C-Crash-Kurs für Java-Programmierer

„Systemnahe Programmierung“ (BA-INF 034)

im Wintersemester 2012/2013

im Wintersemester 2012/2013

Prof. Dr. Michael Meier

Institut für Informatik 4
Universität Bonn
e-mail: mm@cs.uni-bonn.de
Sprechstunde: nach der Vorlesung bzw. nach Vereinbarung

Vergleich Java und C

- Java und C nutzen fast gleiche Syntax, aber unterschiedliche Konzepte:

Java	C

▪ Objektorientiert
  ▪ Objekte, Kapselung, Vererbung
▪ Interpretiert
  ▪ JVM, portabel, feste Typgrößen
▪ Automatische Speicherverwaltung
  ▪ *new*, Garbage Collector
▪ Strenge Typprüfung
  ▪ Keine Zeiger, eingeschränkte Casts, Indexprüfung zur Laufzeit
	▪ Prozedural
	  ▪ Funktionen, globale Variablen
	▪ Übersetzt
	  ▪ Maschinensprache, Typgrößen hardwareabhängig
	▪ Manuelle Speicherverwaltung
	  ▪ Zeiger, *malloc()/free()*
	▪ Schwache Typprüfung
	  ▪ Beliebige casts, keine Überprüfungen zur Laufzeit

Warum C?

C ist eine alte Sprache, Java (relativ) modern. C ist voller Fallstricke:

- Der Entwickler muss viel mehr „selber machen“:
  - Speicher reservieren (freigeben nicht vergessen!)
  - Sicherstellen, dass alle Programmteile genau wissen, wie verwendete Daten strukturiert sind (Position, Größe, …)
- C ist fehleranfälliger
  - Sowohl Compiler als auch Laufzeitumgebung überprüfen weniger (Variableninitialisierung, …)
  - Zeiger erlauben das Lesen und Schreiben an (fast) beliebiger Stelle im Speicher

Warum C?

Warum wird trotzdem C für systemnahes Programmieren eingesetzt?

- Unix, Linux und Windows sind alle in C geschrieben!
  - Datenstrukturen und APIs des Betriebssystems in C am „natürlichsten“ nutzbar
- Schneller als Java, C++, …
  - Nicht interpretiert wie bei Java
  - Kein OO-Overhead wie bei C++
- Totale Kontrolle
  - Keine Sandbox
  - Kein Garbage Collector (Timing!)
- „Systemnah“ bedeutet oft Bits und Bytes herum schieben
  - Netzwerkpakete, I/O-Ansteuerung, …
  - In C meist mit weniger Aufwand verbunden als in Java

HelloWorld.c

```
/* HelloWorld */

#include <stdio.h>  // für printf()

int main() {
  printf("Hello World\n");
  return 0;          // Programm fehlerfrei beendet
}
```

Übersetzen

Ein C-Programm muss vor dem Starten in eine ausführbare Datei übersetzt werden:

- gcc (Linux), MS Visual C++ (Windows), …
- Hier in der SysProg: gcc (GNU C Compiler)

Schritte beim Übersetzen (siehe auch SysInf Kap. 2):

- Präprozessor
  - Verarbeitet alle Präprozessor-Direktiven im Sourcecode (Makros, …)
- Übersetzer (Compiler)
  - Übersetzt den vorverarbeiteten Sourcecode in Maschinensprache: Objektdatei(en)
- Binder (Linker)
  - Bindet die Objektdateien und Bibliotheken zu einer ausführbaren Datei

Übersetzen mit gcc

Alle drei Schritte (Präprozessor, Übersetzen, Binden) erledigt gcc in der Voreinstellung automatisch:

```
$ gcc HelloWord.c
```

- Erzeugt aus „HelloWorld.c“ die ausführbare Datei „a.out“

```
$ gcc HelloWorld.c -o HelloWorld
```

- Nennt die ausführbare Datei „HelloWorld“ (statt „a.out“)

Hilfreiche Optionen von gcc:

- -Wall alle Warnungen einschalten
- -g Symbolnamen in Objektdatei einfügen zum Debuggen

Für mehr Informationen: „`man gcc`“!

Dateinamen und Optionen jedes Mal in der Kommandozeile anzugeben ist umständlich. Lösung: Makefile (Übungsblatt 1!)

Beispiel: backwards.c

```
#include <stdlib.h>                 /* für malloc(), free() */
#include <stdio.h>                  /* für printf() */
#include <string.h>                 /* für strlen() */
#include "backwards.h"              /* eigene Definitionen */

char *buffer, *dest;                // Zeiger auf Zielpuffer
int main(int argc, char **argv) {
   int i, j;
   buffer = malloc(BUFFER_SIZE);    // Speicher reservieren
   dest = buffer;                   // Zeiger merken
   for (i = argc-1; i > 0; i--) {   // Rückwärts über alle Argumente
     for (j = strlen(argv[i])-1; j >= 0; j--) {    // Rückwärts über
                                                   // aktuelles Argument
       *dest++ = argv[i][j];        // char in Puffer schreiben
     }
     *dest++ = ' ';                 // Wörter mit Leerzeichen trennen
   }
   *dest=0;                         // Ergebnis mit null terminieren
   printf("%s\n", buffer);          // ... und ausgeben
   free(buffer);                    // Speicher freigeben
}
```

Laufe rückwärts über alle Argumente

Laufe rückwärts über alle Zeichen des aktuellen Arguments

Kopiere Zeichen in Puffer

Trenne Wörter durch Leerzeichen

Gib Puffer aus

backwards.c

Wie sieht die Ausgabe des Programms aus?

```
$ ./backwards uni_bonn
nnob_inu
$ ./backwards systemnahe programmierung
gnureimmargorp ehanmetsys
$
```

Das Programm „backwards.c“ ist weder systemnah noch sonderlich sinnvoll…

- …aber nutzt C-typische Features (includes, Zeiger, Speichermanagement, Strings, …)
- …und enthält einen C-typischen Fehler! 

Struktur eines C-Programms

Includes

Globale Variablen

main()

weitere Funktionen (vor oder nach main)

```
#include <stdlib.h>            /* für malloc(), free() */
#include <stdio.h>             /* für printf() */
#include <string.h>            /* für strlen() */
#include "backwards.h"         /* eigene Definitionen */

char *buffer, *dest;           // Zeiger auf Zielpuffer
int main(int argc, char **argv) {
   int i, j;
   buffer = malloc(BUFFER_SIZE);        // Speicher reservieren
   dest = buffer;                       // Zeiger merken
   for (i = argc-1; i > 0; i--) {       // Rückwärts über alle
                                        // Argumente
     for (j = strlen(argv[i])-1; j >= 0; j--) {   // Rückwärts über
                                                  // aktuelles Argument
       *dest++ = argv[i][j];            // char in Puffer schreiben
     }
     *dest++ = ' ';                     // Wörter mit Leerzeichen trennen
   }
   *dest=0;                             // Ergebnis mit null terminieren
   printf("%s\n", buffer);              // ...und ausgeben
   free(buffer);                        // Speicher freigeben
}
```

Präprozessor

Anweisungen, die mit „#“ beginnen, werden vom Präprozessor vor der eigentlichen Übersetzung verarbeitet.

```
#include <stdlib.h>                    /* für malloc(), free() */
#include <stdio.h>                     /* für printf() */
#include <string.h>                    /* für strlen() */
…
```

`#include <…>` ist ähnlich zu Java `import` und bindet andere Sourcecode-Dateien in den Quelltext ein, meist Funktionsbibliotheken.

- stdio.h: Ein-/Ausgabefunktionen
- stdlib.h: Standardfunktionen und Makros
- math.h: Mathematische Funktionen
- string.h: String-Funktionen
- …mehr im Laufe der Vorlesung, z.B. für Netzwerksockets, …

Präprozessor

```
#include <stdlib.h>          /* für malloc(), free() */
#include <stdio.h>           /* für printf() */
#include <string.h>          /* für strlen() */
#include "backwards.h"       /* eigene Definitionen */

int main(int argc, char **argv) {
   …
   buffer = malloc(BUFFER_SIZE); // Speicher reservieren
   …
}
```

Hier wird Speicher reserviert – aber wie viel?

Was ist der Wert von BUFFER_SIZE?

Mittels `#include "…"` werden (eigene) Header-Dateien (aus dem selben Verzeichnis) mit Definitionen eingebunden.

Typischerweise gehört zu jeder .c-Datei eine entsprechende .h-Datei.

backwards.h

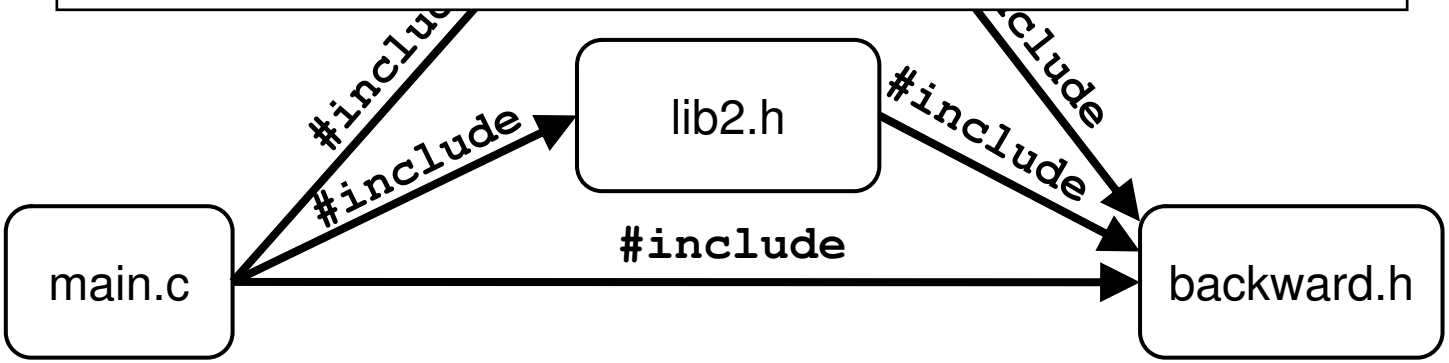
```
#ifndef _BACKWARDS_H
    #define _BACKWARDS_H
    #define BUFFER_SIZE 32
#endif
```

- `#define`: definiert Konstanten und Makros
  - Hier: Jedes Vorkommen von „BUFFER_SIZE“ wird im einbindenden Sourcecode vom Präprozessor durch „32“ ersetzt
  - Erst danach wird die so veränderte Datei übersetzt
  - Vorsicht: **Kein „;“** am Ende der Zeile! Dieses würde sonst ebenfalls eingefügt und kann beim Übersetzen zu schwer zu findenden Fehlern führen!

- Funktionssignaturen usw. können ebenfalls hier deklariert bzw. definiert werden

backwards.h

```
#ifndef _BACKWARDS_H
    #define _BACKWARDS_H
    #define BUFFER_SIZE 32
#endif
```



- Im Beispiel: backward.h mehrfach von main.c inkludiert
  - einmal direkt, zweimal indirekt
  - ⇨ Problem: mehrfache Deklarationen und Definitionen
  - ⇨ Prüfen ob backward.h bereits inkludiert

- `#ifdef/#ifndef`: entspricht „if x is defined/not defined…“
  - Hier: Sorgt dafür, dass **`BUFFER_SIZE`** nur einmal definiert wird, egal wie viele Dateien „backwards.h“ per **`#include`** einbinden – typisch für .h-Dateien

main()

Jedes C-Programm hat genau eine main-Methode:

```
int main(int argc, char **argv) {
}
```

Ähnlich wie bei Java beginnt hier der Programmfluss.

- Rückgabewert **`int`**: 0 = ok, 1=Fehler
  - Nichts zurückgeben klappt aber auch… ☺
- **`int argc`**: Anzahl Argumente
  - Mindestens 1, da Programmname immer als erstes Argument übergeben wird
- **`char **argv`** (oder auch **`char *argv[]`**): Zeiger auf Argumente
  - Genauer: Zeiger auf Zeiger auf **`char`**
  - Gleich mehr zu Zeigern und Strings…
  - argv[0] enthält immer den Name des Programms

Modulares Programmieren und Linken

Modul **fibo:**

Datei **fibo.h:**

```
#ifndef fibo_h
#define fibo_h

/* berechnet n-te Fibonaccizahl */
int fibo(int n);
#endif
```

Datei **fibo.c**:

```
#include "fibo.h"

int fibo(int n) {
  if (n < 2) return 1;
  else return fibo(n-1)+fibo(n-2);
}
```

Hauptprogramm, Datei ***main.c***

```
#include <stdio.h>
#include "fibo.h"
int main() {
  int j;
  for (j=1; j<10; j++)
    printf("%d\n", fibo(j));
  return 0;
}
```

Modul **fibo** kompilieren:

```
gcc -Wall -c fibo.c
```
⇨ `fibo.o`

Hauptprogramm kompilieren:

```
gcc -Wall -c main.c
```
⇨ `main.o`

Modul und Hauptprogramm zu ausführbarem Programm binden/linken:

```
gcc main.o fibo.o
```
⇨ `a.out`

Konstanten und Variablen

Konstanten und Variablen müssen **vor** ihrer ersten Benutzung deklariert werden:

Lokal deklariert…

…dann verwendet

```
…
int main(int argc, char **argv) {
  int i, j;
  buffer = malloc(BUFFER_SIZE);     // Speicher reservieren
  dest = buffer;                    // Zeiger merken
  for (i = argc-1; i > 0; i--) {    // Rückwärts über alle Argumente
    for (j = strlen(argv[i])-1; j >= 0; j--) { // Rückwärts
                                    // über aktuelles Argument
…
```

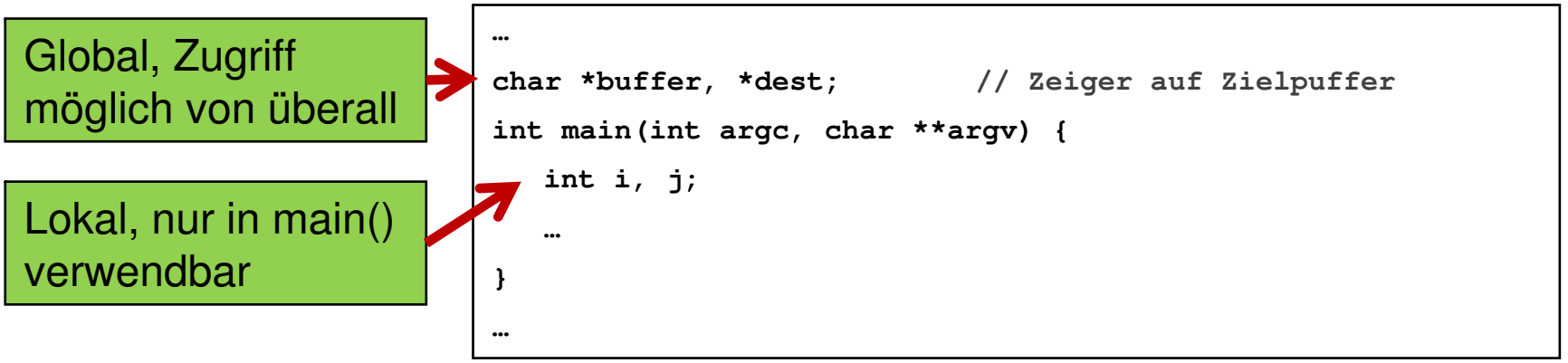
Gleichzeitig deklarieren und verwenden (wie in Java möglich) ist verboten:

Java:

```
…
for (int i = argc-1; i > 0; i--) {    // Rückwärts über alle Argumente
  for (int j = strlen(argv[i])-1; j >= 0; j--) {  // Rückwärts
                                    // über aktuelles Argument
…
```

Konstanten und Variablen

In C gibt es nicht nur lokale, sondern auch **globale Konstanten und Variablen**:



- Variablen *i* und *j* können nur in main() „gesehen“ werden
- *buffer* und *dest* können von **jeder** Funktion (innerhalb derselben Datei) gelesen und überschrieben werden!
  - Vorteil: Weniger Parameter an Funktionen zu übergeben
  - Nachteil: Fehleranfälliger (keine Kapselung, Seiteneffekte)
  - Vorsicht: Lokale Variablen können globale Variablen „überdecken“, z.B. „**`int buffer;`**“ in main()! Wird mit `-Wall` angezeigt

Datentypen

- Größe eines Datentyps ist nicht fix, sondern hardwareabhängig!
  - Java: **`int`** ist immer 4 Byte
  - C: 4 Byte für gcc/i386, 2 Byte für viele embedded devices, …
  - Tatsächliche Größe eines Typs: z.B. **`sizeof(int)`**, Werte zwischen INT_MIN und INT_MAX

- C unterstützt vorzeichenlose Typen:
  - **`unsigned int, unsigned char, …`**

- Es gibt keinen Datentyp `boolean` wie in Java!
  - Stattdessen: `int`, mit 0 = `false`, alles andere = `true`
    - `if (0) {…}`: Wird nicht ausgeführt
    - `if (42) {…}`: Wird ausgeführt

Datentypkonvertierung

- Explizite/manuelle Datentypkonvertierung (Typecasting) möglich `(Datentyp) EXPRESSION`

- Beispiel: Dividieren wir zwei Ganzzahlen (int)
  - so wird eine Ganzzahldivision durchgeführt
  - entstehender Rest wird verworfen
  - also ergibt 1/2 den Wert 0 und 7/3 hätte den Wert 2
- Durch explizites Typecasting kann hier anderes Verhalten erzwungen werden:

```
int x = 1, y = 2;
double half = (double)x/y; // nun hat half den Wert 0.5
```

Strukturen

Mit `struct` erstellt man kombinierte Datentypen:

```
struct point2d {
   double x, y;
};

struct point2d firstpoint;
firstpoint.x = 2.0;
firstpoint.y = 5.6;
```

- Jede Variable vom Typ `point2d` muss „`struct`“ in der Deklaration haben
- „;“ am Ende nicht vergessen!
- Zugriff auf Elemente der Struktur mit dem Selektor „.“

```
struct rect2d {
   struct point2d topleft,
   topright, bottomleft,
   bottomright;
};

struct rect2d r;
r.topleft.x = 4;
```

- Schachtelung möglich!
- Zugriff wieder mit dem Selektor „.“

Adressen und Zeiger

- Adressoperator `&` liefert Adresse einer Variablen
- Zeiger: Variablen, die als Werte Adressen anderer Variablen enthalten („zeigen auf andere Variablen“)
  - Analog alpha-Notation: Indirekte Adressierung (SysInf Kap. 1.3.4.5)
    - $\rho(\rho(i))$: Inhalt der Speicherzelle, deren Adresse in Speicherzelle i steht

`int c;`	Eine Variable vom Typ `int`
`int *c;`	Ein Zeiger auf eine Variable vom Typ `int`
`int *c = &d;`	Ein Zeiger auf die Adresse der Variable *d*

- Dieser Zeiger kann sich ändern! Man kann `int *c` also auf verschiedene Variablen vom Typ `int` zeigen lassen

- Dereferenzierungsoperator * liefert Variable aus/zu einem Zeiger
  - Invers zum Adressoperator: *(&c) liefert c

Zeiger und Felder

- **`int vektor[3];`**
  - Feld mit drei Integer-Komponenten **`vektor[0], vektor[1], vektor[2]`**

- **`vektor`** gibt Adresse der ersten Komponente an und kann als Zeiger auf die erste Komponente aufgefasst werden

- **`vektor`** und **`&vektor[0]`** sind gleichbedeutend

Zeigerarithmetik und Felder

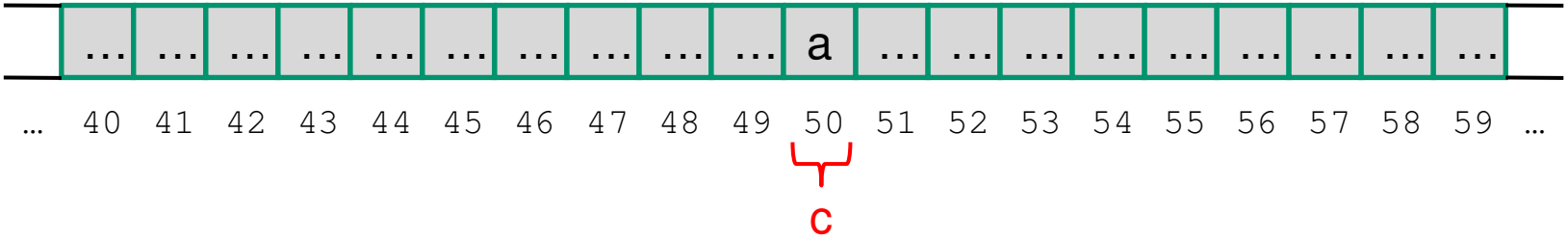
- *Größe eines Datentyps* = Anzahl der Bytes die eine Variable des Datentyps im Speicher belegt
  - `sizeof(`***datentyp***`);` `sizeof(int);`
  - `sizeof(`***variablenname***`);` `int i; sizeof(i);`

- Zeigerarithmetik
  - Additive Verknüpfung von Zeigern und ganzen Zahlen
  - Zeiger/Pointer und ganze Zahl z: `p + z`
    - ⇨ zu Adresse p wird das z-fache von d addiert, wobei d die Größe des Datentyps ist, den p referenziert

  - `vektor[i]` gleichbedeutend mit `*(vektor + i)`

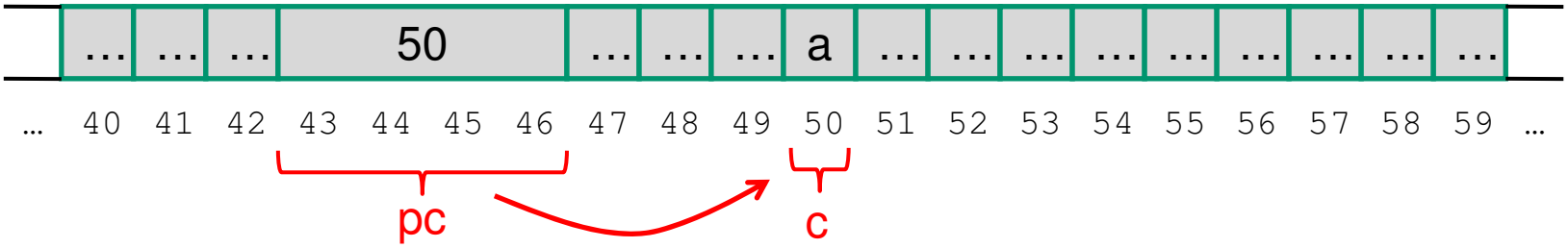
Zeiger und Speicher

```
char c = 'a';
```



Der Inhalt der Variable „c“ wird z.B. in Speicherzelle 50 gespeichert.

```
char *pc;
pc = &c;
```

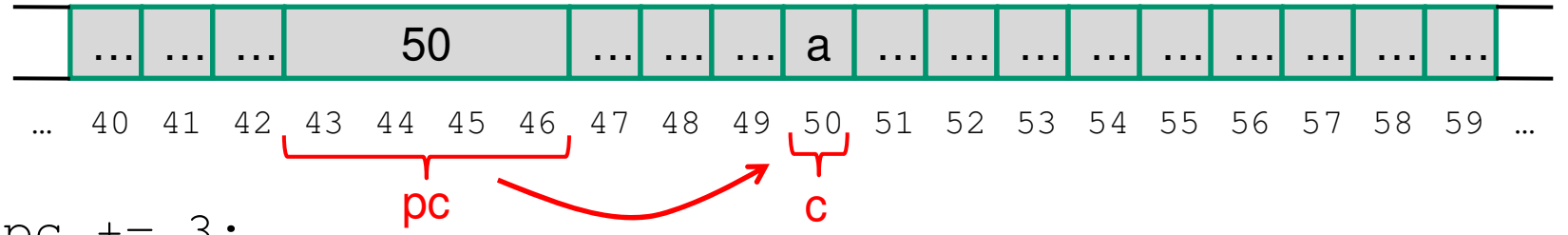


Der Inhalt der Variable „pc“ wird in Speicherzelle 43 gespeichert (hier: 4 Byte pro Adresse) und ist 50: Die Adresse der Variablen „c“

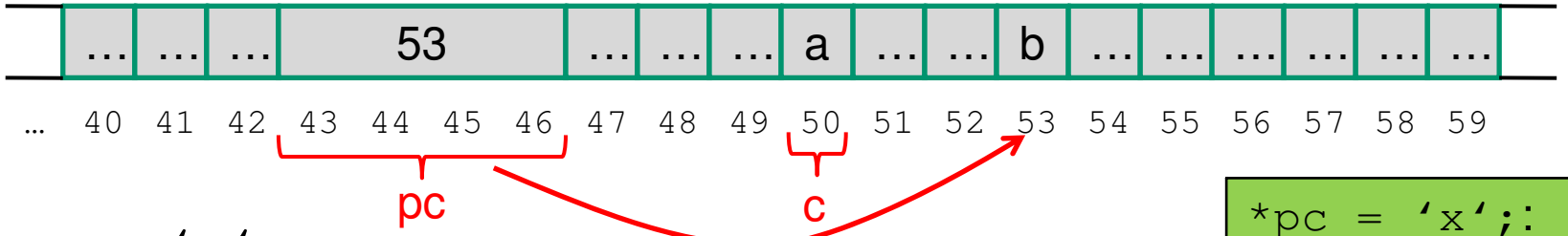
Zeiger und Speicher

Mit Zeigern kann man rechnen! Beispiel:

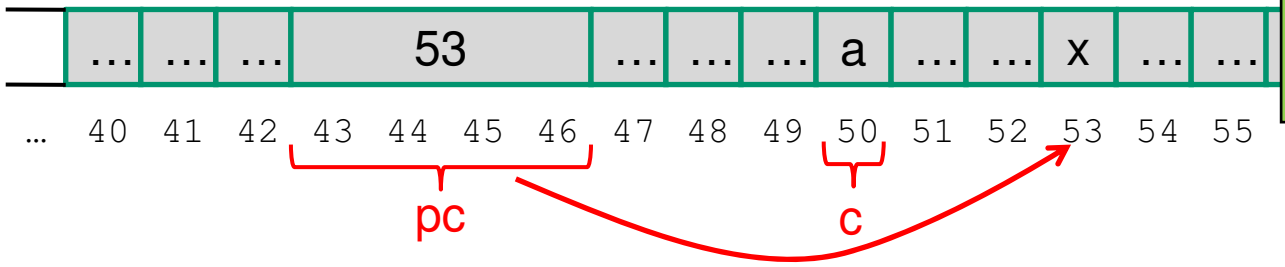
```
char *pc;
pc = &c;
```




```
pc += 3;
```




```
*pc = 'x';
```



`*pc = 'x';`:
Schreibe x in die Speicherstelle, auf die pc zeigt

Zeiger

- Zugriff auf den Speicherinhalt, auf den ein Zeiger zeigt: *
  - **`*pc = 'x';`** schreibt **`'x'`** in die Speicherzelle, auf die **`pc`** zeigt
  - **`char c = *pc;`** setzt **`c`** auf den Inhalt der Speicherzelle, auf die **`pc`** zeigt

- Auch möglich: Zeiger auf Zeiger auf…
  - **`int **ppc`** ist ein Zeiger auf einen Zeiger, der auf einen **`int`** zeigt
  - Damit möglich: (Mehrdimensionale) Arrays. Der Zeiger zeigt auf einen Speicherbereich, in dem Zeiger auf andere Variablen stehen
  - **`int **ppc`** entspricht also **`int *ppc[]`** (mit unbekannter Feldgröße)
  - **`char **argv`** ist also ein Zeiger, der auf einen (oder mehrere!) Zeiger auf **`char`** zeigt: Array von Strings!

- Zeiger auf Strukturen: Zugriff auf Elemente mit „**`->`**“ statt „**`.`**“

```
struct point2d *pp2d;
pp2d->x = 4.5;
```

  - **`pp2d->x`** äquivalent zu **`(*pp2d).x`**

Zeiger in backwards.c

```
...
char *buffer, *dest;              // Zeiger auf Zielpuffer
int main(int argc, char **argv) {
   int i, j;
   buffer = malloc(BUFFER_SIZE); // Speicher reservieren
   dest = buffer;                          // Zeiger merken
   for (i = argc-1; i > 0; i--) {          // Rückwärts über alle Argumente
     for (j = strlen(argv[i])-1; j >= 0; j--) { // Rückwärts über akt.
   Argument
       *dest++ = argv[i][j];               // char in Puffer schreiben
     }
     *dest++ = ' ';                // Wörter mit Leerzeichen trennen
   }
...
```

`*dest++ = argv[i][j]`:

- **`argv[i][j]`** interpretiert **`char **argv`** als zweidimensionales Array von **`chars`**
- **`argv[i][j]`** ist also der j-te Buchstabe des i-ten Arguments
- **`*dest++`** schreibt diesen Buchstaben in die Speicherstelle, auf die **`dest`** zeigt, und erhöht **`dest`** danach um eins

Kommandozeilenargumente

```
// arg_print.c:
#include <stdio.h>

int main( int argc, char **args ) {
  int i;
  for (i = 0; i < argc; i++)
    printf("Argument %i: %s\n",i,args[i]);
  return 0;
}
```

```
$ gcc -Wall -o arg_print arg_print.c
$ ./arg_print Dies ist ein Test
Argument 0: ./arg_print
Argument 1: Dies
Argument 2: ist
Argument 3: ein
Argument 4: Test
$ ./arg_print "Dies ist ein Test"
Argument 0: ./arg_print
Argument 1: Dies ist ein Test
```

Zeiger - Gefahren

Zeiger sind mächtig, aber gefährlich!

- Bei Zeigerarithmetik ist wichtig, auf was der Zeiger zeigt!
  - `char *pc; pc++;` erhöht pc um 1 = sizeof(char)
  - `int *pc; pc++;` erhöht pc um sizeof(int)**!**
- **Zur Laufzeit wird nicht geprüft, auf was der Zeiger zeigt!**
  - Auf 0
  - auf falsche Variablen
  - Auf Variablen vom falschen Typ
  - mitten ins eigene Programm…

Strings

- Strings in Java: Eigene Klasse mit „eingebauter“ Längenverwaltung
- In C: kein eigener String-Typ
- Behelf: **`char *`**
  - Um Speicherplatz für den String muss sich der Programmierer selber kümmern
  - Das Ende eines Strings wird mit einem Null-Byte ‚\0‘ markiert
  - **`sizeof("hallo") = 6`**
  - **`'h' 'a' 'l' 'l' 'o' '\0'`**

Strings

- Viele Bibliotheksfunktionen in <string.h>
  - Kopieren eines Strings
    - `char *strcpy(char *dest, char *src);`
    - `char *strncpy(char *dest, char *src, size_t n);`
  - Vergleich von zwei Strings
    - `int strcmp(const char *s1, const char *s2);`
    - `int strncmp(const char *s1, const char *s2, size_t n);`
  - Länge eines Strings
    - `size_t strlen(const char *s)`
    - `strlen("hallo")` = 5; `sizeof("hallo")` = 6;
  - Konvertieren von Strings in `int`/`long`
    - `int  atoi(const char *s)`
    - `long atol(const char *s);`
  - Zwei Strings aneinanderhängen **`strcat`**, **`strncat`**
  - Suchen in Strings suchen **`strchr`**, **`strstr`**
  - …
- „man 3“ und Google wissen mehr!

printf()

Wichtige Funktion, um Text auf der Konsole auszugeben: **`printf()`**

- Syntax: **`printf("Formatstring", var1, var2, …);`**
  - Gibt **`"Formatstring"`** auf der Konsole aus
  - **`var1, …`** optional
    - dient dazu, Variablenwerte auszugeben
    - Variablen ersetzen Platzhalter im **`"Formatstring"`**
    - Beispiel: **`printf(„Variable a ist %d\n", a);`**
    - **`%d: int`**
    - **`%s: char *`** (String)
    - **`%f: double`**
    - …
  - Formatierung möglich
- Varianten: **`sprintf`** (Ausgabe in String); **`fprintf`** (Ausgabe in Datei), …
- „man 3 printf“ ☺

Speicherverwaltung

- Dynamische Objekte in Java: **`new`** und Garbage Collector
- In C: Manuelles Speichermanagement durch **`malloc()`** und **`free()`**

```
…
   char *buffer, *dest;                 // Zeiger auf Zielpuffer
   buffer = malloc(BUFFER_SIZE);        // Speicher reservieren
   dest = buffer;                       // Zeiger merken
   …
   free(buffer);                        // Speicher freigeben
…
```

- **`malloc()`** allokiert Speicher
  - Mengenangabe in Byte
  - Rückgabe: Ein Zeiger, oder **`NULL`** falls Allokation fehlgeschlagen ist
- **`free()`** gibt vorher allokierten Speicher wieder frei
  - Nicht vergessen! ☺
  - In backwards.c: **`dest = buffer;`** wird benötigt, damit Zeiger auf allokierten Speicher für **`free()`** nicht verloren geht, da **`dest`** in der Schleife verändert wird!

Preisfrage

Was gibt „backwards“ bei folgender Eingabe aus?

```
$ ./backwards Zweite Vorlesung Systemnahe Programmierung
```

Preisfrage

Warum bricht „backwards“ mit einer Fehlermeldung ab?

```
…
   buffer = malloc(BUFFER_SIZE);      // Speicher reservieren
   dest = buffer;                     // Zeiger merken
   for (i = argc-1; i > 0; i--) {     // Rückwärts über alle Argumente
        for (j = strlen(argv[i])-1; j >= 0; j--) {
             *dest++ = argv[i][j];    // char in Puffer schreiben
…
```

32 Byte!

- Die Eingabe „Zweite Vorlesung Systemnahe Programmierung“ ist 43 Byte lang und damit länger als der reservierte Puffer: Speicherüberlauf!
- In diesem Fall: Glück, dass nicht andere Variablen überschrieben wurden ⇨ Seiteneffekte. Solche Fehler sind sehr schwer zu finden!
- Solch ein „Buffer Overflow“ kann Angriff auf Rechner ermöglichen!
- **Lektion: Im Umgang mit Zeigern und Speicher vorsichtig sein!**

Literatur

- Das war nur eine Auswahl von Grundlagen zu C.
  Vieles wurde nicht erwähnt:
  - **`typedef`**, **`enum`**, Pointer auf Funktionen, verschiedene Sprachstandards (C89, C99, …), Parameter in Makros, Debugger (gdb, …), Bibliotheken, …

- Wenn Fragen auftauchen:
  - „man 3“ und Google
  - „The C Book“ (online): http://publications.gbdirect.co.uk/c_book/
  - „C for Java Programmers“ (online): http://www.cs.vu.nl/~jason/college/dictaat.pdf
  - „Learning C from Java“ (online): http://www.comp.lancs.ac.uk/~ss/java2c/
  - Brian W. Kernighan und Dennis Ritchie: “The C Programming Language”
  - Mailingliste