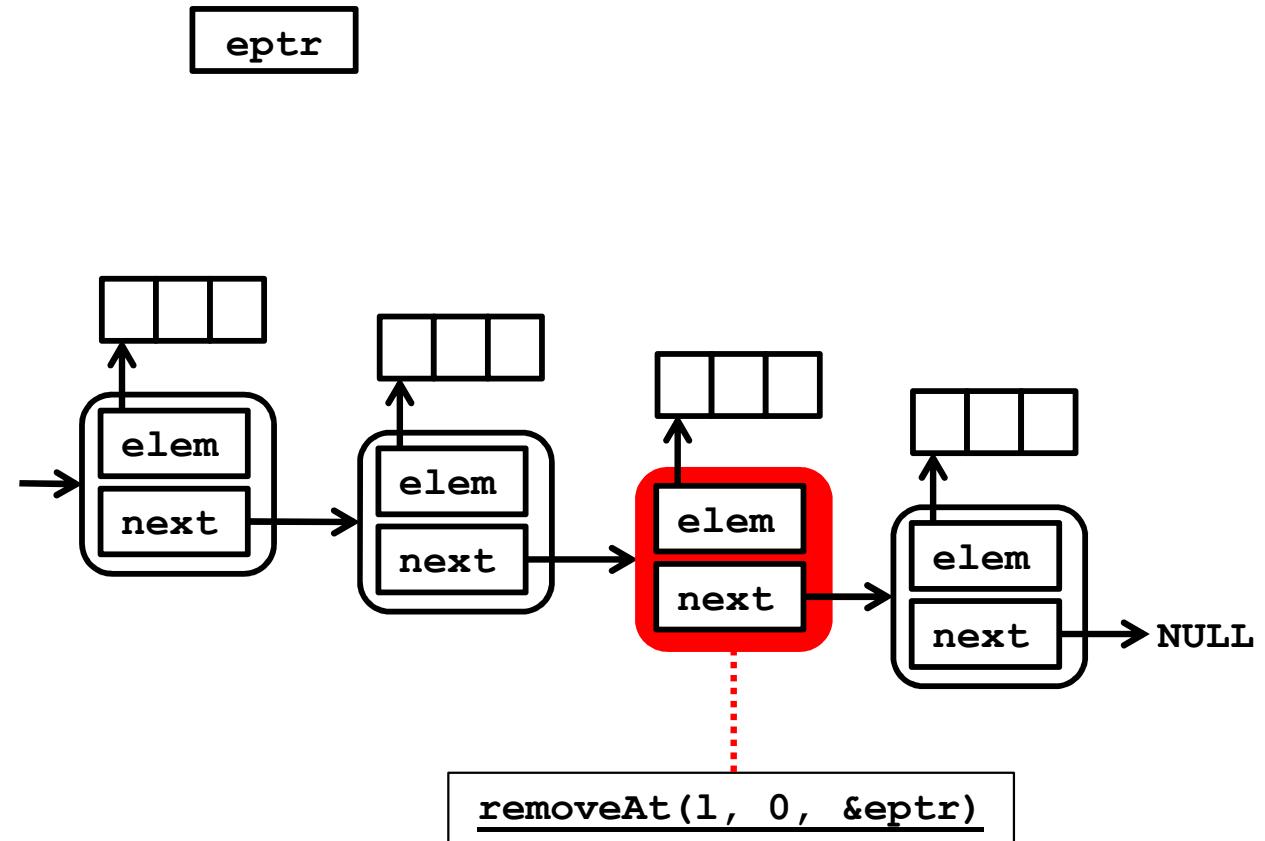
eptr



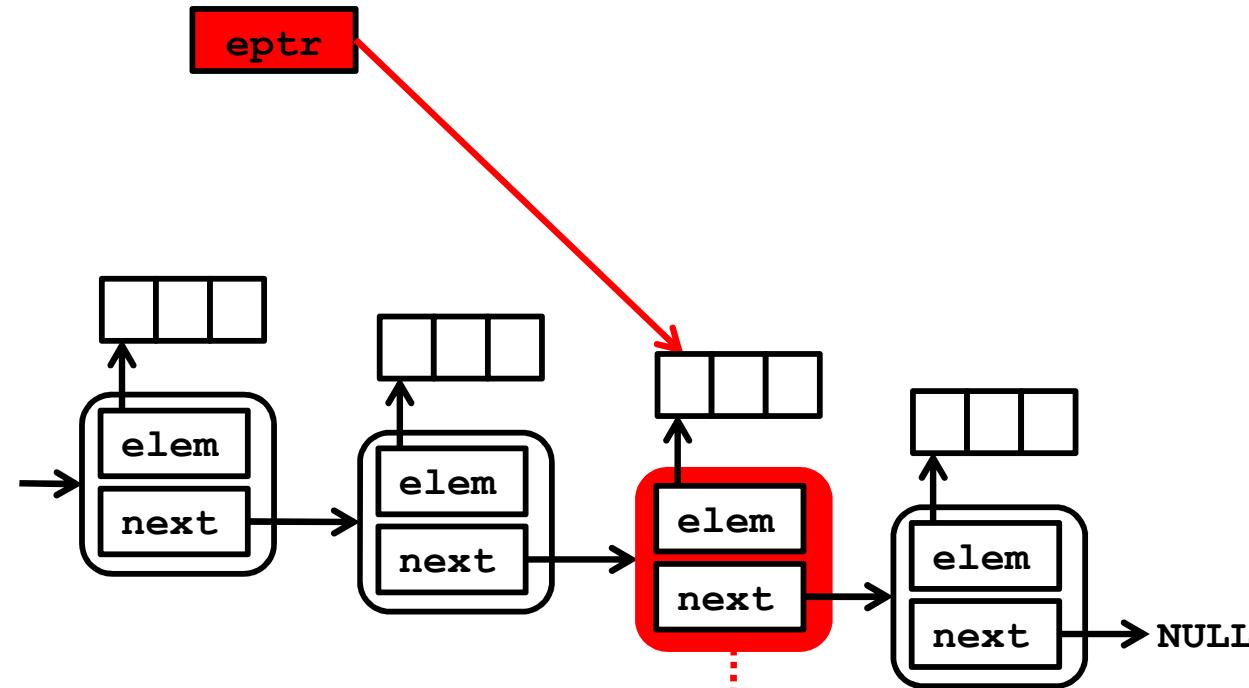
```

removeAt(l, 1, &eptr)
return idx == 0
?
: ( l->next = removeAt( tail(l), idx-1, e ), l )
;
  
```

## Verkettete Listen (6) – Löschen aus einer Liste



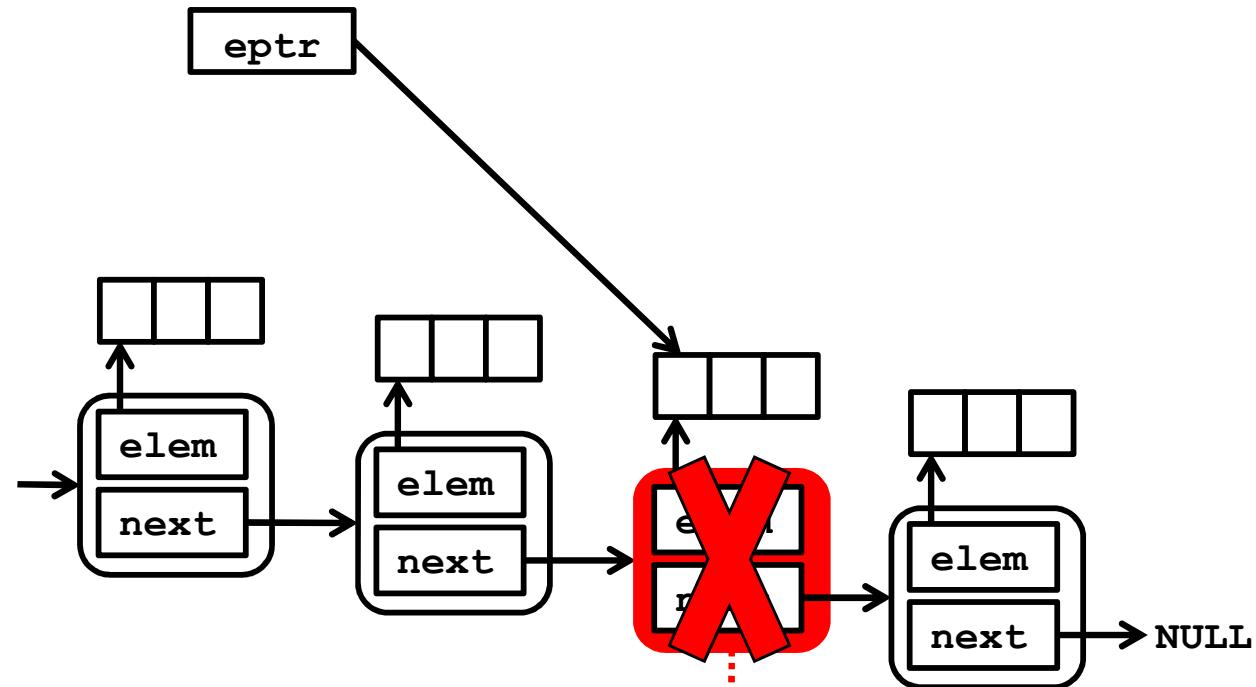
## Verkettete Listen (6) – Löschen aus einer Liste



```

removeAt(l, 0, &eptr)
return idx == 0
? ( *e = head(l), deleteFirst(l) )
: ...
;
  
```

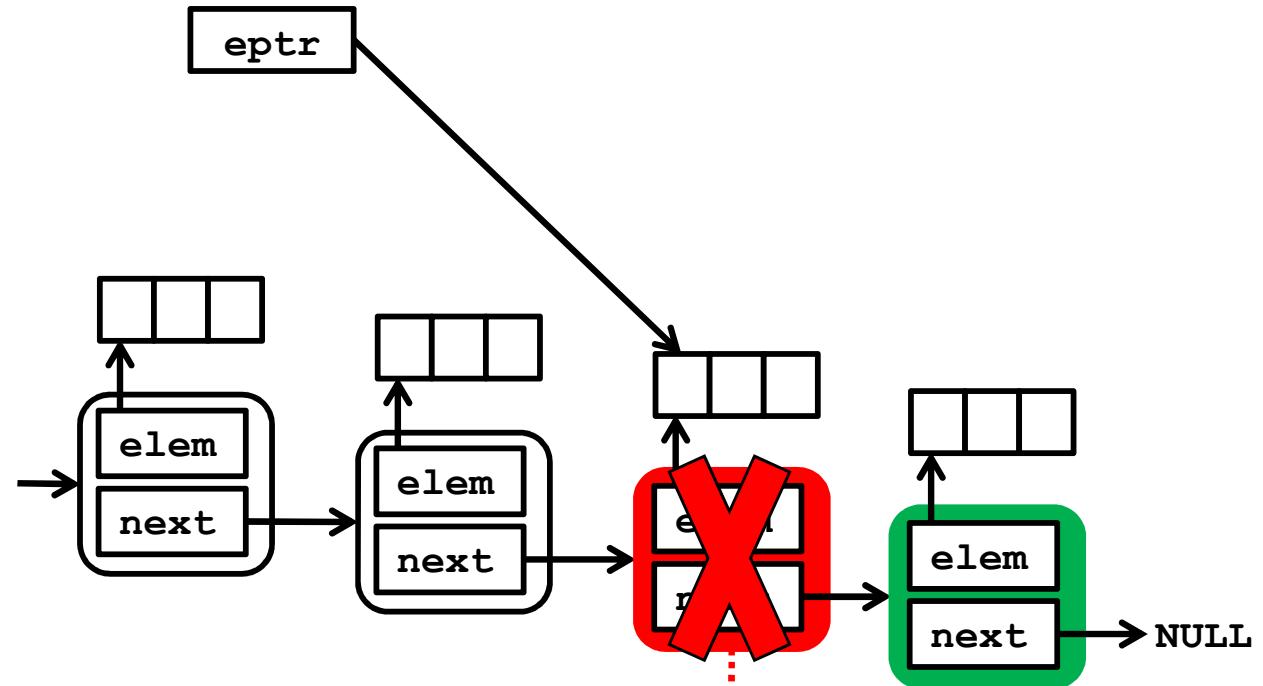
## Verkettete Listen (6) – Löschen aus einer Liste



```

removeAt(l, 0, &eptr)
return idx == 0
? ( ..., deleteFirst(l) )
: ...
;
  
```

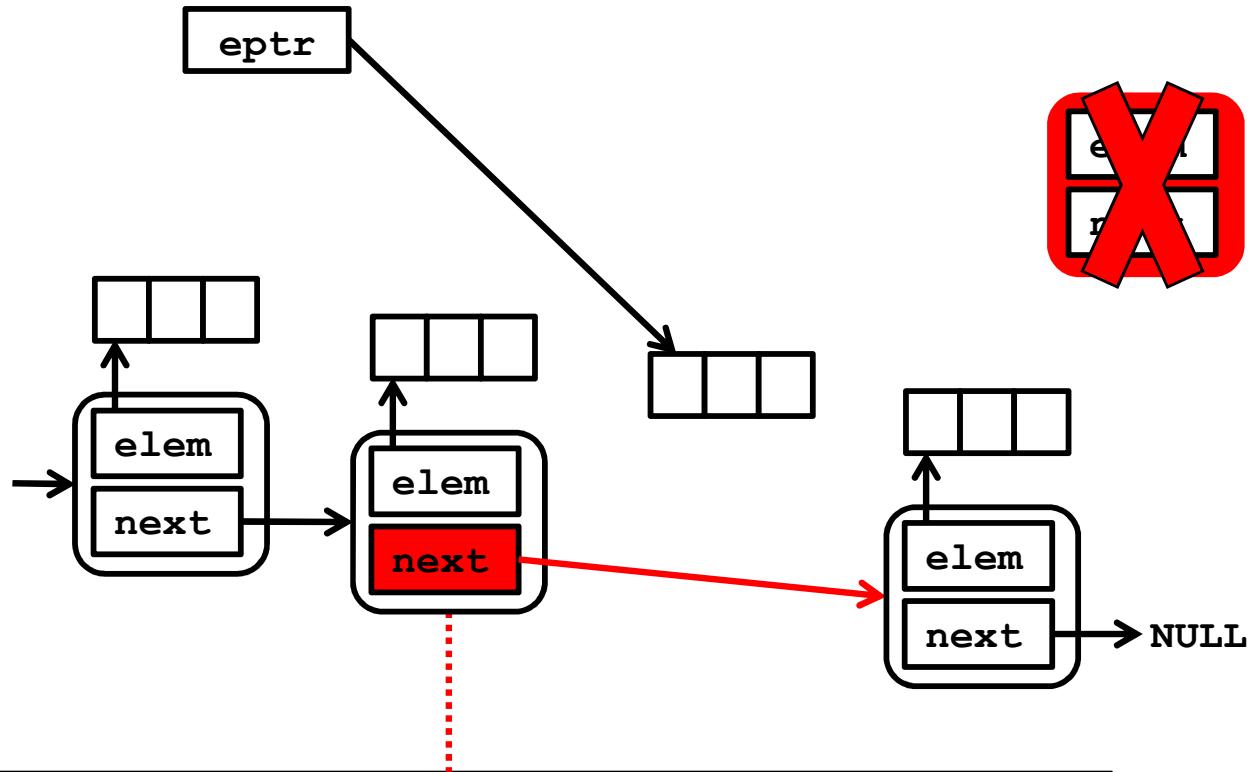
## Verkettete Listen (6) – Löschen aus einer Liste



```

removeAt(l, 0, &eptr)
return idx == 0
? ( ..., deleteFirst(l) )
: ...
;
  
```

## Verkettete Listen (6) – Löschen aus einer Liste

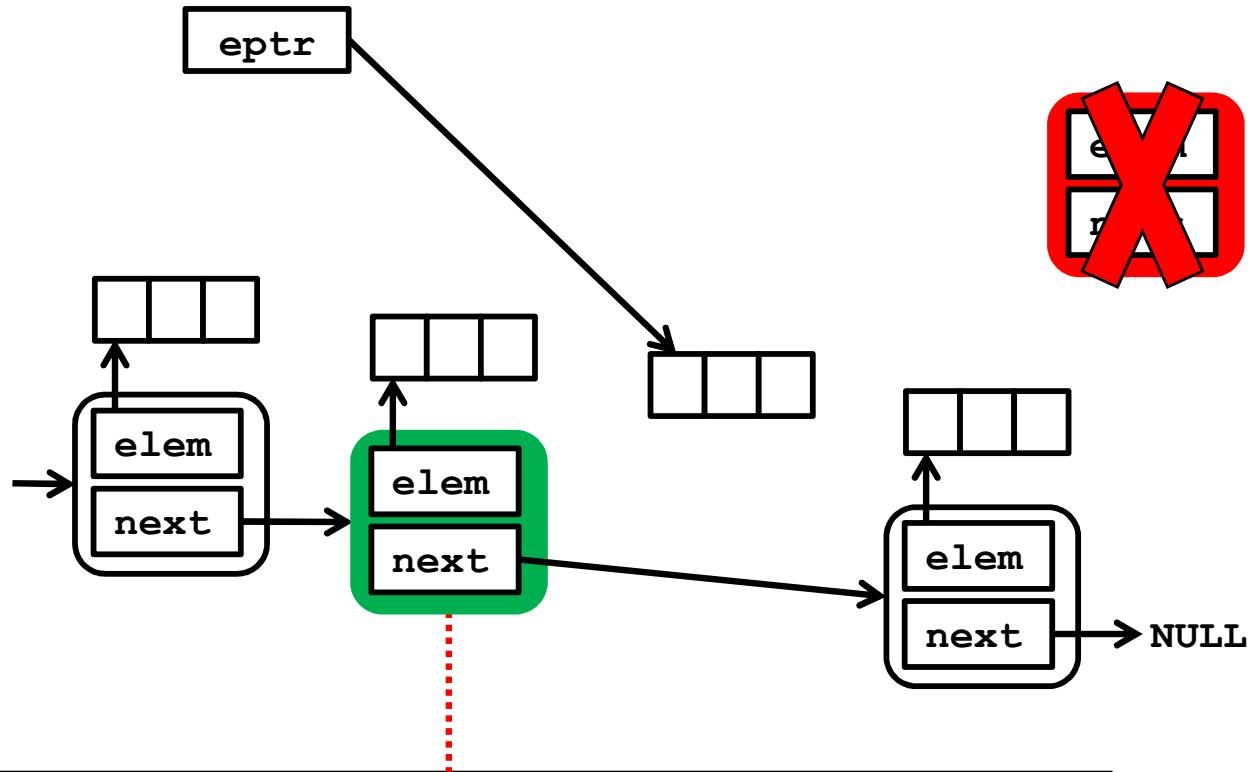


```

removeAt(l, 1, &eptr)
return ...
? ...
: ( l->next = removeAt( tail(l), idx-1, e ), l )
;

```

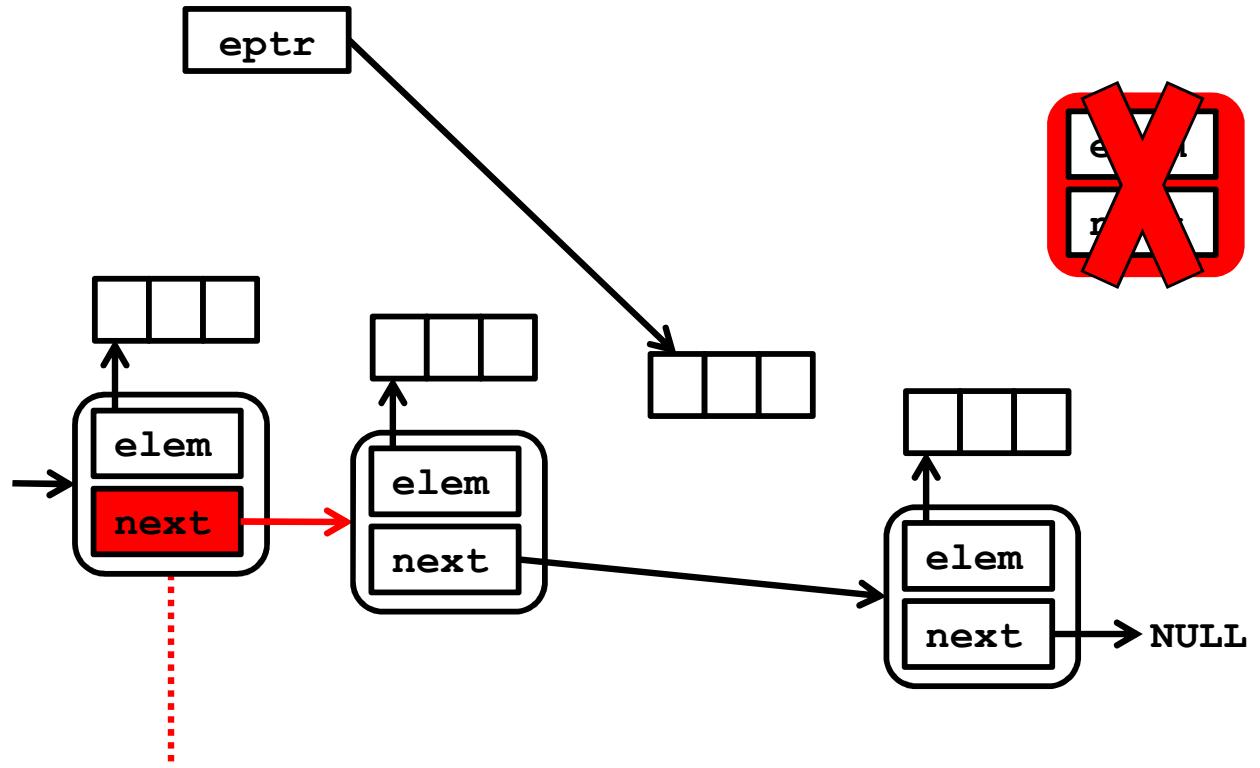
## Verkettete Listen (6) – Löschen aus einer Liste



```

removeAt(1, 1, &eptr)
return ...
? ...
: ( ..., 1 )
;
  
```

## Verkettete Listen (6) – Löschen aus einer Liste

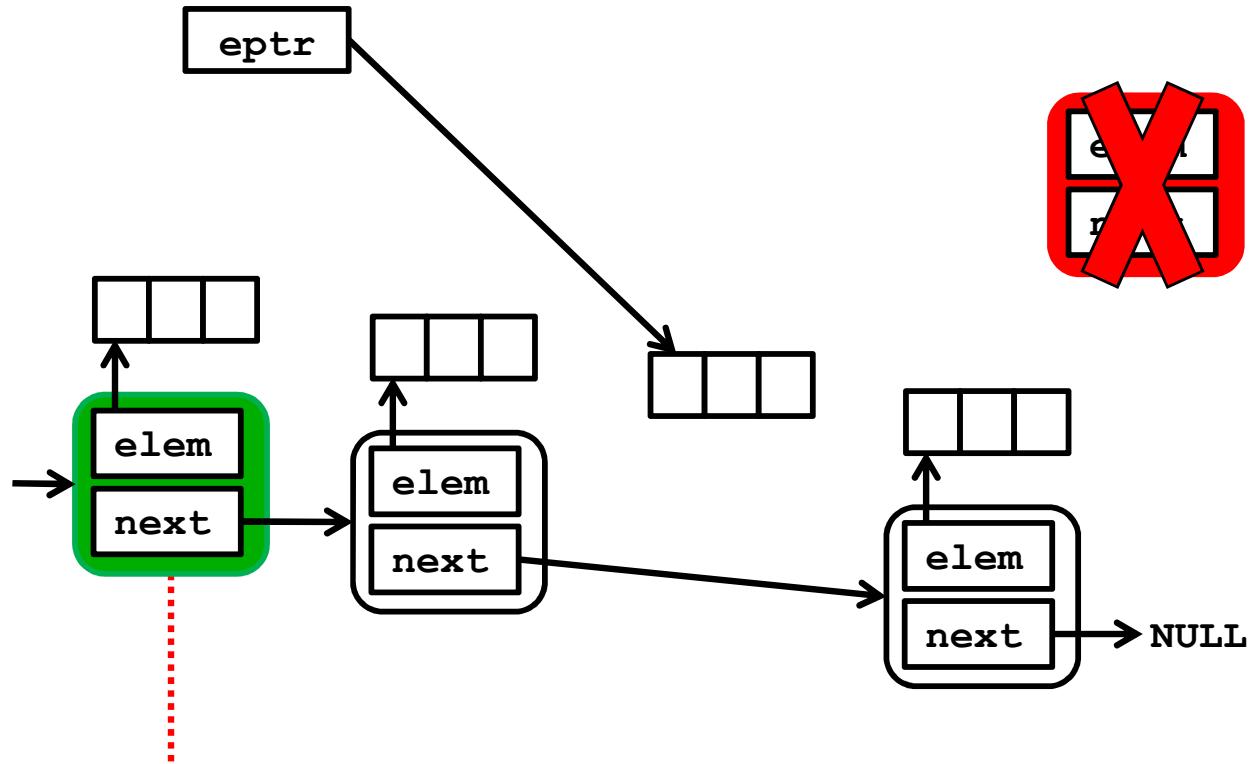


```

removeAt( l, 0, &eptr )

return ...
? ...
: ( l->next = removeAt( tail(l), idx-1, e ), l )
;
  
```

## Verkettete Listen (6) – Löschen aus einer Liste



```

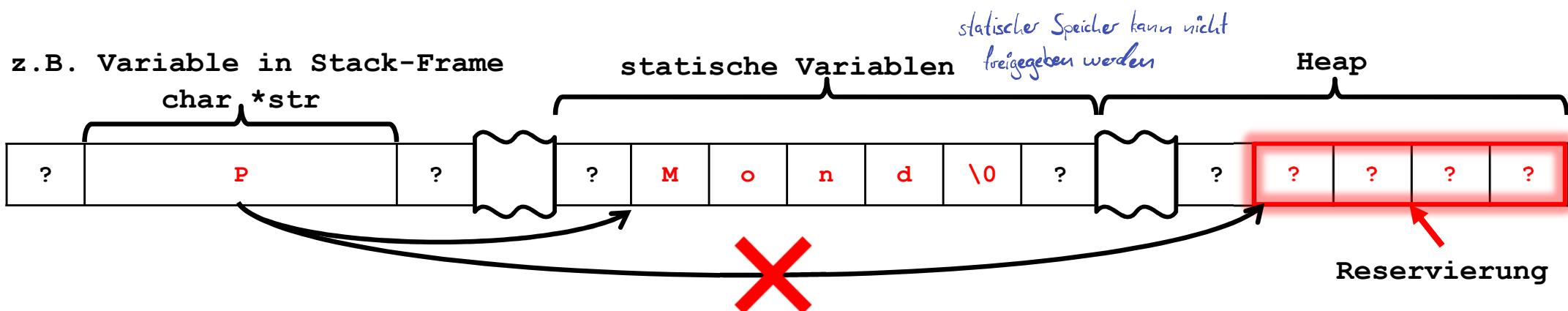
removeAt(1, 0, &eptr)
return ...
? ...
: ( ..., 1 )
;
  
```

# Typische Fehler bei dynamischer Speicherverwaltung (1)

- **Memory Leaks** (z.B. Überschreiben von Zeigern oder fehlendes `free()`)

```

char *str = malloc( ... );
str = "Mond";      /* Überschreibt Zeiger auf reservierten Speicher... */
free(str);         /* ... und später auch noch ein großer Knall */
    
```



- **Nullsymbol nicht korrekt (neu) gesetzt**

```

char *str = malloc( ... );
strncpy(str, "Zeichenfolge", 5);
/* Zeichenfolge in str nicht terminiert */
    
```

- `char *strncpy(char *dest, const char *src, size_t n):`  
Kopiert max. `n` Zeichen von `src` nach `dest`
- **Ist `src` länger als `n`, so wird kein Nullsymbol kopiert**

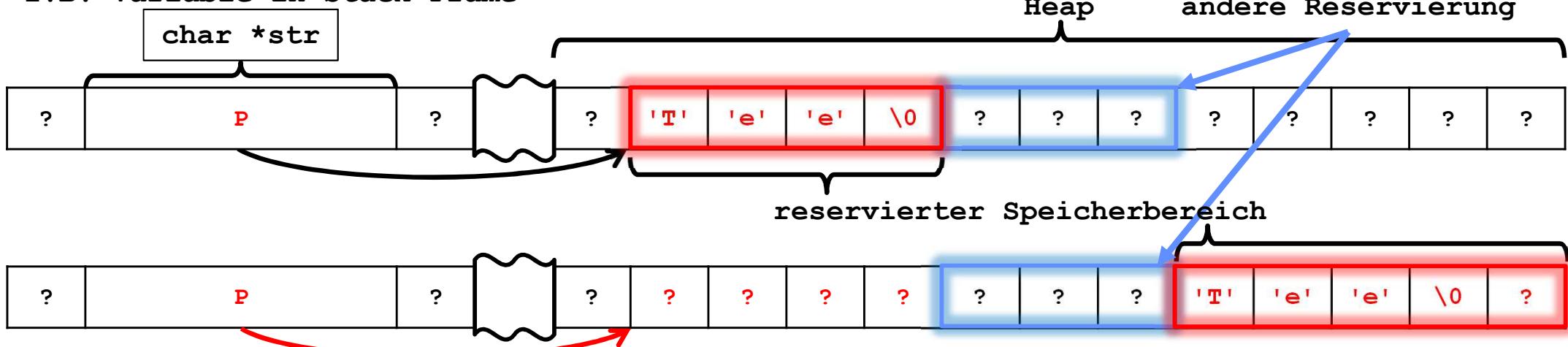
- Rückgabewert von `realloc()` nicht gespeichert, Weiternutzung ungültiger Zeiger

```
char *str = malloc( strlen("Tee")+1 ) * sizeof(char) );
strcpy(str, "Tee");
...
realloc( str, (strlen("Bier")+1) * sizeof(char) ); /* Rückgabewert vergessen */

```

↳ immer abspeichern!

z.B. Variable in Stack-Frame



## Typische Fehler bei dynamischer Speicherverwaltung (3)

- **Zu kleine Reservierung:** Fehlerhafter Ausdruck zur Berechnung der notwendigen Speichermenge (z.B. nur Elementanzahl)

```
unsigned int n = ...;
int *numbers = malloc( n ); /* reserviert nur n statt n * sizeof(int) Byte */
for( i = 0 ; i < n ; i++ ) { ... numbers[i] ... }
```

- **Zu kleine Reservierung:** Speicher für Zeiger statt für den Typ, auf den gezeigt wird

```
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
List l = malloc( sizeof(List) ); /* reserviert nur Speicher für einen Zeiger */
richtig wäre sizeof(struct ListNode) oder sizeof(*l)
```

- **Zu kleine Reservierung:** Nullsymbol nicht berücksichtigt

```
char *str = malloc( strlen("Zeichenfolge") );
strcpy(str, "Zeichenfolge"); /* Speicher für Nullsymbol fehlt */
```

- **Zugriff außerhalb des reservierten Bereichs**, z.B. durch fehlerhafte Indexberechnung

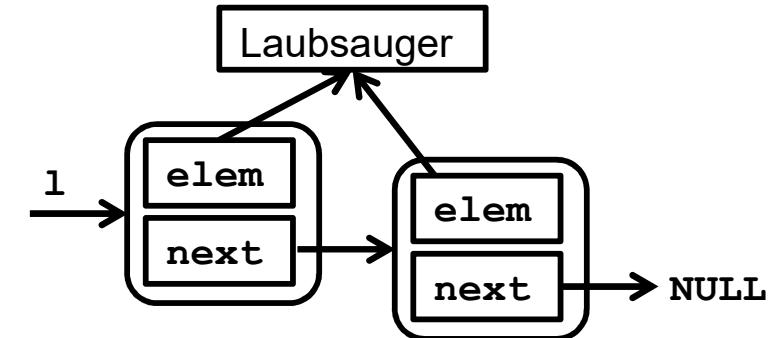
```
unsigned int n = ...;
int *numbers = malloc( n * sizeof(int) );
for( i = 0 ; i <= n ; i++ ) { ... numbers[i] ... } /* höchster Index ist n-1 */
```

## Typische Fehler bei dynamischer Speicherverwaltung (4)

- Double Free oder Freigabe statischen Speichers

```

List l = emptyList();
char *str = malloc( strlen("Laubsauger") + 1 );
strcpy(str, "Laubsauger");
l = insertAt(l, 0, str); } Beide verweisen auf selben Speicher
l = insertAt(l, 0, str);
l = removeAt(l, 0, &str);
  free(str);           /* klappt noch */
l = removeAt(l, 0, &str);
  free(str);           /* klappt nicht */
  
```



```

List l = emptyList();
char *str = "Laubsauger"; /* statischer Speicher */
l = insertAt(l, 0, str);
l = removeAt(l, 0, &str);
  free(str);           /* klappt nicht */
  
```

- Generelle Gefahr: Auswirkungen der meisten Fehler nicht deterministisch, augenscheinlich kann alles funktionieren

- Referenzzählung zur Erkennung von Speicherlecks
  - **Speicherreservierung** zu einem Datentypen zentralisieren (ideal: nur eine Funktion)
  - **Speicherfreigabe** zu einem Datentypen zentralisieren (ideal: nur eine Funktion)
  - **globale Zählvariable** je dynamisch allokiertem Datentyp
  - Zählvariable bei jeder Reservierung **erhöhen** und bei jeder Freigabe **vermindern**

```
unsigned int list_refs = 0;

List cons(List l, Element e) {
    List res = malloc( sizeof(*l) );
    list_refs++;
    ...
}

List deleteFirst(List l) {
    free(l);
    list_refs--;
    ...
}
```

```
unsigned int str_refs = 0;

String create(unsigned int len) {
    String s = malloc( sizeof(*s) );
    str_refs++;
    ...
}

String delete(String str) {
    free(str);
    str_refs--;
    ...
}

/* Hauptprogramm */
extern unsigned int list_refs;
extern unsigned int str_refs;
```

- Bisherige Implementierung verketteter Listen per **rekursiver Funktionen**
  - `length()`
  - `insertAt()`
  - `removeAt()`
  - `getAt()`
- Rekursive Verarbeitungen werden von der Maschine zur Laufzeit in bekannter Weise über den **Aufrufstapel** umgesetzt
- Nachfolgend beispielhafte Ausführung für den Aufruf

```
insertAt(1, 2, "abc")
```

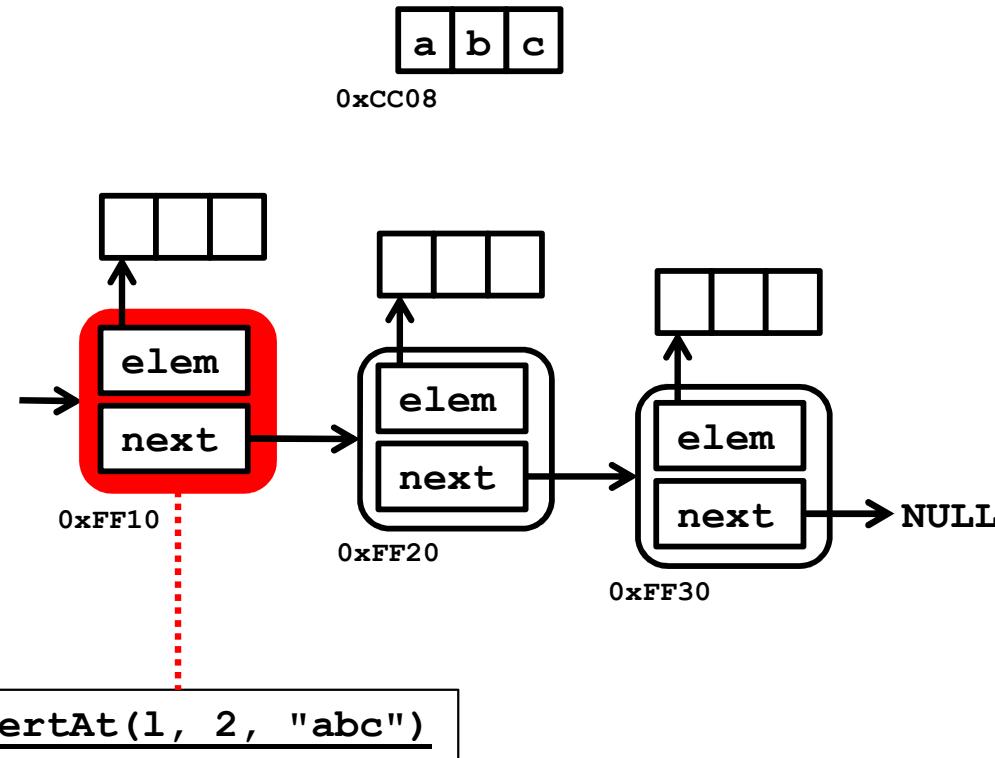
auf einer verketteten Liste **1** mit **3 Knoten**, also Einfügen des Strings **"abc"** an Indexposition **2** (als dritter Knoten)

## Aufrufstapel & rekursive Listenverarbeitung (2)

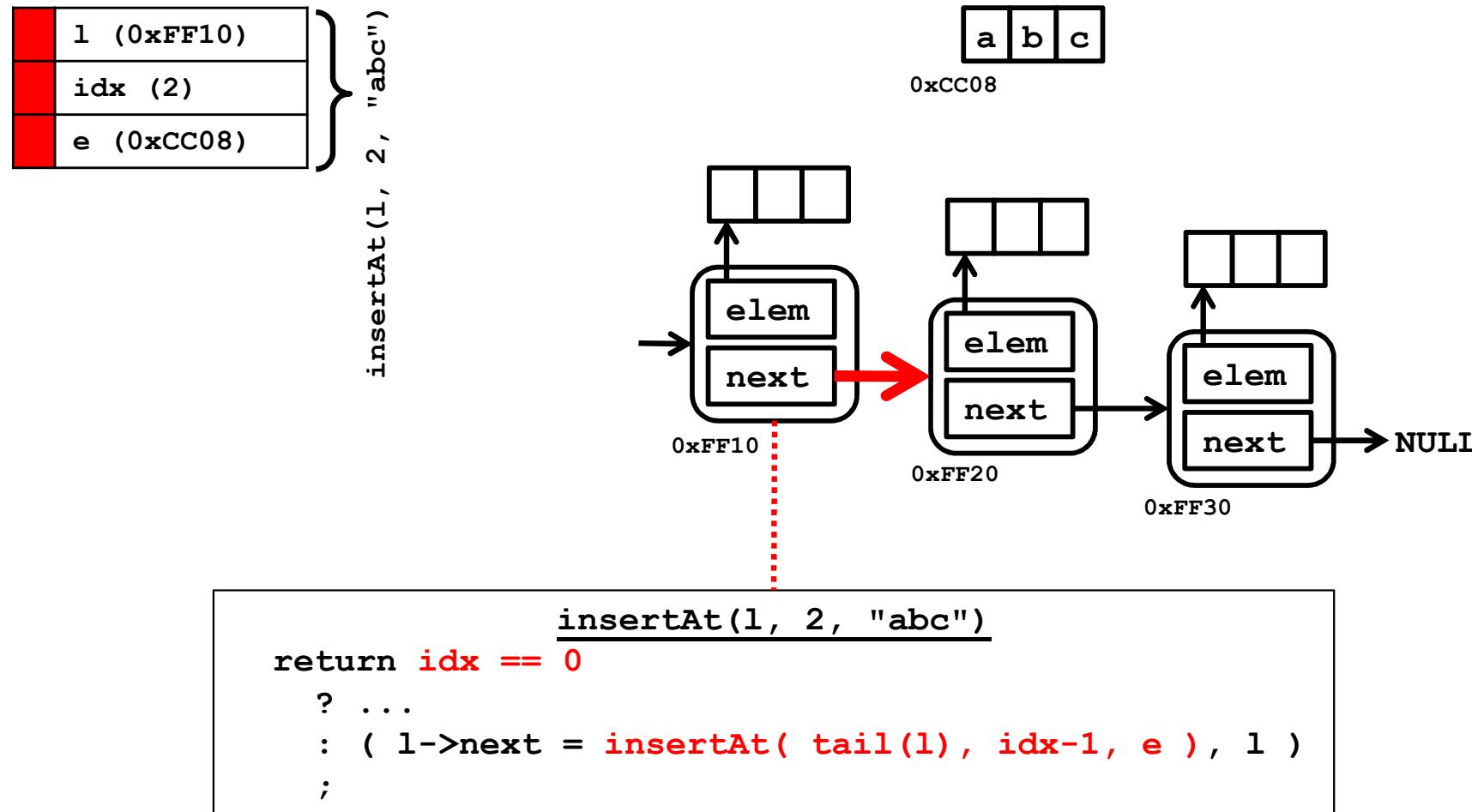
Stackframe:

1 (0xFF10)	}
idx (2)	
e (0xCC08)	

insertAt(1, 2, "abc")



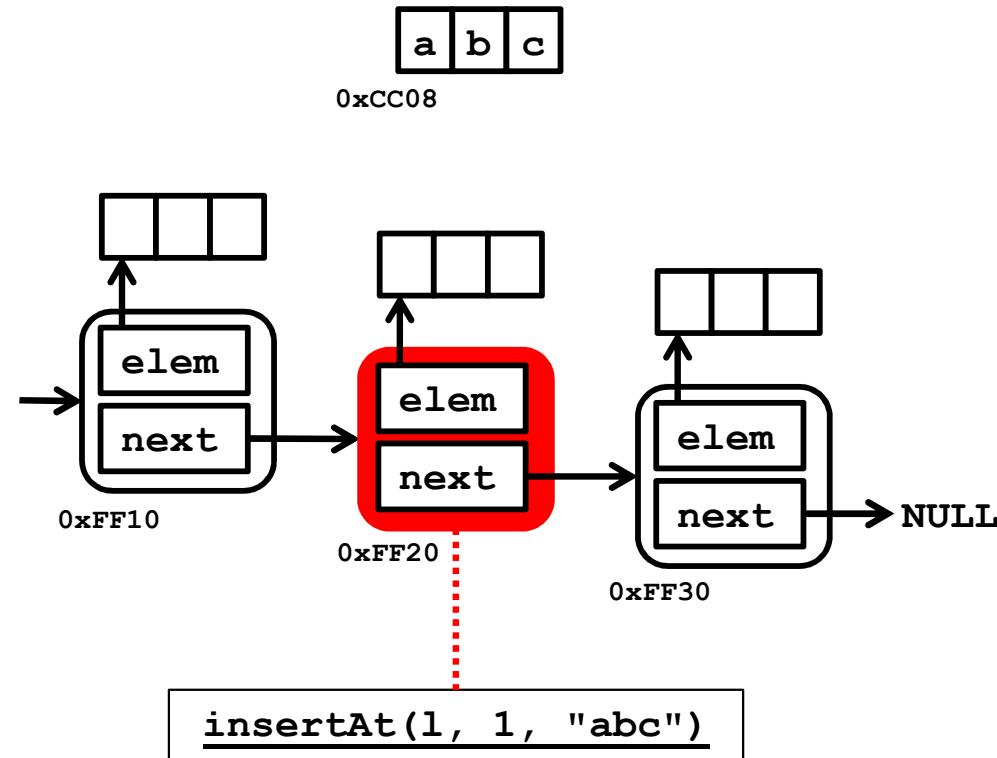
## Aufrufstapel & rekursive Listenverarbeitung (2)



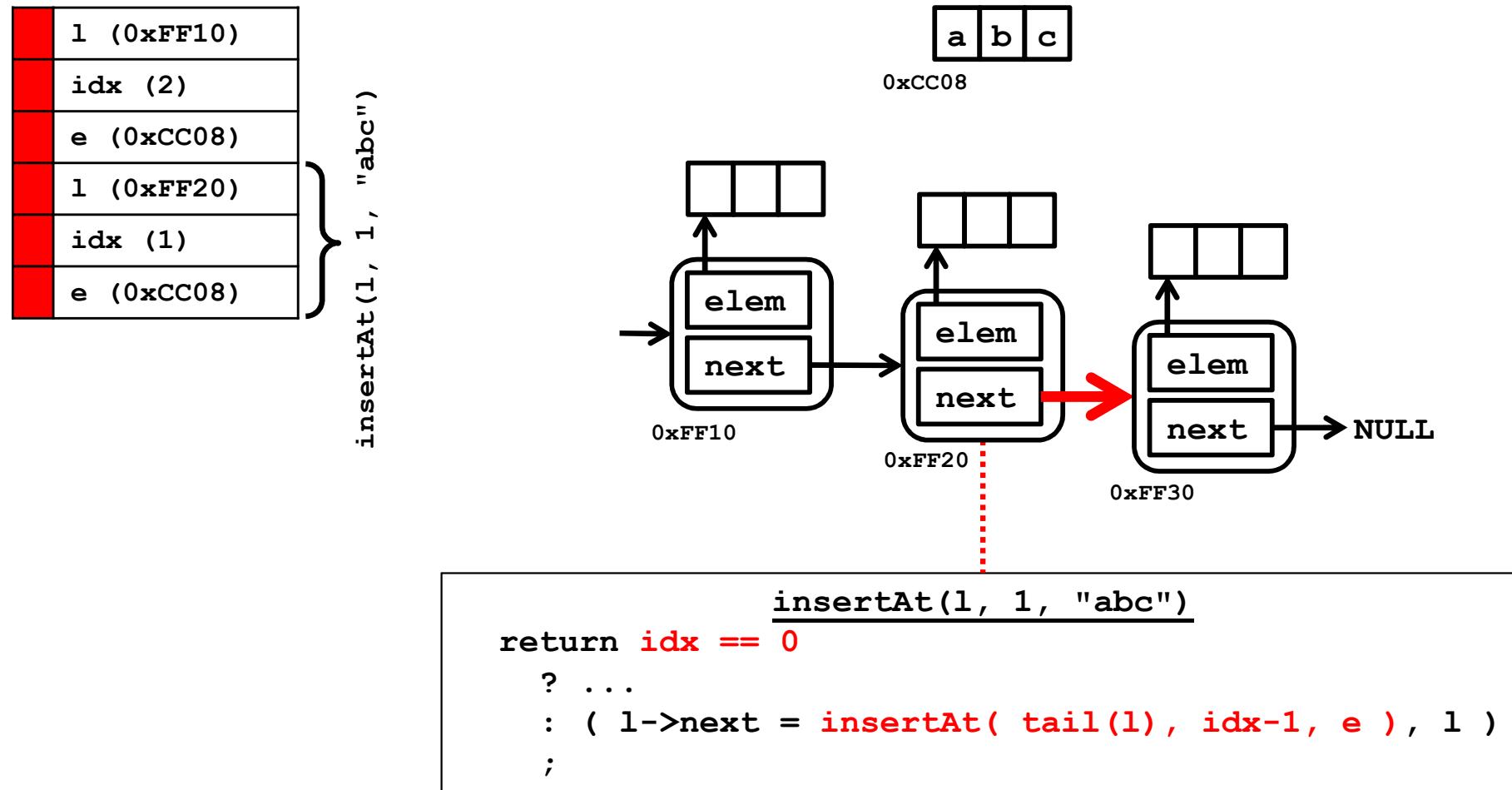
## Aufrufstapel & rekursive Listenverarbeitung (2)

1 (0xFF10)
idx (2)
e (0xCC08)
1 (0xFF20)
idx (1)
e (0xCC08)

} insertAt(1, 1, "abc")



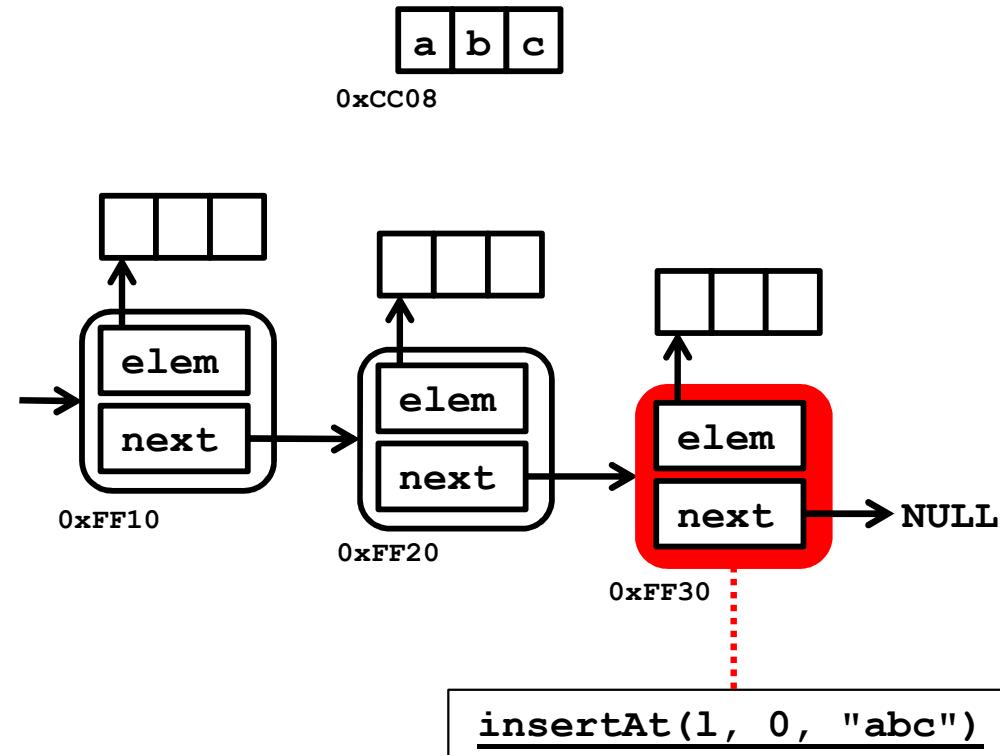
## Aufrufstapel & rekursive Listenverarbeitung (2)



## Aufrufstapel & rekursive Listenverarbeitung (2)

1 (0xFF10)
idx (2)
e (0xCC08)
1 (0xFF20)
idx (1)
e (0xCC08)
1 (0xFF30)
idx (0)
e (0xCC08)

} insertAt(1, 0, "abc")

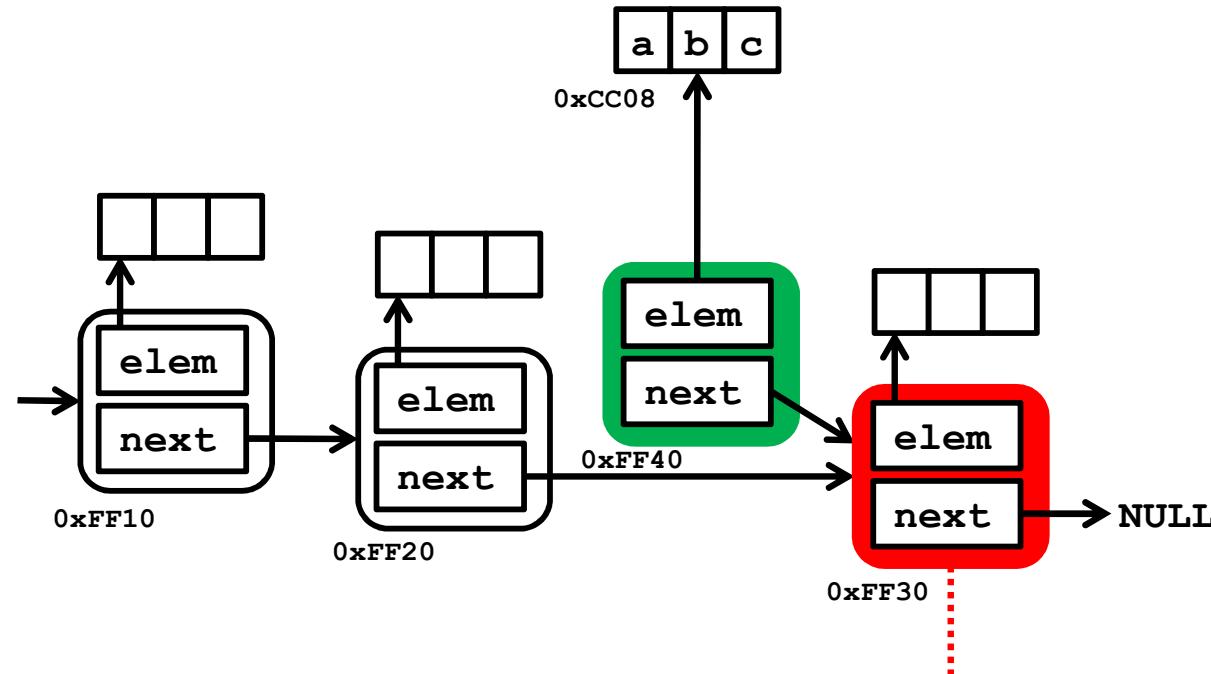


## Aufrufstapel & rekursive Listenverarbeitung (2)

1 (0xFF10)
idx (2)
e (0xCC08)
1 (0xFF20)
idx (1)
e (0xCC08)
1 (0xFF30)
idx (0)
e (0xCC08)
1 (0xFF30)
e (0xCC08)
<b>res (0xFF40)</b>

}

`cons(1, "abc")`



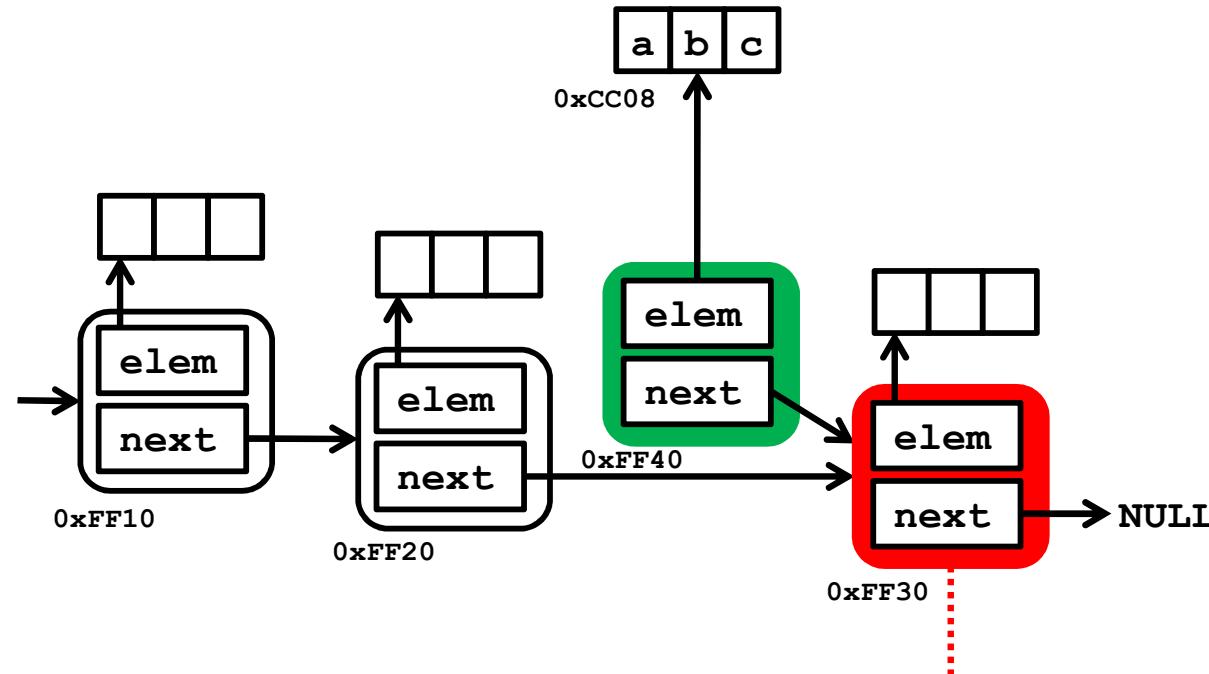
```

insertAt(l, 0, "abc")
return idx == 0
? cons(l, e)
: ...
;
```

## Aufrufstapel & rekursive Listenverarbeitung (2)

l (0xFF10)
idx (2)
e (0xCC08)
l (0xFF20)
idx (1)
e (0xCC08)
l (0xFF30)
idx (0)
e (0xCC08)

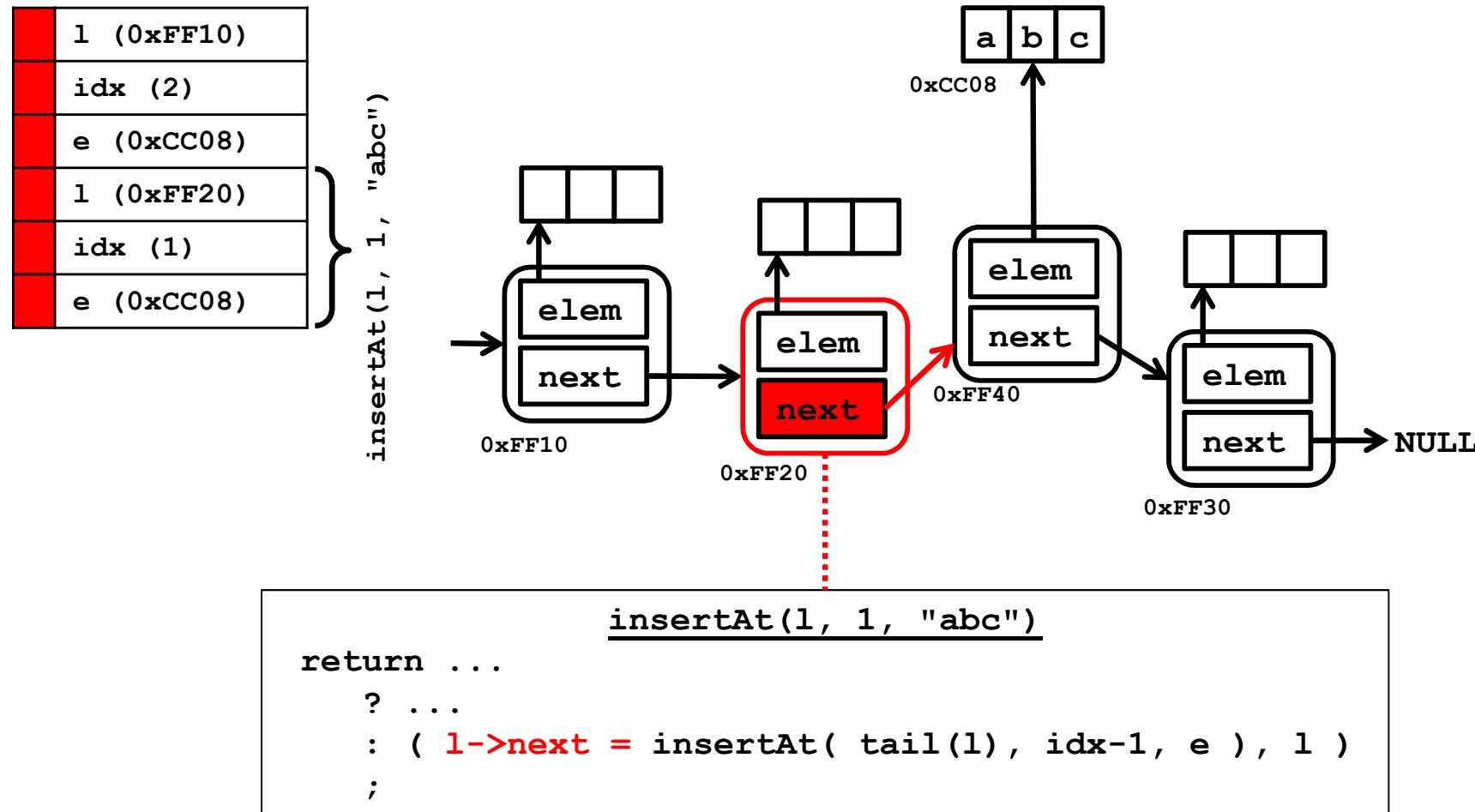
} insertAt(1, 0, "abc")



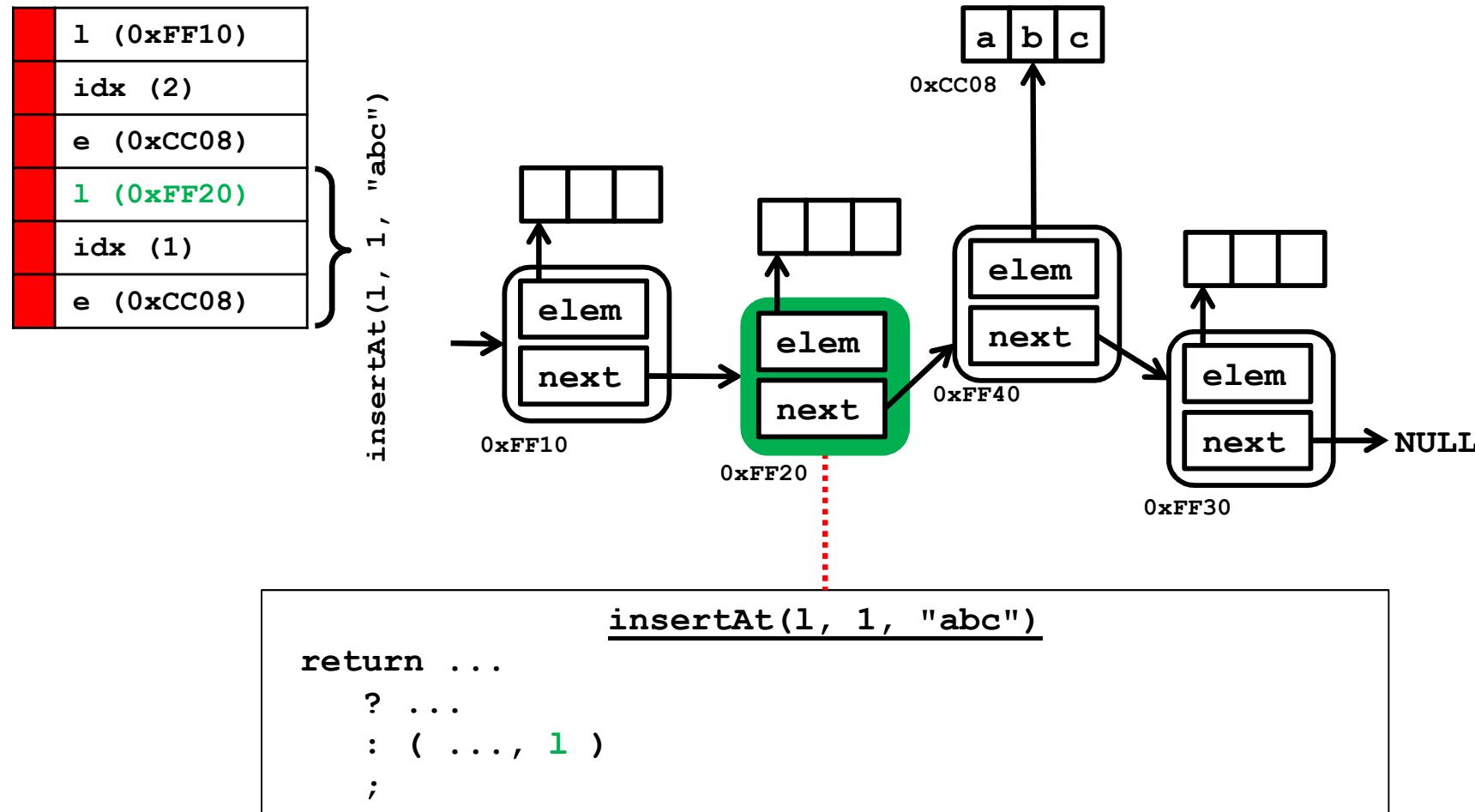
```

insertAt(l, 0, "abc")
return idx == 0
? cons(l, e)
: ...
;
```

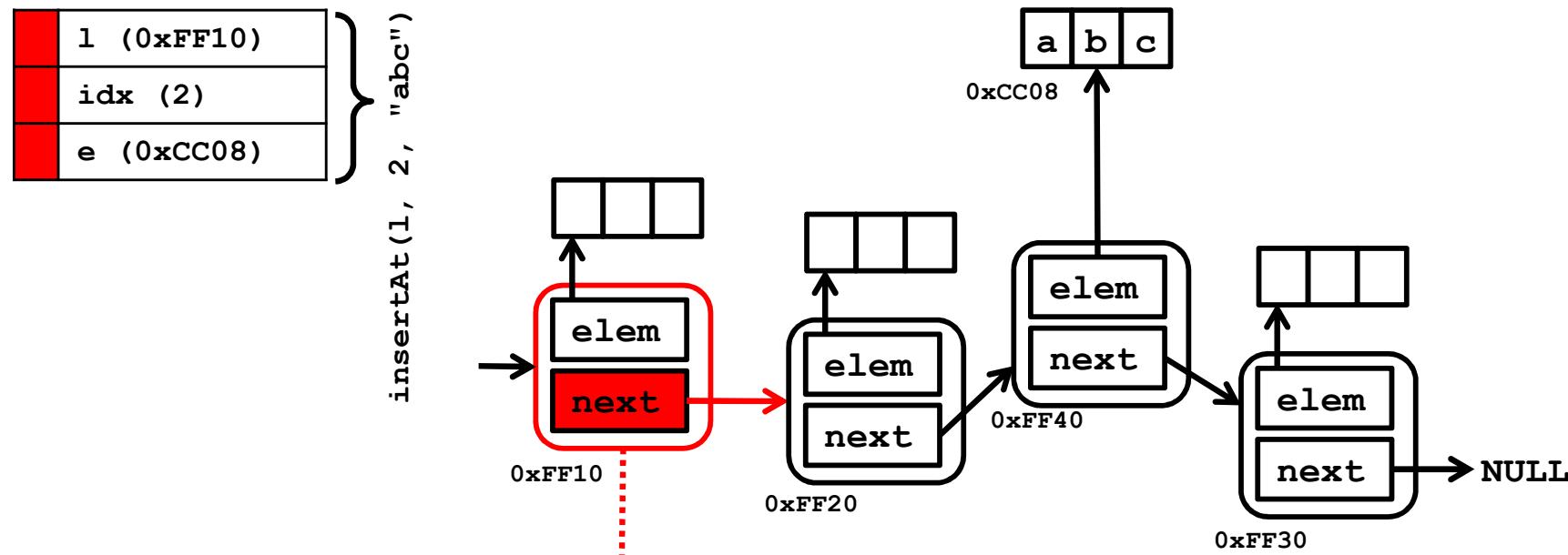
## Aufrufstapel & rekursive Listenverarbeitung (2)



## Aufrufstapel & rekursive Listenverarbeitung (2)



## Aufrufstapel & rekursive Listenverarbeitung (2)



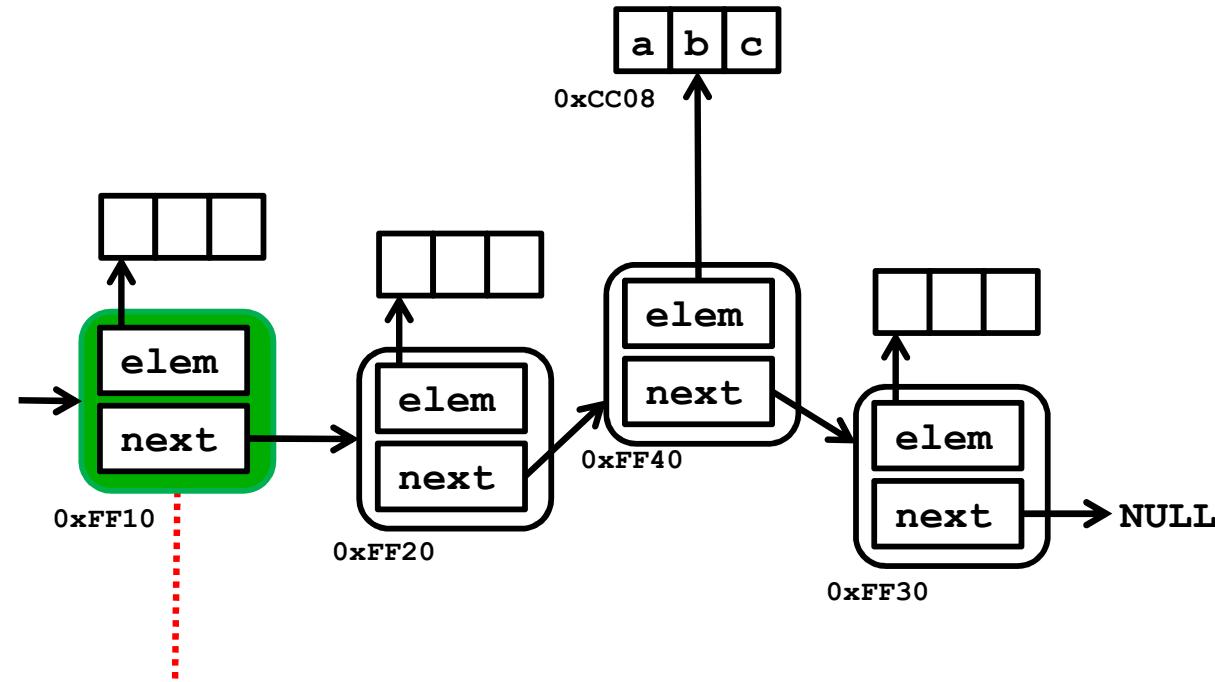
```

insertAt(1, 2, "abc")

return ...
? ...
: ( l->next = insertAt( tail(l), idx-1, e ), l )
;
  
```

## Aufrufstapel & rekursive Listenverarbeitung (2)

1 (0xFF10)
idx (2)
e (0xCC08)



```

insertAt(1, 2, "abc")

return ...
? ...
: ( ..., 1 )
;
  
```

- **Performance?**

z.B. 32.000 Knoten **einfügen** (immer am Listenende) und wieder **löschen** (immer vom Listenende)

```
int i = 0;
List l = emptyList();
for( i = 0 ; i < 32000 ; i++ ) {
    l = insertAt(l, i, ...);
}
for( i = 0 ; i < 32000 ; i++) {
    char *str = NULL;
    l = removeAt(l, (31999-i), &str);
}
```

- **Problemgröße?**

```
int i = 0;
List l = emptyList();
for( i = 0 ; i < 64000 ; i++ ) {
    l = insertAt(l, i, ...);
}
```

- **Lösungsmöglichkeit: Iterative Implementierung**

# Verkettete Listen – Iterative Implementierung (1)

```

typedef char           *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
  
```

```

Element head(List l);
List tail(List l);
int isEmpty(List l);
List emptyList();
List cons(List l, Element e);
List deleteFirst(List l);
  
```

- **Listenlänge**

```

unsigned int length(List l) {
    unsigned int res = 0;
    for( res = 0 ; !isEmpty(l) ; l = l->next ) res++;
    return res;
}
  
```

- **Einfügen in eine Liste**

```

List insertAt(List l, unsigned int idx, Element e) {
    List run = l;
    if( idx == 0 ) return cons(l, e);
    for( ; idx > 1 ; idx-- ) {
        assert( !isEmpty(run) );
        run = run->next;
    }
    run->next = cons(run->next, e);
    return l;
}
  
```

# Verkettete Listen – Iterative Implementierung (2)

```
typedef char *Element;
typedef struct ListNode *List;
struct ListNode { Element elem; List next; };
```

- **Zugriff auf ein Listenelement**

```
Element getAt(List l,
              unsigned int idx) {
    for( ; idx > 0 ; idx-- ) {
        assert( !isEmpty(l) );
        l = l->next;
    }
    return l->elem;
}
```

- **Löschen aus einer Liste**

```
List removeAt(List l,
               unsigned int idx,
               Element *e) {
    List run = l;
    if( idx == 0 ) {
        *e = head(l);
        return deleteFirst(l);
    }
    for( ; idx > 1 ; idx-- ) {
        assert( !isEmpty(run) );
        run = run->next;
    }
    *e = head(run->next);
    run->next = deleteFirst(run->next);
    return l;
}
```

## <AusdruckL> , <AusdruckR>

- Zusammenfassung von zwei Ausdrücken zu einem einzigen Ausdruck, so dass die enthaltenen Ausdrücke von links nach rechts ausgewertet werden
  - AusdruckL** wird ausgewertet und das **Ergebnis vergessen**
  - AusdruckR** wird ausgewertet zum **Ergebnis des Komma-Ausdrucks**

```

int a = 0, b = 0, c = 0;
a = 1;
b = 2;           Von links nach rechts ausgew.
c = a + b;           
(a = 1, b = 2, c = a + b);
c = (a = 1, b = 2, a + b);
  
```

*↳ Zuweisungen besitzen Rückg.wert*

```

char *s1 = ..., *s2 = ...;
char *l = NULL, *r = NULL;
int equals = 1;           Rückgabe d. Kommaausdrucks
for( l = s1, r = s2 ; equals && *l && *r ; l++, r++ ) {
    equals &= (*l == *r);
}
equals &= (*l == *r);
  
```

	Operand AusdruckL	Operand AusdruckR	Ergebnis
Typen	Beliebiger Typ <b>TL</b>	Beliebiger Typ <b>TR</b>	<b>TR,</b> ist <b>kein lvalue</b>
Anmerkungen	<b>Auswertung von links nach rechts</b> (AusdruckL vor AusdruckR), <b>Ergebnis von AusdruckL wird verworfen</b>		

**<AusdruckL> ? <AusdruckM> : <AusdruckR>**

- Das Ergebnis soll in Abhängigkeit von einer Bedingung durch einen von zwei alternativen Unterausdrücken bestimmt werden
  - **AusdruckL** wird zu einem Wert **c** ausgewertet und mit 0 verglichen
  - **c != 0**: **AusdruckM** wird ausgewertet und Ergebnis des Ausdrucks
  - **c == 0**: **AusdruckR** wird ausgewertet und Ergebnis des Ausdrucks

```

int      a = ...;
char *str = (a % 2 == 0)
           ? "gerade Zahl"
           : "ungerade Zahl"
;
  
```

	Operand <b>AusdruckL</b>	Operand <b>AusdruckM</b>	Operand <b>AusdruckR</b>	Ergebnis
<b>Konvertierung</b>	<b>Implizite unäre Konvertierungen</b>	ggf. <b>implizite binäre Konvertierungen</b>		
<b>Typen</b> (nach Konvertierung)	Beliebiger <b>skalarer</b> Typ <b>c</b>	<b>T</b>	<b>T</b>	<b>T, kein lvalue</b>  Zeiger auf <b>P</b> , der andere <b>NULL</b> -Zeiger <b>kein lvalue</b>

```
if( a && b || c ) { ... }
```

- Vorrang / Bindungsstärke? Bindet `&&` oder `||` stärker?

((a && b) || c) ✓  
oder  
(a && (b || c)) ✗

```
((0 && 0) || 1) == 1  
(0 && (0 || 1)) == 0
```

```
int x = a - b - c;
```

- Assoziativität? Ist die Subtraktion recht- oder links-assoziativ?

((a - b) - c) ✓  
oder  
(a - (b - c)) ✗

```
((5 - 5) - 5) == -5  
(5 - (5 - 5)) == 5
```

## Ausdrücke – Vorrang und Assoziativität (2)

Merksatz	Beispiel	entspricht...	...entspricht nicht...
<b>Unäre Operatoren binden stärker als binäre Operatoren</b>	<code>!a &amp;&amp; b</code>	<code>(!a) &amp;&amp; b</code>	<code>!(a &amp;&amp; b)</code>
<b>Post-Inkrement/Dekrement, Index und Funktionsaufruf binden stärker als die übrigen unären Operatoren</b>	<code>!a[b]</code>	<code>!(a[b])</code>	<code>(!a)[b]</code>
<b>Binäre Operatoren sind grundsätzlich links-assoziativ</b>	<code>a - b - c</code>	<code>(a - b) - c</code>	<code>a - (b - c)</code>
<b>Arithmetik bindet stärker als Shift</b>	<code>a * b &lt;&lt; c</code>	<code>(a * b) &lt;&lt; c</code>	<code>a * (b &lt;&lt; c)</code>
<b>Punkt bindet stärker als Strich</b>	<code>a * b + c</code>	<code>(a * b) + c</code>	<code>a * (b + c)</code>
<b>Shift bindet stärker als Vergleiche</b>	<code>a &lt;&lt; b &gt;= c</code>	<code>(a &lt;&lt; b) &gt;= c</code>	<code>a &lt;&lt; (b &gt;= c)</code>
<b>Vergleiche binden stärker als bitweise Operatoren</b>	<code>a &amp; b &gt;= c</code>	<code>a &amp; (b &gt;= c)</code>	<code>(a &amp; b) &gt;= c</code>
<b>Bitweise Operatoren binden stärker als logische Operatoren</b>	<code>a &amp; b    c</code>	<code>(a &amp; b)    c</code>	<code>a &amp; (b    c)</code>

## Ausdrücke – Regeln zu Vorrang und Assoziativität

nach unten abnehmende Bindungsstärke

	Assoz.	Beispiel	
<code>a[b], a(...), a.b, a-&gt;b, a++, a--</code>	links	<code>a-&gt;b++</code>	$\Leftrightarrow ((a \rightarrow b)++)$
<code>++a, --a, sizeof a, ~ a, ! a, - a, + a, &amp; a, * a</code>	rechts	<code>sizeof *a</code>	$\Leftrightarrow (\text{sizeof}(*a))$
<code>(t)a</code>	rechts		
<code>a * b, a / b, a % b</code>	links		
<code>a + b, a - b</code>	links	<code>a - b - c * d</code>	$\Leftrightarrow ((a - b) - (c * d))$
<code>a &lt;&lt; b, a &gt;&gt; b</code>	links	<code>a &lt;&lt; b &gt;&gt; c</code>	$\Leftrightarrow ((a << b) >> c)$
<code>a &lt; b, a &gt; b, a &lt;= b, a &gt;= b</code>	links	<code>a &lt; b &lt; c</code>	$\Leftrightarrow ((a < b) < c)$
<code>a == b, a != b</code>	links	<code>a &lt; b == c &lt; d</code>	$\Leftrightarrow ((a < b) == (c < d))$
<code>a &amp; b</code>	links		
<code>a ^ b</code>	links	<code>a   b   c</code>	$\Leftrightarrow ((a   b)   c)$
<code>a   b</code>	links	<code>a   b &amp; c</code>	$\Leftrightarrow (a   (b \& c))$
<code>a &amp;&amp; b</code>	links	<code>a    b    c</code>	$\Leftrightarrow ((a    b)    c)$
<code>a    b</code>	links	<code>a    b &amp;&amp; c</code>	$\Leftrightarrow (a    (b \&& c))$
<code>a ? b : c</code>	rechts		
Zuweisungen (einfach, zusammengesetzt)	rechts	<code>a = b = c * d</code>	$\Leftrightarrow a = (b = (c * d))$
Komma-Ausdrücke <code>a, b</code>	links	<code>a, b, c</code>	$\Leftrightarrow ((a, b), c)$

# Ausdrücke – Typische Verwirrungen (1)

- \*ptr++

- **ptr** sei ein Zeiger auf den Anfang eines Feldes von Ganzzahlen, z.B. { 0, 0, 0 }
- ✗ ▪ **(\*ptr)++:** Dereferenziere **ptr** und inkrementiere die erste Zahl => { 1, 0, 0 }
- ✓ ▪ **\* (ptr++):** Setze **ptr** um eine Zahl weiter (zeigt anschließend auf die zweite Zahl des Feldes), dereferenziere den bisherigen Zeiger und vergesse das Ergebnis => { 0, 0 }

- \*argv[1]

- **Beispiel:** **argv** = { "abc", "def", "ghi" }
- ✗ ▪ **(\*argv)[1]:** Liefert das zweite Zeichen der ersten Zeichenfolge ('b')
- ✓ ▪ **\* (argv[1]):** Liefert das erste Zeichen der zweiten Zeichenfolge ('d')

- a & b statt a && b: Funktioniert aufgrund des schwachen Typsystems, aber...

- ... Ergebnis für **&** hängt von Bitpositionen ab
- ... keine verkürzte Auswertung mit garantierter Reihenfolge beim **&**

```
int a = 1, b = 2;
if( a & b ) { ... } /* 0 bzw. false */
if( a && b ) { ... } /* 1 bzw. true */
(fprintf(stdout, "Left"), 0) & (fprintf(stdout, "Right"), 0); /* > LeftRight */
(fprintf(stdout, "Left"), 0) && (fprintf(stdout, "Right"), 0); /* > Left */
```

## Ausdrücke – Typische Verwirrungen (2)

- **a == b == c**

- $(a == b) == c$ : Berechnet **nicht**, ob die drei Werte identisch sind
- Wird nur aufgrund des schwachen Typsystems akzeptiert (Vergleich Boolean mit Ganzzahl)

```
int a = 1, b = 2, c == 0;
if( a == b == c ) { ... }
/* (a == b) == c
=> (1 == 2) == 0
=> 0 == 0
=> 1 bzw. true */
```

- **! a || b** bedeutet:  $(!a) \mid\mid b$ , nicht  $!(a \mid\mid b)$

- **&a->b** bedeutet:  $\&(a->b)$ , d.h. Berechnung der Adresse der Komponente **b** des Strukturwertes **a**      *(Pfeiloperator: a muss ein Zeiger sein)*

- **\*ps.elem** bedeutet:  $*(\text{ps}.\text{elem})$ , nicht  $(*\text{ps}).\text{elem}$  (*ps kein Zeiger*)

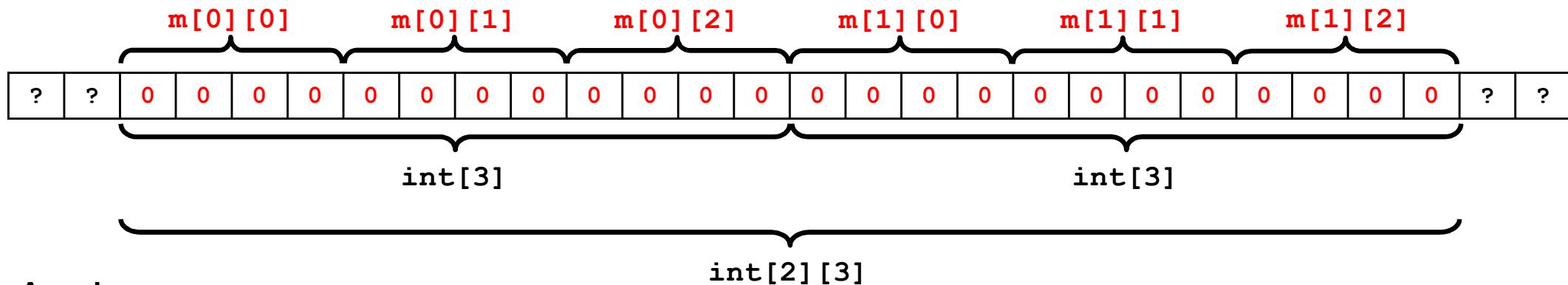
- **a << 2 + b >> 4** bedeutet  $a <\> (2 + b) >\> 4$

```
int a = 1, b = 63;                                /* 1: 0000 0001, 63: 0011 1111 */
(a << 2) + (b >> 4)                            => 4 + 3 => 7
(a << (2 + b)) >> 4                           => (a << 65) >> 4 => 0 bzw. undefined */
```

- **Beispiel:** Matrix aus Zeilen und Spalten, Elementtyp `int`
- Matrix per **Arrays (statische Speicherverwaltung)**
  - Matrixdimensionen werden per Konstanten `COLS` und `ROWS` zur **Übersetzungszeit** festgelegt
  - Arrays können geschachtelt werden (Array von Arrays), die **innersten Klammern** geben die **äußerste** Schachtelungsebene an

```
int m[ROWS][COLS] = { { 0 } };
```

- z.B. für `ROWS 2` und `COLS 3`:



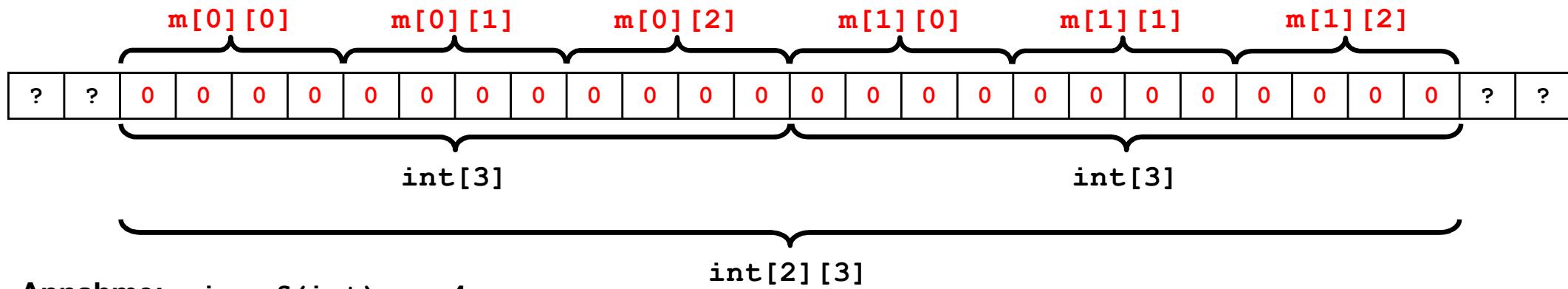
- Matrix per Arrays (statische Speicherverwaltung) (Forts.)

```
int m[ROWS][COLS] = { { 0 } };
```

- Wie gewohnt: An Funktionen wird anstelle des Arrays nur ein Zeiger übergeben
- Aber: Zeigerkonvertierung bezieht sich nur auf äußerste Arrayschachtelung

```
int foo( int m[ROWS][COLS] ) {
  ...
}
```

```
int foo( int (*m)[COLS] ) {
/* sizeof(m)    => 8 */
/* sizeof(*m)   => 12 */
/* sizeof(**m)  => 4 */
...
}
```



- Matrix per Arrays (statische Speicherverwaltung) (Forts.)

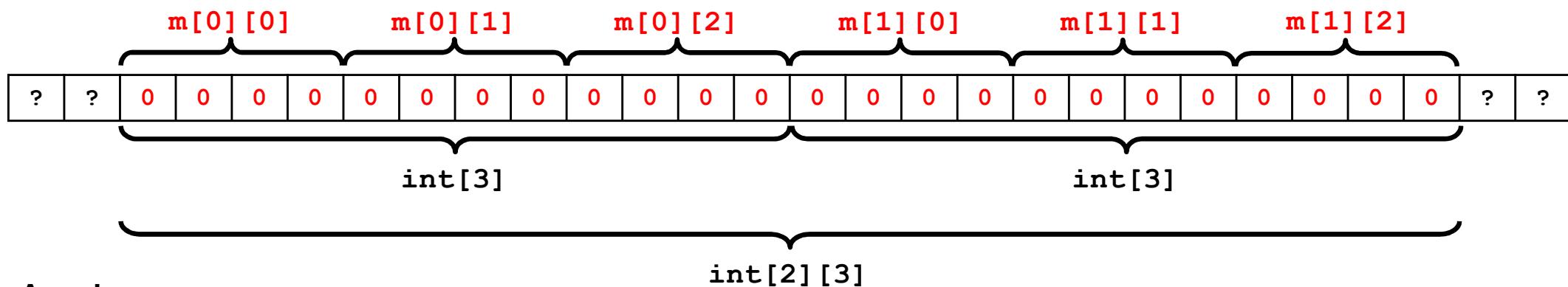
```

int m[ROWS][COLS] = { { 0 } };
int s = sum1( m );
  
```

```

int sum1( int m[ROWS][COLS] ) {
    int res = 0;
    int col = 0, row = 0;
    for( row = 0 ; row < ROWS ; row++ ) {
        for( col = 0 ; col < COLS ; col++ ) {
            res += m[row][col];
        }
    }
    return res;
}
  
```

**Beispiel:**  
 Summenbildung

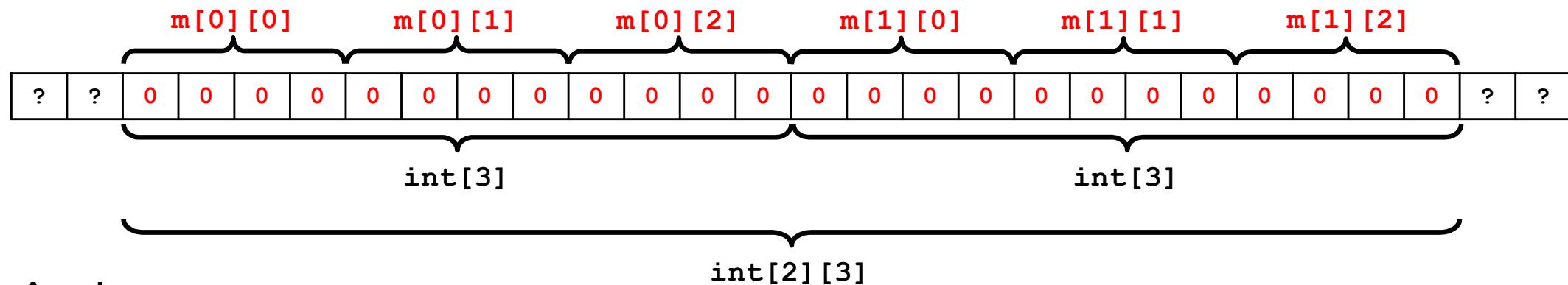
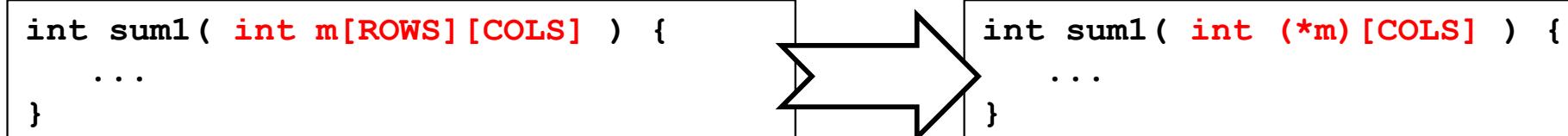


## Mehrdimensionale Felder (4)

- Matrix per Arrays (statische Speicherverwaltung) (Forts.)
  - Prinzipiell auch dynamisch allokierbar:

```

int (*m) [COLS] = calloc( sizeof(int[COLS]), ROWS );
int s = sum1( m );
  
```

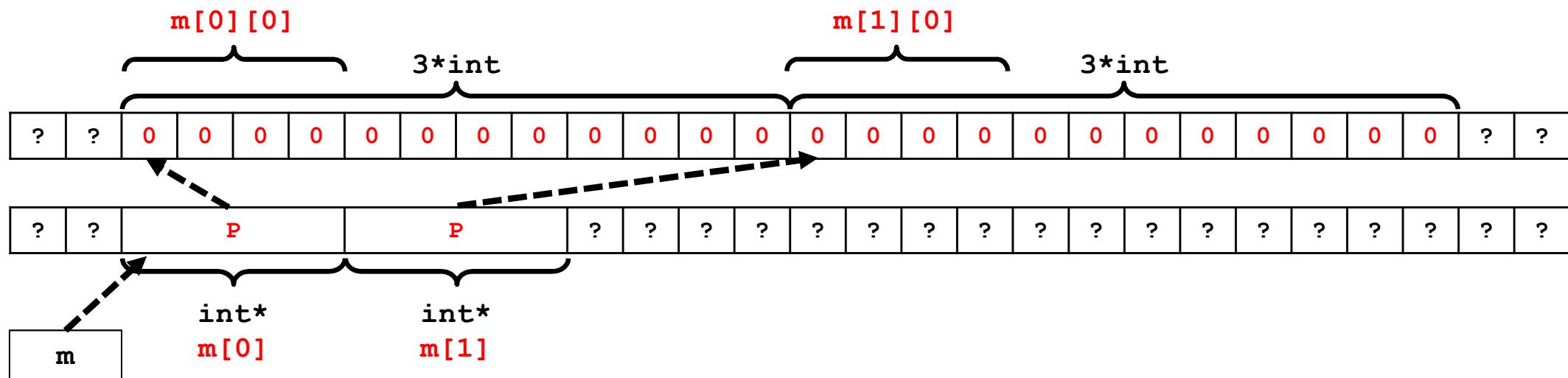


- Matrix per **dynamischer Speicherverwaltung**
  - Allokation eines Feldes von `rows` Zeilenzeigern, je Zeilenzeiger Allokation eines Feldes von `cols` Ganzzahlen (eine je Spalte)

```

int **m = malloc( sizeof(int*) * rows );
for( row = 0 ; row < rows ; row++ ) {
  m[row] = calloc( sizeof(int), cols );
}
  
```

z.B. für `rows == 2` und `cols == 3`:



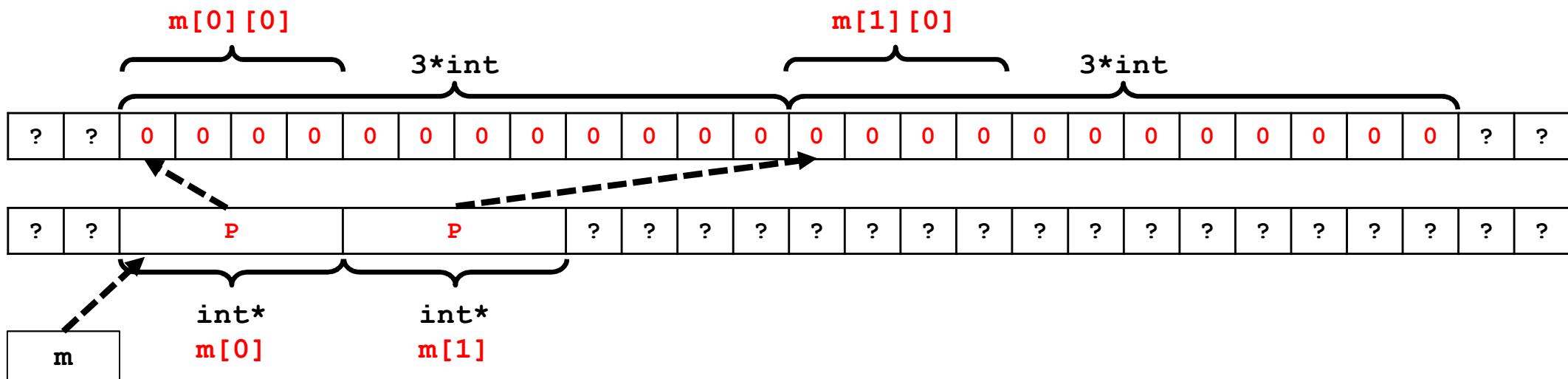
- Matrix per **dynamischer Speicherverwaltung** (Forts.)

```

int sum2( int **m, unsigned int rows, unsigned int cols ) {
    int res = 0;
    unsigned int col = 0, row = 0;
    for( row = 0 ; row < rows ; row++ ) {
        for( col = 0 ; col < cols; col++ ) {
            res += m[row][col];
        }
    }
    return res;
}
  
```

```

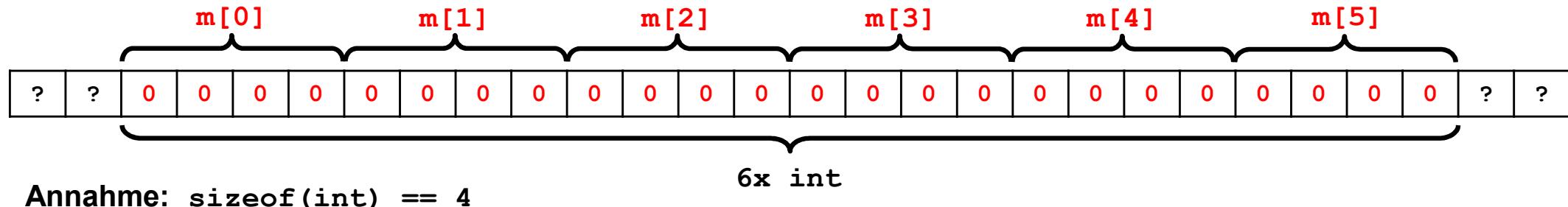
int **m = malloc( sizeof(int*) * rows );
for( row = 0 ; row < rows ; row++ ) {
    m[row] = calloc( sizeof(int), cols );
}
sum2(m);
  
```



- Matrix per **dynamischer Speicherverwaltung** (Forts.)
  - Allokation eines seriellen (eindimensionalen) Feldes von `cols * rows int`-Werten, Position im Feld wird jeweils ausgehend von Zeilen- und Spaltennummer errechnet

```
int *m = malloc( sizeof(int) * rows * cols );
```

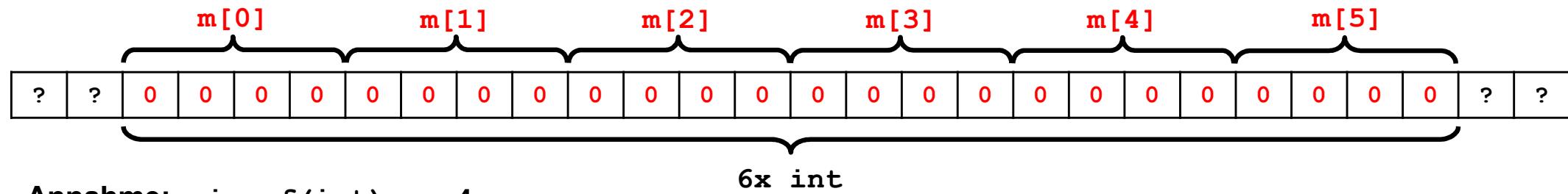
z.B. für `rows == 2` und `cols == 3`:



- Matrix per **dynamischer Speicherverwaltung** (Forts.)

```
int *m = malloc( sizeof(int) * rows * cols );
int s = sum3( m );
```

```
int sum3( int *m, unsigned int rows, unsigned int cols ) {
    int res = 0;
    unsigned int col = 0, row = 0;
    for( row = 0 ; row < rows ; row++ ) {
        for( col = 0 ; col < cols ; col++ ) {
            res += m[ row*cols+col ];
        }
    }
    return res;
}
```



**Annahme: `sizeof(int) == 4`**

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)
10. Ausdrücke 2: Konvertierungen, Vergleich / Logik / Arithmetik
11. Ausdrücke 3: Bit-Operationen
12. C-Präprozessor
13. Strukturierte Datentypen
14. Dynamische Speicherverwaltung
15. Fließkomma-Zahlen

- Aufteilung in signifikante Bits (**Mantisse M**) und Exponenten-Bits (**Exponent e**), wobei der Exponent  $e$  die Position des Kommas steuert, d.h. die Wertigkeit der Mantissen-Bits um  $e$  Stellen verschiebt

$$\text{Wert} = \text{Vorzeichen VZ} * b^e * M \quad \text{mit} \quad e_{\min} \leq e \leq e_{\max}$$

- Negative Exponenten** verschieben die Mantisse zu **kleineren Zahlen** (mehr Nachkommastellen), **positive Exponenten** zu **größeren Zahlen**
- Um die Speicherung des Exponenten z.B. im Zweiterkomplement zu vermeiden, wird für das jeweilige Format ein **Bias** festgelegt, der den Exponenten verschiebt; tatsächlich gespeichert wird dann der positive Biased Exponent E =  $e + \text{Bias}$

- Beispiel mit Bias 4, Kodierung im Dezimalsystem:**

$$\begin{aligned} \text{VZ} &= -1, \quad M = 42, \quad E = 2, \quad e = E - \text{Bias} = -2 \\ \Rightarrow -1 * 10^{-2} * 42 &= \underline{-0,42} \end{aligned}$$

*↑4*

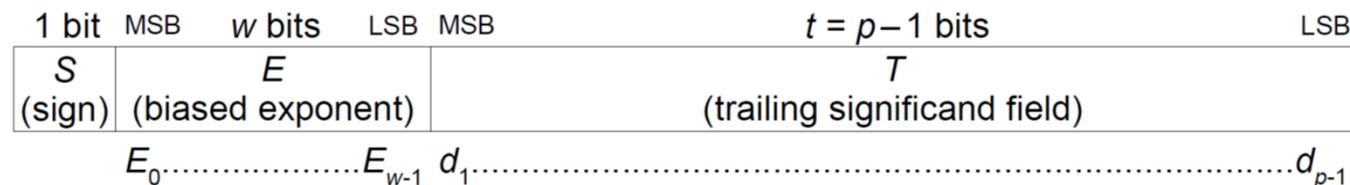
VZ	E	M		
	$10^1$	$10^0$	$10^1$	$10^0$
	10	1	10	1
-1	0	2	4	2

- Beispiel mit Bias 2, Kodierung im Binärsystem:**

$$\begin{aligned} \text{VZ} &= 1, \quad M = 3, \quad E = 1, \quad e = E - \text{Bias} = -1 \\ \Rightarrow 1 * 2^{-1} * 3 &= \underline{+1,50} \end{aligned}$$

	$2^1$	$2^0$	$2^1$	$2^0$
	2	1	2	1
1	0	1	1	1

# Für Interessierte: Grobzüge der Kodierung nach IEEE 754-2008



- **k Bit Gesamtbreite**
- **1 Bit Vorzeichen S**
- **w Bit für den Biased Exponenten E,**  
d.h. 0 **bis**  $2^w - 1$
- **tatsächlicher Exponent e = E - Bias**
  - maximaler Exponent  $e_{\max} = 2^{(w-1)} - 1$
  - **Bias =  $e_{\max}$**
  - minimaler Exponent  $e_{\min} = 1 - e_{\max}$
  - **Beispiel 8bit Exponent:**  $e_{\max} = 127$ ,  $\text{Bias} = 127$ ,  $e_{\min} = -126$
- **p Bit für die Mantisse T**
  - es werden nur  $t = p - 1$  Bits gespeichert (trailing bits)
  - das "fehlende" erste Bit  $t'$  ergibt sich implizit aus dem Exponenten
  - höchstes Bit  $t'$  entspricht Stellenwert 1, alle niedrigeren Bits sind folglich Nachkommastellen (Betrachtung **vor Anwendung des Exponenten**)

IEEE 754-2008	binary32	binary64
Breite k	32	64
Vorzeichenbits	1	
Exponentenbits	8	11
Genauigkeit p	24	53
Mantissenbits	23	52
$e_{\max}$	127	1023
Bias	127	1023
$e_{\min}$	-126	-1022

# Beispiel: 6bit-Format, 3bit Exponent, 3bit Mantisse, ohne VZ (1)

*↳ In der Realität Mantisse meist länger*

$$e_{\max}: 2^{(3-1)} - 1 = 3$$

bias: 3

$$e_{\min}: 1-3 = -2$$

$$E = 0$$

=>

$$e = e_{\min} = -2$$

=>

$t' = 0$  ("subnormale" Zahlen)

$$E = 1$$

=>

$$e = 1-\text{Bias} = 1-3 = -2$$

=>

$t' = 1$  ("normale" Zahlen)

**Die Stellenwerte der Mantisse T verstehen sich hier nach Anwendung des Exponenten e**

Exponent E			t'	Mantisse T					
$2^2$ 4	$2^1$ 2	$2^0$ 1	$2^{-2}$ $1/4$	$2^{-3}$ $1/8$	$2^{-4}$ $1/16$	$2^{-5}$ $1/32$	Dezimalwert	Auflösung	
0	0	0	0	0	0	0	0,00000	<i>↳ Abstand zwischen Zahlen</i> <b>0,03125</b>	
0	0	0	0	0	0	1	0,03125		
0	0	0	0	0	1	0	0,06250		
0	0	0	0	0	1	1	0,09375		
0	0	0	0	1	0	0	0,12500		
0	0	0	0	1	0	1	0,15625		
0	0	0	0	1	1	0	0,18750		
0	0	0	0	1	1	1	0,21875		
$2^2$ 4	$2^1$ 2	$2^0$ 1	$2^{-2}$ $1/4$	$2^{-3}$ $1/8$	$2^{-4}$ $1/16$	$2^{-5}$ $1/32$			
0	0	1	1	0	0	0	0,25000	<b>0,03125</b>	
0	0	1	1	0	0	1	0,28125		
0	0	1	1	0	1	0	0,31250		
0	0	1	1	0	1	1	0,34375		
0	0	1	1	1	0	0	0,37500		
0	0	1	1	1	0	1	0,40625		
0	0	1	1	1	1	0	0,43750		
0	0	1	1	1	1	1	0,46875		

*↑ Wenn Exponent nicht 0 auf 1 gesetzt*

## Beispiel: 6bit-Format, 3bit Exponent, 3bit Mantisse, ohne VZ (2)

$$e_{\max}: 2^{(3-1)} - 1 = 3$$

$$\text{bias: } 3$$

$$e_{\min}: 1-3 = -2$$

$$E = 2$$

=>

$$e = 2-\text{Bias} = 2-3 = -1$$

=>

$t' = 1$  (normale Zahlen)

$$E = 3$$

=>

$$e = 3-\text{Bias} = 3-3 = 0$$

=>

$t' = 1$  (normale Zahlen)

Die Stellenwerte der Mantisse T verstehen sich  
hier nach Anwendung des Exponenten e

Exponent E			t'	Mantisse T					
$2^2$ 4	$2^1$ 2	$2^0$ 1	2 <sup>1</sup> 1/2	2 <sup>2</sup> 1/4	2 <sup>3</sup> 1/8	2 <sup>4</sup> 1/16	Dezimalwert	Auflösung	
0	1	0	1	0	0	0	0,5000	0,0625	
0	1	0	1	0	0	1	0,5625		
0	1	0	1	0	1	0	0,6250		
0	1	0	1	0	1	1	0,6875		
0	1	0	1	1	0	0	0,7500		
0	1	0	1	1	0	1	0,8125		
0	1	0	1	1	1	0	0,8750		
0	1	0	1	1	1	1	0,9375		
$2^2$ 4	$2^1$ 2	$2^0$ 1	2 <sup>0</sup> 1	2 <sup>-1</sup> 1/2	2 <sup>-2</sup> 1/4	2 <sup>-3</sup> 1/8			
0	1	1	1	0	0	0	1,000	0,125	
0	1	1	1	0	0	1	1,125		
0	1	1	1	0	1	0	1,250		
0	1	1	1	0	1	1	1,375		
0	1	1	1	1	0	0	1,500		
0	1	1	1	1	0	1	1,625		
0	1	1	1	1	1	0	1,750		
0	1	1	1	1	1	1	1,875		

## Beispiel: 6bit-Format, 3bit Exponent, 3bit Mantisse, ohne VZ (3)

$$e_{\max}: 2^{(3-1)} - 1 = 3$$

bias: 3

$$e_{\min}: 1-3 = -2$$

$$E = 4$$

=>

$$e = 4-\text{Bias} = 4-3 = 1$$

=>

$t' = 1$  (normale Zahlen)

$$E = 5$$

=>

$$e = 5-\text{Bias} = 5-3 = 2$$

=>

$t' = 1$  (normale Zahlen)

Die Stellenwerte der Mantisse T verstehen sich  
hier nach Anwendung des Exponenten e

Exponent E			t'	Mantisse T					
$2^2$ 4	$2^1$ 2	$2^0$ 1	2 <sup>1</sup> 2	$2^0$ 1	$2^{-1}$ 1/2	$2^{-2}$ 1/4	Dezimalwert	Auflösung	
1	0	0	1	0	0	0	2,00	0,25	
1	0	0	1	0	0	1	2,25		
1	0	0	1	0	1	0	2,50		
1	0	0	1	0	1	1	2,75		
1	0	0	1	1	0	0	3,00		
1	0	0	1	1	0	1	3,25		
1	0	0	1	1	1	0	3,50		
1	0	0	1	1	1	1	3,75		
$2^2$ 4	$2^1$ 2	$2^0$ 1	2 <sup>2</sup> 4	2 <sup>1</sup> 2	2 <sup>0</sup> 1	$2^{-1}$ 1/2			
1	0	1	1	0	0	0	4,0	0,5	
1	0	1	1	0	0	1	4,5		
1	0	1	1	0	1	0	5,0		
1	0	1	1	0	1	1	5,5		
1	0	1	1	1	0	0	6,0		
1	0	1	1	1	0	1	6,5		
1	0	1	1	1	1	0	7,0		
1	0	1	1	1	1	1	7,5		

## Beispiel: 6bit-Format, 3bit Exponent, 3bit Mantisse, ohne VZ (4)

$$e_{\max}: 2^{(3-1)} - 1 = 3$$

bias: 3

$$e_{\min}: 1-3 = -2$$

$$E = 6$$

=>

$$e = 6-\text{Bias} = 6-3 = 3 \quad (e_{\max})$$

=>

$t' = 1$  (normale Zahlen)

$E = 7, T = 0 \Rightarrow \text{unendlich}$

$E = 7, T \neq 0 \Rightarrow \text{NaN}$

Die Stellenwerte der Mantisse T verstehen sich  
hier nach Anwendung des Exponenten e

Exponent E			t'	Mantisse T					
$2^2$ 4	$2^1$ 2	$2^0$ 1	$2^3$ 8	$2^2$ 4	$2^1$ 2	$2^0$ 1	Dezimalwert	Auflösung	
1	1	0	1	0	0	0	8	1	
1	1	0	1	0	0	1	9		
1	1	0	1	0	1	0	10		
1	1	0	1	0	1	1	11		
1	1	0	1	1	0	0	12		
1	1	0	1	1	0	1	13		
1	1	0	1	1	1	0	14		
1	1	0	1	1	1	1	15		
$2^2$ 4	$2^1$ 2	$2^0$ 1							
1	1	1	1	0	0	0	unendlich		
1	1	1	1	?	?	?	NaN		

## Beispiel: 7bit-Format, 4bit Exponent, 3bit Mantisse, ohne VZ

$$e_{\max}: 2^{(4-1)} - 1 = 7$$

bias: 7

$$e_{\min}: 1-7 = -6$$

$$E = 11$$

=>

$$e = 11-7 = 4$$

=>

$t' = 1$  (normale Zahlen)

$$E = 12$$

=>

$$e = 12-7 = 5$$

=>

$t' = 1$  (normale Zahlen)

Die Stellenwerte der Mantisse T verstehen sich  
hier nach Anwendung des Exponenten e

Exponent E				t'	Mantisse T					
$2^3$ 8	$2^2$ 4	$2^1$ 2	$2^0$ 1	$2^4$ 16	$2^3$ 8	$2^2$ 4	$2^1$ 2	Dezimalwert	Auflösung	
1	0	1	1	1	0	0	0	16	$\hookrightarrow$ Kann z.B. nur gerade Zahlen darstellen	
1	0	1	1	1	0	0	1	18		
1	0	1	1	1	0	1	0	20		
1	0	1	1	1	0	1	1	22		
1	0	1	1	1	1	0	0	24		
1	0	1	1	1	1	0	1	26		
1	0	1	1	1	1	1	0	28		
1	0	1	1	1	1	1	1	30		
$2^3$ 8	$2^2$ 4	$2^1$ 2	$2^0$ 1	$2^5$ 32	$2^4$ 16	$2^3$ 8	$2^2$ 4	Dezimalwert	Auflösung	

1	1	0	0	1	0	0	0	32	4
1	1	0	0	1	0	0	1	36	
1	1	0	0	1	0	1	0	40	
1	1	0	0	1	0	1	1	44	
1	1	0	0	1	1	0	0	48	
1	1	0	0	1	1	0	1	52	
1	1	0	0	1	1	1	0	56	
1	1	0	0	1	1	1	1	60	

# Charakteristika der Fließkomma-Typen in C

- Drei Typen in C (aufsteigende Genauigkeit, müssen sich aber nicht unterscheiden): **float**, **double**, **long double**
- Implementationsabhängige Merkmale der Fließkommatypen in **float.h**

Konstante	Bedeutung	Mindestanforderung
<b>FLT_EPSILON</b>	Jeweils kleinster Wert $x$ , so dass $1.0 + x$ repräsentierbar ist	$10^{-5}$
<b>DBL_EPSILON</b>		$10^{-9}$
<b>LDBL_EPSILON</b>	$(1.0 + x \neq 1.0)$	$10^{-9}$
<b>FLT_DIG</b>	Dezimale Genauigkeit, d.h. Anzahl repräsentierbarer signifikanter dezimaler Stellen	6
<b>DBL_DIG</b>		10
<b>LDBL_DIG</b>		10
<b>FLT_MANT_DIG, DBL_...</b>	Anzahl (binärer) Mantissenstellen	keine
<b>FLT_MIN_EXP, DBL_...</b>	Kleinster Exponent $e_{\min}$	keine
<b>FLT_MAX_EXP, DBL_...</b>	Größter Exponent $e_{\max}$	keine
<b>FLT_MIN, DBL_...</b>	Kleinste positive normalisierte Zahl	$10^{-37}$
<b>FLT_MAX, DBL_...</b>	Größte repräsentierbare endliche Zahl	$10^{37}$
...	...	...

Darstellung	Syntax	Typ	Beispiele
Dezimal	<p>Ganzzahlige Ziffernfolge mit Exponent (negativ oder positiv)</p> $[0-9]+ \quad [eE] \quad [-]? \quad [0-9+] \quad <\text{FloatSuffix?}>$ <p>Ganzahlanteil . Nachkommaanteil, anschließend optional ein Exponent (negativ oder positiv)</p> $[0-9]+ \quad [\cdot] \quad [0-9]+ \quad ([eE] \quad [-]? \quad [0-9+])? \quad <\text{FloatSuffix?}>$ $\quad [\cdot] \quad [0-9]+ \quad ([eE] \quad [-]? \quad [0-9+])? \quad <\text{FloatSuffix?}>$ $[0-9]+ \quad [\cdot] \quad \quad ([eE] \quad [-]? \quad [0-9+])? \quad <\text{FloatSuffix?}>$	double	42e0 2E-3  42.42 42. .42 4.2e4 1.0E-3

- **<FloatSuffix?>**: Erzwingen eines anderen Typen als **double**
  - **Suffix f/F:** `float` (*in d. Regel weniger Genauigkeit als double*) **Beispiel:** `42F` (-> `float`)
  - **Suffix l/L:** `long double` (*i.d.R. höher Genauigkeit*) **Beispiel:** `42L` (-> `long double`)
- Die Implementierung wählt zur Repräsentation im jeweiligen Typen entweder...
   
**Beispiel:** Konstante `33.2`, repräsentierbare Werte: ..., `32`, `32.5`, `33`, `33.5`, `34`, ...
  - ... den nächstgelegenen repräsentierbaren Wert (`33`) **oder**
  - ... ausgehend vom nächstgelegenen Wert den nächsthöheren Wert (`33.5`) **oder**
  - ... ausgehend vom nächstgelegenen Wert den nächstniederigeren Wert (`32.5`)
- Ergebnis bei Verlassen des Wertebereichs undefined

## ▪ Addition / Subtraktion / Multiplikation / Division

- Fließkomma- oder **signed** Ganzzahl-Typ: Überlauf möglich, undefiniertes Verhalten
- Rundung des Ergebnisses bei Fließkommazahlen implementationsabhängig
- **unsigned** Ganzzahl-Typ: Mathematisches Ergebnis wird **Modulo**  $2^n$  gerechnet (wobei  $n$  die Anzahl Bits des Typen ist) => erhält die repräsentierbaren Bits des Ergebnisses, formal kein Überlauf (definiertes Verhalten)

**Beispiel:** Fingierte Addition von 200 und 200 in **unsigned char**

$$\begin{array}{r} 1100 \ 1000 \ (200) \\ + \ 1100 \ 1000 \ (200) \\ \hline 1 \ 1001 \ 0000 \ (400) \% 256 \\ \textcolor{red}{1001 \ 0000 \ (144)} \end{array}$$

- Verhalten bei **Division durch Null** undefiniert
- Modulo-Operation % nur auf integralen Typen möglich
- **Division und Modulo** stehen in Zusammenhang, es gilt stets:  
 $(a/b) * b + a \% b == a$ , wenn  $a/b$  repräsentierbar ist; Vorzeichen des Modulo bei Division negativer Zahlen ist implementationsabhängig

## Probleme mit Fließkommaarithmetik (1)

- Genauigkeit eingeschränkt und abhängig von der Größenordnung der Zahlen, d.h. Darstellung wird für größere Zahlen ungenauer
- Nicht jede Zahl exakt repräsentierbar  
(sowohl Literale als auch Ergebnisse von Operationen)
  - **Beispiel:** 0,1 im Binärsystem nicht exakt repräsentierbar

```

float flt = 0.1;
fprintf(stdout, "flt: %.5f\n", fp); /* flt: 0.10000 */
fprintf(stdout, "flt: %.10f\n", fp); /* flt: 0.1000000015 */
  
```

- **Beispiel:** Ergebnis einer Addition nicht exakt repräsentierbar

<pre> float flt = 8388608;       /* flt: 8388608.00 */       flt += 1.0f;       /* flt: 8388609.00 */     </pre>
--

<pre> float flt = 16777216;       /* flt: 16777216.00 */       flt += 1.0f;       /* flt: 16777216.00 */     </pre>	<i>(Bsp. immer verdoppelt)</i> <i>↳ Genauigkeit 'halb' so genau</i>
---	--

<pre> float flt = 33554432;       /* flt: 33554432.00 */       flt += 2.0f;       /* flt: 33554432.00 */       flt += 3.0f;       /* flt: 33554436.00 */     </pre>
---

- Genauigkeit eingeschränkt und abhängig von der Größenordnung der Zahlen, d.h. Darstellung wird für größere Zahlen ungenauer
- Nicht jede Zahl exakt repräsentierbar  
(sowohl Literale als auch Ergebnisse von Operationen)
- **Beispiel:** Ergebnis einer Addition nicht exakt repräsentierbar (Forts.)

```
float flt = 1.0;
      flt += FLT_EPSILON;
int equals = (flt == 1.0);           /* equals: 0 (false) */

      flt = 1.0;
      flt += (FLT_EPSILON / 2);
int equals = (flt == 1.0);           /* equals: 1 (true) */

      flt = 2.0;
      flt += FLT_EPSILON;
int equals = (flt == 2.0);           /* equals: 1 (true) */
```

Was ist da kleinste Wert  
da ich auf flt drauf  
addieren darf?

- Auswertungsreihenfolge kann Ergebnisgenauigkeit beeinflussen

- **Beispiel:** Assoziativgesetz möglicherweise nicht erfüllt (*bei untersch. Genauigkeiten*)

=>  $(a + b) + c == a + (b + c)$  ?

```
float flt1 = (2.0f + FLT_EPSILON) + FLT_EPSILON;
float flt2 = 2.0f + (FLT_EPSILON + FLT_EPSILON);
int equals = (flt1 == flt2);                                /* equals: 0 (false) */
```

- `FLT_EPSILON` wird bei zweimaliger Addition zu `2.0f` vollständig absorbiert
  - Wird hingegen zunächst `FLT_EPSILON + FLT_EPSILON` berechnet, kann das (nun größere) Ergebnis zu `2.0f` addiert werden, ohne absorbiert zu werden
- Möglicherweise unterschiedliche Ergebnisse durch unterschiedlich genaue Typen (Operanden, Berechnungstyp, Zuweisungstyp)
- **Beispiel:** `0,25` ist sowohl in `float` als auch in `double` präzise repräsentierbar, `0,1` hingegen nicht präzise und mit unterschiedlicher Genauigkeit repräsentierbar

```
int equals = (0.25F == 0.25);    /* 1 (true) */
int equals = (0.1F == 0.1);      /* 0 (false) */
```

## Probleme mit Fließkommaarithmetik (4)

- Daher einige Empfehlungen...
- Vergleiche zwischen Fließkommazahlen (mit Konstanten oder Berechnungsergebnissen) immer unter Berücksichtigung einer gewissen Unschärfe  
**Beispiel:** Test auf 0.0

```
float f = ...;
int isZero = (f == 0.0);           /* gefährlich, möglicherweise nicht exakt 0 */
int isZero = (fabs(f) < FLT_EPSILON); /* berücksichtigt Darstellungs- und
                                         ↳ generiert absolute Zahl (verwirrt VZ) Rundungsfehler */
```

- Nach Möglichkeit Operationen mit Werten stark unterschiedlicher Größenordnungen vermeiden. Außerdem: Ganzzahlen sind genauso gefährdet...

```
float flt = 33554432;           /* flt: 33554432.00 */
      flt += 2.0f;                /* flt: 33554432.00 */
```

- Nach Möglichkeit erst kleine Werte untereinander verknüpfen, dann mit großen Werten *(Berechnung am besten im einheitlichen Wertebereich (Optimalfall: double))*

```
float flt1 = (2.0f + FLT_EPSILON) + FLT_EPSILON;
float flt2 = 2.0f + (FLT_EPSILON + FLT_EPSILON);
```

# Überblick Konvertierungen (weitere Erweiterung folgt...)

Quelltyp Q	Zieltyp z	Anmerkungen
Integral (inkl. Aufzählungen)		für bestimmte Konvertierungen Informationsverlust oder undefiniertes Ergebnisse möglich
<b>Fließkomma</b>	Integral (inkl. Aufzählungen)	<b>Nachkommastellen werden abgeschnitten. Außerhalb des Wertebereichs des integralen Typen undefiniertes Verhalten.</b>
Zeiger auf <b>A</b>		Möglich, aber implementationsabhängig (nicht portabel)
<b>Integral (inkl. Aufzählungen)</b>	<b>Fließkomma</b>	<b>Innerhalb des Wertebereichs: Nächsthöherer oder nächstniedrigerer Wert. Außerhalb Wertebereich: undefiniert.</b>
<b>Fließkomma</b>		<b>Innerhalb des Wertebereichs: Nächsthöherer oder nächstniedrigerer Wert. Außerhalb Wertebereich: undefiniert.</b>
Integral (inkl. Aufzählungen)		Möglich, aber implementationsabhängig (nicht portabel)
Zeiger auf <b>A</b>	<b>Zeiger auf B</b>	
Array von <b>A</b>		Konvertierung in Zeiger auf erstes Element ( <b>B = A</b> )
Strukturtyp	Strukturtyp	Nur möglich für einen kompatiblen Strukturtypen
beliebig	<b>void</b>	Langweilig

# Überblick Zuweisungskonvertierungen (weitere Erweiterung folgt...)

Quelltyp $Q$	Zieltyp $z$	Anmerkungen
Integral (inkl. Aufzählungen)	Integral (inkl. Aufzählungen)	möglicherweise Informationsverlust oder undef. Ergebnis
<b>Fließkomma</b>		<b>möglicherweise Informationsverlust oder undef. Ergebnis</b>
Zeiger auf $A$		Nicht zulässig
<b>Integral (inkl. Aufzählungen)</b>	<b>Fließkomma</b>	<b>möglicherweise Informationsverlust oder undef. Ergebnis</b>
<b>Fließkomma</b>		<b>möglicherweise Informationsverlust oder undef. Ergebnis</b>
Integral	Zeiger auf $B$	Ausschließlich für die Zahl 0 ( <b>NULL-Zeiger</b> ) zulässig
Zeiger auf $A$		Nur zulässig für <b>NULL</b> -Zeigerkonstante oder wenn $A$ zu $B$ kompatibel (typischerweise also identischer Zeigertyp); außerdem zulässig, wenn $A$ oder $B$ der Typ <b>void*</b> ist
Array von $A$		Konvertierung in Zeiger auf erstes Element ( $A = B$ )
Strukturtyp	Strukturtyp	Nur möglich für einen kompatiblen Strukturtypen
beliebig	<b>void</b>	Nicht möglich, da <b>AusdruckL</b> nie den Typ <b>void</b> hat

# Überblick implizite binäre Konvertierungen

Reihenfolge signifikant, erste passende Regel gilt. Trifft keine Regel zu => keine Konvertierung.

ein Operand	anderer Operand	Gemeinsamer Typ	Beispiel
<code>long double</code>	arithmetischer Typ	<code>long double</code>	<code>double</code>   <code>double</code> + <code>int</code>   + <code>float</code> => <code>double</code>   => <code>double</code>
<code>double</code>	arithmetischer Typ	<code>double</code>	
<code>float</code>	arithmetischer Typ	<code>float</code>	
<code>unsigned</code> -Typ	<code>unsigned</code> -Typ	größerer der beiden <code>unsigned</code> -Typen	<code>unsigned long</code> + <code>unsigned int</code> => <code>unsigned long</code>
<code>signed</code> -Typ	<code>signed</code> -Typ	größerer der beiden <code>signed</code> -Typen	<code>long</code> + <code>int</code> => <code>long</code>
<code>unsigned</code> -Typ	<code>signed</code> -Typ von gleichem oder geringerem Rang	der <code>unsigned</code> -Typ	<code>int</code> <code>unsigned int</code> => <code>unsigned int</code>
<code>unsigned</code> -Typ	<code>signed</code> -Typ höheren Ranges, der den <code>unsigned</code> -Typen enthält	der <code>signed</code> -Typ	32bit   <code>unsigned int</code> 64bit   <code>long</code> => <code>long</code>
<code>unsigned</code> -Typ	<code>signed</code> -Typ höheren Ranges, der den <code>unsigned</code> -Typen nicht enthält	die <code>unsigned</code> -Variante des <code>signed</code> -Typen	32bit   <code>unsigned int</code> 32bit   <code>long</code> => <code>unsigned long</code>

# Agenda

---

1. Bedienungsanleitung
2. Ein erstes C-Programm: Header, Funktionen, einfache Typen und Anweisungen
3. Ganzzahlen
4. Aufzählungen
5. Ausdrücke 1: Zuweisungen, Literale, Bezeichner
6. Anwendungsbeispiele für Aufzählungen
7. Zeiger
8. Arrays
9. Anweisungen: Schleifen (while, do while, for)
10. Ausdrücke 2: Konvertierungen, Vergleich / Logik / Arithmetik
11. Ausdrücke 3: Bit-Operationen
12. C-Präprozessor
13. Strukturierte Datentypen
14. Dynamische Speicherverwaltung
15. Fließkomma-Zahlen
16. **Funktionszeiger**

## Funktionszeiger (1)

- Deklaration einer Funktion ohne Definition (Prototyp)

```
int magic(char ch, int *ptr, unsigned int count);
```

- Deklaration einer Funktion mit Definition

```
int magic(char ch, int *ptr, unsigned int count) { ... }
```

- Funktionen vergleichsweise unflexibel in Ausdrücken und im Typsystem

- kein zulässiger Typ für Parameter, Rückgabewerte, Variablen und als Operanden der meisten Operatoren

- einzige Einsatzmöglichkeit: Aufrufen

- **Mehr Flexibilität:** Ein Zeiger auf eine Funktion kann als Typ für Parameter, Rückgabewerte und Variablen genutzt werden (und als Operand einiger Operatoren)

- Spezieller Zeigertyp, der statt der Adresse eines Datenobjektes die Adresse des ausführbaren Codes einer Funktion (zeigt auf eine Funktion im Speicher) repräsentiert

## Funktionszeiger (2)

- Man beachte: Funktions- und Arraydeklarationen (`a()`, `a[]`) binden stärker als Zeigerdeklarationen (`*a`) => gegebenenfalls klammern

- **Zur Übung:** Deklaration einer Funktion, die einen **Zeiger auf int** liefert

```
int *magic(char ch, int *ptr, unsigned int count);
```

- **Noch mehr Übung:** Deklaration einer Funktion, die einen **Zeiger auf ein Array von 5 int-Werten** liefert

```
int (*magic(char ch, int *ptr, unsigned int count))[5];
```

- **Deklaration eines Zeigers auf eine Funktion**

```
int (*magic)(char ch, int *ptr, unsigned int count);
```

`magic` ist ein **Zeiger (innerste Bindung)** auf eine **Funktion** mit Rückgabetyp **int**

- **Zum Vergleich:**

```
int *magic(char ch, int *ptr, unsigned int count);
```

`magic` ist eine **Funktion (innerste Bindung)** mit Rückgabetyp **Zeiger auf int**

## Funktionszeiger (3)

- Deklaration von Variablen, Funktionsparametern und Rückgabetypen wie bei allen anderen Zeigertypen

- Funktion `add`, die zwei Ganzzahlen erwartet und eine Ganzzahl liefert:

```
int add(int l, int r) { return l + r; }
```

- Variable `binop` soll **Zeiger** auf solche Funktionen speichern:

```
int (*binop)(int, int);
```

- **Alternative:** Deklaration eines Typnamens `BinOp` für Zeiger auf solche Funktionen

```
typedef int (*BinOp)(int, int);  
BinOp bop;
```

- Anwendungsmöglichkeiten von Funktionszeigern

- Dynamische Auswahl von Funktionen für Aufrufe
- Übergabe von Funktionen an andere Funktionen zur Parametrisierung
- Speicherung von Funktionen in Datenstrukturen (z.B. arithmetische Operatoren in Ausdrucksbäumen)

- Wie bei Objektzeigern Verwendung als **boolescher Wert** möglich
- **Aber:** Anders als bei Objektzeigern **keine Zeigerarithmetik**
- **Adressoperator & <Ausdruck>:**  
Anwendung auf eine Funktion liefert einen Zeiger auf die Funktion
- **Funktionsaufruf <Ausdruck> ( <tatsächliche Argumente\*> ):**  
Bisher war der **<Ausdruck>** für uns lediglich ein Funktionsbezeichner, in Wahrheit wird hier ein Ausdruck erwartet, der zu einem **Zeiger auf eine Funktion** auswertet => **keine Dereferenzierung** für Aufruf erforderlich
- **Implizite unäre Konvertierungen:** Implizite Konvertierung einer Funktion in einen Zeiger auf die Funktion

```
int add(int l, int r) {  
    return l + r;  
}
```

```
int (*f)(int, int) = add; /* OK */ implizite Konvertierung  
int (*f)(int, int) = &add; /* OK */ explizite Konvertierung  
int result = f(5, 6); /* result == 11 */  
int result = add(5, 6); /* result == 11 */  
int result = (&add)(5, 6); /* result == 11 */  
int equals = (add == &add); /* equals == 1 */
```

## ▪ Allgemeine Konvertierungen (insbesondere per cast)

- Analog zu Objektzeigern Konvertierung von Funktionszeigern möglich zwischen kompatiblen Zeigertypen, außerdem von und zu integralen Typen
- **Aber:** Keine Konvertierungen zwischen Funktions- und Objektzeigern (inkl. `void*`)
- **Zuweisungskonvertierungen:** Auf Funktionszeiger dürfen nur der `NULL`-Zeiger sowie kompatible Funktionszeiger zugewiesen werden

```
int dbl(int o) {  
    return 2 * o;  
}
```

```
int (*f)(int, int) = NULL; /* OK */  
int (*f)(int, int) = add; /* OK */  
int (*f)(int, int) = dbl; /* Fehler */
```

↳ `dbl` besitzt falsche Signatur

- **Ausgangspunkt:** Ein Feld von Elementen
- **Sortieren des Feldes** erfordert den Vergleich von jeweils zwei Elementen
- **Funktionszeiger für die Vergleichsfunktion:**  
Sortieralgorithmus nur einmal implementieren, Vergleich dynamisch austauschbar

```
Field sort(Field f) {  
    ...  
    Element l, r;  
    if( l > r ) {  
        ...  
    }  
    ...  
}
```

```
typedef int (*CompareOp)(Element, Element);  
  
Field sort(Field f, CompareOp cmp) {  
    ...  
    Element r, r;  
    if( cmp(l, r) > 0 ) {  
        ...  
    }  
    ...  
}
```

- **Anwendungsbeispiel:** Feld ganzer Zahlen, aufsteigende Sortierung

```
int icmp(int l, int r) {  
    return l - r;  
}
```

```
Field fld = ...;  
fld = sort(fld, icmp);
```

## Funktionszeiger – Beispiel Map

- **Ausgangspunkt:** Ein Feld von Elementen
- Anwendung einer **unären Operation** auf jedes Element
- **Funktionszeiger für die Operation**, um das Traversieren des Feldes nur einmal implementieren zu müssen und die **Operation dynamisch auszutauschen**

```
typedef Element (*UnaryOp) (Element);

Field map(Field f, UnaryOp mop) {
    unsigned int idx = 0;
    for( idx = 0 ; idx < f.size ; idx++ ) {
        f.elems[idx] = mop(f.elems[idx]);
    }
    return f;
}
```

```
typedef int Element;
typedef struct {
    Element *elems;
    unsigned int size;
} Field;
```

- **Anwendungsbeispiel:** Feld ganzer Zahlen, Verdoppeln aller Elemente

```
int dbl(int o) {
    return 2 * o;
}
```

```
Field fld = ...;
fld = map(fld, dbl);
```

- **Ausgangspunkt:** Ein Feld von Elementen
- Anwendung eines **Kommandos** auf jedes Element, das jeweils einen **Seiteneffekt** bewirkt, aber die Elemente nicht verändert
- **Funktionszeiger für das Kommando (Rückgabetyp void)**

```
typedef void (*Command) (Element);

void foreach(Field f, Command cmd) {
    unsigned int idx = 0;
    for( idx = 0 ; idx < f.size ; idx++ ) {
        cmd(f.elems[idx]);
    }
}
```

```
typedef int Element;
typedef struct {
    Element *elems;
    unsigned int size;
} Field;
```

- **Anwendungsbeispiel:** Feld ganzer Zahlen, Ausgeben der Elemente

```
void printint(int o) {
    fprintf(stdout, "%d ", o);
}
```

```
Field fld = ....;
foreach(fld, printint);
```

- **Ausgangspunkt:** Ein Feld von Elementen
- Reduzieren des Feldes zu einem Wert per sequenzieller Anwendung einer **binären Operation** auf jedes Element und das Zwischenergebnis bis zu diesem Element
- **Funktionszeiger für die Operation**

```
typedef int          Result;
typedef Result (*CombineOp) (Result, Element);

Result reduce(Field f, CombineOp cop, Result init) {
    unsigned int idx = 0;
    for( idx = 0 ; idx < f.size ; idx++ ) {
        init = cop(init, f.elems[idx]);
    }
    return init;
}
```

```
typedef int Element;
typedef struct {
    Element *elems;
    unsigned int size;
} Field;
```

- **Anwendungsbeispiel:** Feld ganzer Zahlen, Summe aller Elemente

```
int add(int l, int r) {
    return l + r;
}
```

```
Field fld = ...;
int result = reduce(fld, &add, 0);
```

9.01.19%

- Folie 283: p = Zeigerzeiger, calloc initialisiert Werte mit Nullen.

286% Arithmetik effizienter als Dereferenzierung, außerdem weniger Speicher als bei doppelter Verzweigung.

- Maschinencodes & Hacking nicht klausurrelevant

Klausur SS18

1. Es wird "Zeichen" ausgegeben

2. Es wird das B ausgegeben

3. Summe = 0

Besonders wichtig: dynamische Speicherverwaltung von Strings