



Vorlesung

Betriebssysteme und Systemsoftware

Kapitel 2:

C-Programmierung

Bastian Leibe

Computer Vision
Chair of Computer Science 8
RWTH Aachen University

<http://www.vision.rwth-aachen.de>

- **Betriebssysteme und Systemsoftware**

- ▶ Betriebssysteme: Aufbau und Aufgaben
- ▶ Shell- und C-Programmierung
- ▶ Prozesse und Threads, Prozessverwaltung und -kommunikation
- ▶ CPU-Scheduling
- ▶ Prozesssynchronisation, Deadlocks
- ▶ Speicherverwaltung, virtueller Speicher
- ▶ Dateisystem, Zugriffsrechte und I/O-System
- ▶ Kommunikation, verteilte Systeme

- **C-Programmierung**

- ▶ Vielen Dank an André Wichmann von der Uni Bonn (Informatik 4)
 - die Folien sind zum großen Teil orientiert an seinem „C-Crashkurs für Java-Programmierer“.

2.1 Die Programmiersprache C

- ▶ Motivation, Konzepte, Unterschiede zu Java

2.2 Aufbau eines C-Programms

- ▶ Grundelemente, Funktionen, Variablentypen, lokale/globale Variablen

2.3 Pointer

- ▶ Konzept, Arrays, Strings, Speicherverwaltung

2.4 Ausführen von C-Programmen

- ▶ Präprozessor, Compiler, Linker

- **1972 von Dennis Ritchie in den Bell Laboratories entwickelt**
 - ▶ Wurde zur Programmierung des UNIX-Betriebssystems verwendet
 - ▶ Zunächst durch den Klassiker „The C Programming Language“ von Brian Kernighan und Dennis Ritchie beschrieben und 1989 vom amerikanischen ANSI-Institut standardisiert
 - ▶ Hohe Flexibilität, kleiner Sprachumfang
 - ▶ Kaum Schutzmechanismen, kein strenges Typkonzept
 - ▶ Teilweise nicht als höhere Programmiersprache angesehen, da maschinennahe Programmierung möglich ist
 - ▶ Wird heute als Programmiersprache für Hardware-nahe Programmierung angesehen

- **Sehr ähnliche Syntax aber unterschiedliche Konzepte:**

Java	C
Objektorientiert <ul style="list-style-type: none">• Objekte, Kapselung, Vererbung	Prozedural <ul style="list-style-type: none">• Funktionen, globale Variablen
Interpretiert <ul style="list-style-type: none">• JVM, portabel, feste Typgrößen	Übersetzt <ul style="list-style-type: none">• Maschinensprache, Typgrößen hardwareabhängig
Automatische Speicherverwaltung <ul style="list-style-type: none">• <i>new</i>, Garbage Collection	Manuelle Speicherverwaltung <ul style="list-style-type: none">• Pointer, <i>malloc()/free()</i>
Strenge Typprüfung <ul style="list-style-type: none">• Eingeschränkte Casts, Indexprüfung zur Laufzeit	Schwache Typprüfung <ul style="list-style-type: none">• Beliebige Casts, keine Überprüfungen zur Laufzeit, Überschreiben möglich

- Java ist „modern“, C ist eine „alte“ Sprache voller Fallstricke:
 - ▶ Der Entwickler muss *viel mehr beachten*:
 - Speicher reservieren (und freigeben!)
 - Sicherstellen, dass alle Programmteile genau wissen, wie verwendete Daten strukturiert sind (Position, Größe, ...)
 - ▶ C ist *fehleranfälliger*
 - Compiler und Laufzeitumgebung überprüfen weniger (Variableninitialisierung, ...)
 - Zeiger erlauben Lesen/Schreiben an (fast) beliebiger Stelle im Speicher
 - ▶ Sourcecode oft *schwerer zu lesen*
 - Kryptische Variablennamen (oft global)
 - „Kreative“ Zeigerarithmetik
 - Manchmal 1337 hax0r Coding Style

A problem has been detected and windows has been shut down to prevent damage to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)

*** SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c


```
#define p struct c
#define q struct b
#define h a->a
#define i a->b
#define e i->c
#define o a=(*b->a) (b->b,b->c)
#define s return a;}q*
#define n (d,b)p*b;{q*a;p*c;
#define z(t) (t*)malloc(sizeof(t))
q{int a;p{q*(*a)();int b;p*c;}*b;};q*u n a=z(q);h=d;i=z(p);i->a=u;
    i->b=d+1;s
v n c=b;do o,b=i;while(!(h%d));i=c;i->a=v;i->b=d;e=b;s
w n o;c=i;i=b;i->a=w;e=z(p);e->a=v;e->b=h;e->c=c;s
t n for(;;)o,main(-h),b=i;}main(b){p*a;if(b>0)a=z(p),h=w,a->c=z(p),
    a->c->a=u,
a->c->b=2,t(0,a);putchar(b?main(b/2),-b%2+'0':10);}
```

The International Obfuscated
C Code Contest:

<http://www.ioccc.org/>

- **Unix, Linux und Windows sind alle in C geschrieben**
 - ▶ *Schneller* als Java
 - Nicht interpretiert wie bei Java
 - Kein Overhead durch Objektorientierung
 - ▶ *Totale Kontrolle*
 - Keine Sandbox
 - Kein Garbage Collector, umfassendere Kontrolle von Low-Level-Operationen durch eigene Speicherverwaltung
 - ▶ „*Systemnah*“ bedeutet oft Bits und Bytes herumschieben
 - Netzwerkpakete, I/O-Ansteuerung, ...
 - In C meist mit weniger Aufwand verbunden als in Java

2.1 Die Programmiersprache C

- ▶ Motivation, Konzepte, Unterschiede zu Java

2.2 Aufbau eines C-Programms

- ▶ Grundelemente, Funktionen, Variablentypen, lokale/globale Variablen

2.3 Pointer

- ▶ Konzept, Arrays, Strings, Speicherverwaltung

2.4 Ausführen von C-Programmen

- ▶ Präprozessor, Compiler, Linker

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

char *buffer, *dest;

int main(int argc, char **argv) {
    int i, j;
    buffer = malloc(BUFFER_SIZE);
    dest = buffer;
    for (i = argc-1; i > 0; i--) {
        for (j = strlen(argv[i])-1; j >= 0; j--) {
            *dest++ = argv[i][j];
        }
        *dest++ = ' ';
    }
    *dest=0;
    printf("%s\n", buffer);
    free(buffer);
    return EXIT_SUCCESS;
}
```

```
/* für malloc(), free() */
/* für printf() */
/* für strlen() */
/* eigene Definitionen */

// Pointer auf Zielpuffer

// Pointer reservieren
// Zeiger merken
// Rückwärts über alle Argumente
// Rückwärts über akt. Arg.
// char in Puffer schreiben

// Wörter mit Leerzeichen trennen

// Ergebnis mit null terminieren
// ...und ausgeben
// Speicher freigeben
// Programm beenden (optional)
```

Includes

Globale Variablen

main()

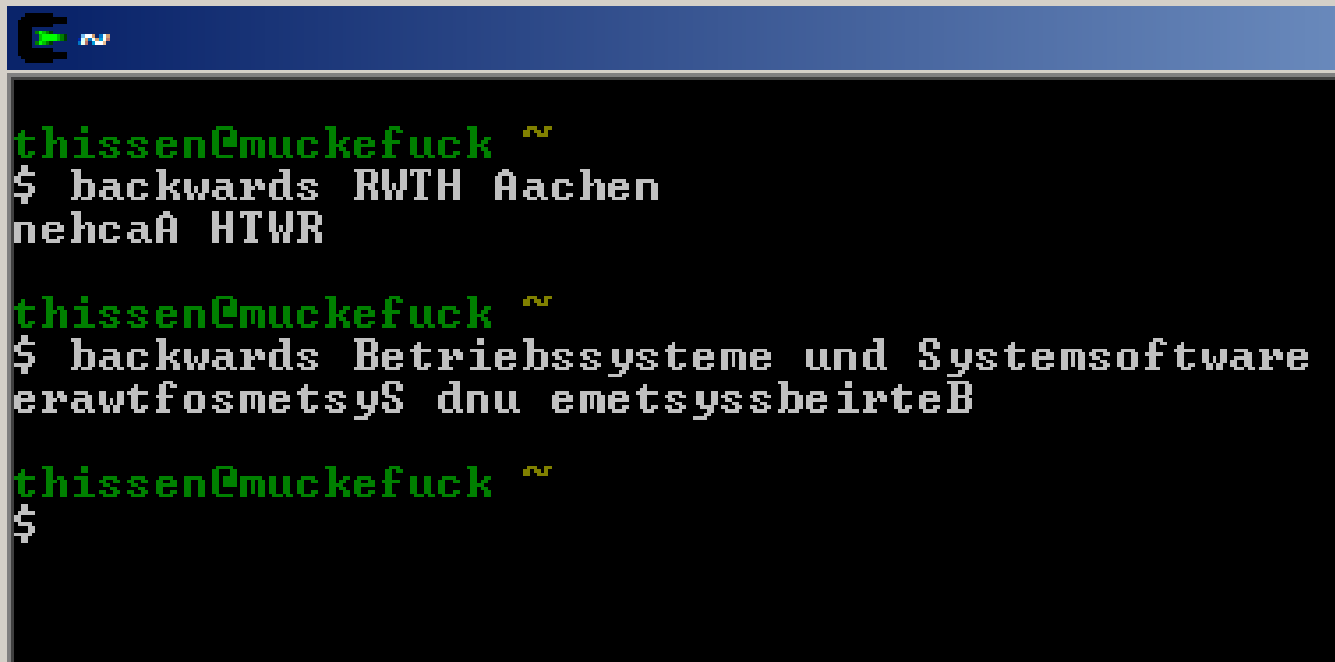
Ggfs. weitere Funktionen (vor oder nach main)

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

char *buffer, *dest;         // Pointer auf Zielpuffer

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

- Verwendung des Programms:



```
thissen@muckefuck ~  
$ backwards RWTB Aachen  
nehcaA HTWR  
  
thissen@muckefuck ~  
$ backwards Betriebssysteme und Systemsoftware  
erawtfosmetsyS dnu emetsyssbeirteB  
  
thissen@muckefuck ~  
$
```

- Das Programm `backwards.c` ...
 - ▶ ... nutzt C-typische Features (includes, Pointer, Speichermanagement, Strings, ...)
 - ▶ ...und enthält einen C-typischen Fehler!

- Jedes Programm hat genau eine **main()** Funktion:

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

- Ähnlich wie bei Java beginnt hier der Programmfluss
 - ▶ Rückgabewert **int**: 0 = ok, alles andere=Fehler
 - ▶ **int argc**: Anzahl Argumente
 - Mindestens 1, da Programmname als erstes Argument übergeben wird
 - ▶ **char **argv** (oder auch **char *argv[]**): Pointer auf Argumente
 - Genauer: Pointer auf Pointer auf **char**
 - **argv[0]** enthält immer den Namen des Programms
 - ▶ Andere Möglichkeit: **int main(void)**

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

```
char *buffer, *dest; // Zeiger auf Zielpuffer
```

```
...
```

```
int main(int argc, char **argv) {  
    int i, j;  
    ...  
}
```

Global, Zugriff von
überall möglich

Lokal, nur in main()
bekannt

- Variablen **i** und **j** sind nur in `main()` sichtbar
- **buffer** und **dest** können von *jeder* Funktion gelesen und überschrieben werden
 - ▶ Vorteil: Weniger Parameter an Funktionen zu übergeben
 - ▶ Nachteil: Fehleranfälliger (keine Kapselung, Seiteneffekte)
 - ▶ Vorsicht: Lokale Variablen können globale Variablen überdecken!

- **Übergabe aller benötigten Parameter**

- ▶ Achtung: pass-by-value, d.h. alter Wert bleibt erhalten!
- ▶ Lösung wird später in diesem Vorlesungsblock erklärt

```
int summiere (int oldsum, int add) {  
    return oldsum + add;  
}
```

- **Statische Funktionsvariablen**

- ▶ Bleiben zwischen Aufrufen erhalten

```
int summiere (int add) {  
    static int sum = 0;  
    sum += add;  
    return sum;  
}
```

- Größen der eingebauten Variablentypen sind **nicht** fix, sondern **hardwareabhängig**!
 - ▶ Java: **int** ist immer 4 Byte
 - ▶ C: 4 Byte für gcc/i386, 2 Byte für viele embedded devices, ...
 - ▶ Typgröße kann durch den Operator **sizeof** ermittelt werden:
sizeof(int)
- Aber es gibt Alternativen in der Standardbibliothek
 - ▶ Nach **#include <stdint.h>** sind u.A. folgende Typen verfügbar:
 - **int8_t, int16_t, int32_t, int64_t**
 - **uint8_t, uint16_t, uint32_t, uint64_t**
 - ▶ Nach **#include <stddef.h>** sind u.A. folgende Typen verfügbar:
 - **size_t, ptrdiff_t**

- **Vorsicht mit Wahrheitswerten:**

- ▶ Java: **boolean**, kann die Werte **true** und **false** annehmen
- ▶ C: **int** und **_Bool** – beide Typen sind *integer*
- ▶ Einfache Regel: **0 ist falsch, alles andere ist wahr**

- **Fallstricke in der Standardbibliothek:**

- ▶ Nach `#include <stdbool.h>` scheint alles ok:
- ▶ **bool**, **true**, **false** sind alle definiert, jay!
- ▶ Wenn eine Funktion aber **int** zurückgibt, und damit einen Wahrheitswert ausdrücken will...

```
bool kaputt(char c) {  
    return isalnum(c) == true;  
}
```

```
bool funktioniert(char c) {  
    return !!isalnum(c);  
}
```

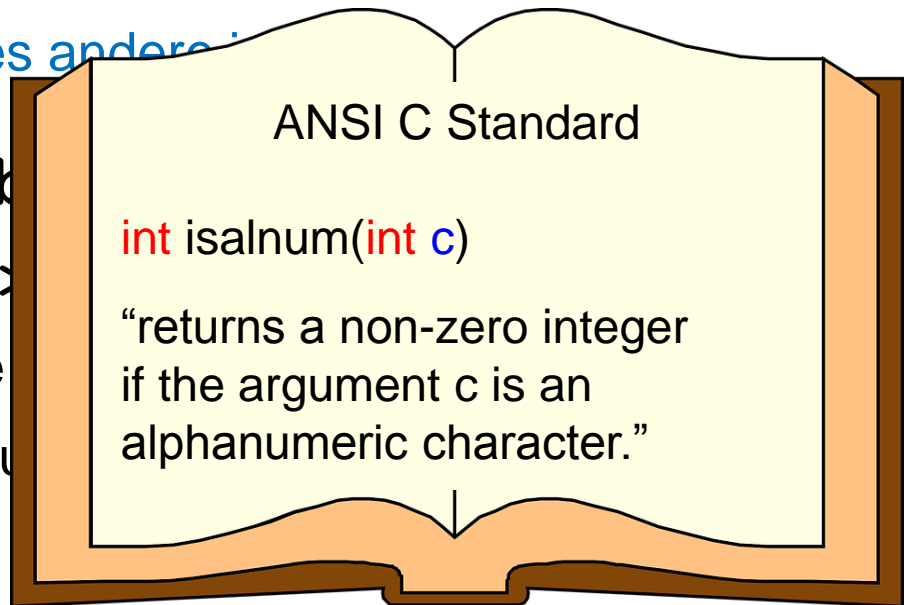
- **Vorsicht mit Wahrheitswerten:**

- ▶ Java: **boolean**, kann die Werte **true** und **false** annehmen
- ▶ C: **int** und **_Bool** – beide Typen sind *integer*
- ▶ Einfache Regel: **0 ist falsch, alles andere ist wahr**

- **Fallstricke in der Standardbibliothek**

- ▶ Nach `#include <stdbool.h>`
- ▶ **bool**, **true**, **false** sind alle
- ▶ Wenn eine Funktion aber **int** zu...
Wahrheitswert ausdrücken will..

```
bool kaputt(char c) {  
    return isalnum(c) == true;  
}
```



```
bool funktioniert(char c) {  
    return !!isalnum(c);  
}
```

- Mit **struct** erstellt man kombinierte Datentypen

- ▶ Jede Variable vom Typ `point2d` muss „**struct**“ in der Deklaration haben
- ▶ Zugriff auf Elemente der Struktur mit dem Selektor „.“

```
struct point2d {  
    double x, y;  
};  
  
struct point2d firstpoint;  
firstpoint.x = 2.0;  
firstpoint.y = 5.6;
```

- Ein **typedef** gibt einem alten Typen einen neuen Namen

- ▶ Der „Typedef-Trick“ kombiniert eine **struct** Deklaration mit einem **typedef**

```
typedef struct _point2d {  
    double x, y;  
} point2d;  
  
point2d firstpoint;  
firstpoint.x = 2.0;  
firstpoint.y = 5.6;
```

- Eine **struct** kann größer sein als die Summe ihrer Teile

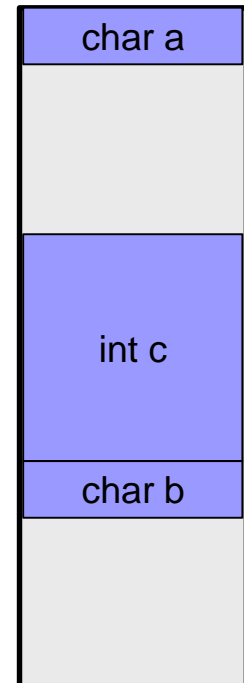
- ▶ `sizeof(int)` = 4
- ▶ `sizeof(char)` = 1
- ▶ `sizeof(struct cci)` = 8
- ▶ `sizeof(struct cic)` = 12

```
struct cci{  
    char a;  
    char b;  
    int c;  
};
```

```
struct cic{  
    char a;  
    int c;  
    char b;  
};
```

- Compiler erzeugt Datenstruktur für effizienten Zugriff

- ▶ CPU-Operationen arbeiten auf Blöcken
 - z. B. 2 Byte, 4 Byte, etc.
- ▶ Variable kleiner als Blockgröße:
 - Padding für Alignment
- ▶ Zugriff auf **int c**: eine (schnelle) Operation



- **Nicht schlauer sein wollen als der Compiler!**
 - ▶ Niemals “von Hand” Größe abschätzen
 - z.B. beim manuellen Allokieren von Speicher
 - Abhängig von Architektur, auf der kompiliert wird
 - ▶ Immer “**sizeof**” nutzen, wenn man die Größe braucht
- **Wenn man Padding abschalten will...**
 - ▶ Gibt es sogenannte “packed structs”
 - Größe der struct = Größe der Summe der Elemente
 - Wird z.B. bei Netzwirkkommunikation genutzt (garantiertes Layout)
 - ▶ Nachteil: wesentlich langsamer!
 - Nur verwenden, wenn man sie wirklich braucht
 - Ordnungsliebe und “Speichereffizienz” sind keine Gründe!

- **Enum: selbstdefinierter Datentyp**

- ▶ Aufzählung von Namen, die der Datentyp annehmen kann
- ▶ Unter der Haube ein Integer
- ▶ Kann Verständlichkeit unterstützen

```
enum farbe {ROT, GELB};

int main (void) {
    enum farbe color;
    color = ROT;
    color = 1; // GELB
}
```

- **Definition als “vollwertiger” Datentyp**

- ▶ **typedef**
- ▶ Vergabe eines Names
- ▶ Kann später wie `int`, `float`, etc. genutzt werden
- ▶ Nicht nur für enums: structs, etc.

```
typedef enum farbe {ROT, GELB}
farbe_t;

int main (void) {
    farbe_t color;
    color = ROT;
    color = 1; // GELB
}
```


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

- ▶ Syntax: `printf("Formatstring", var1, var2, ...);`
- ▶ Gibt formatierten String auf der Konsole aus
- ▶ `var1, var2, ...` optional
 - Dient dazu, Variablenwerte auszugeben
 - Variablen ersetzen Platzhalter im „formatierten String“
 - Beispiel: `printf("Variable a ist %d\n", a);`
 - `%d`: `int`
 - `%u`: `unsigned int`
 - `%s`: `char *` (String)
 - `%f`: `double`
 - ...
- ▶ Varianten: `fprintf`, `snprintf`, ...

2.1 Die Programmiersprache C

- ▶ Motivation, Konzepte, Unterschiede zu Java

2.2 Aufbau eines C-Programms

- ▶ Grundelemente, Funktionen, Variablentypen, lokale/globale Variablen

2.3 Pointer

- ▶ Konzept, Arrays, Strings, Speicherverwaltung

2.4 Ausführen von C-Programmen

- ▶ Präprozessor, Compiler, Linker

- ***Pointer (Zeiger)***: Variablen, die auf andere Variablen (eines bestimmten Typs) zeigen
 - ▶ „Auf Variable zeigen“ bedeutet: der Zeiger ist nur *die Adresse der Speicherstelle*, an der die eigentliche Variable steht
- **C nutzt Zeiger für die Umsetzung vieler Konzepte**
 - ▶ Arrays
 - Strings als spezielle arrays
 - ▶ Pass by value vs. pass by reference
 - ▶ “manuelle Objektorientierung”

- **Beispiele:**

- ▶ `int c;`

c ist eine Variable vom Typ `int`

- ▶ `int *d;`

d ist ein Zeiger auf eine Variable vom Typ `int`

- ▶ `d = &c;`

d zeigt auf c, d.h. d enthält die Adresse von c
(`&` = Referenzierungsoperator)

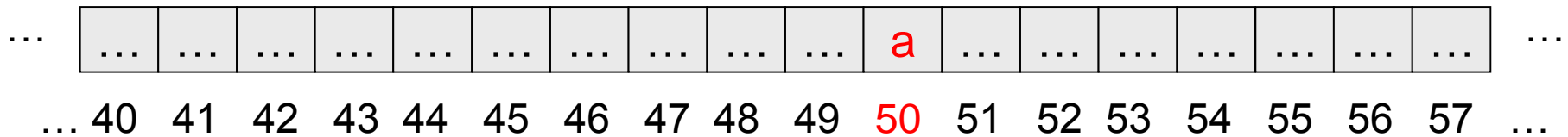
- ▶ `c = *d;`

c wird der Wert zugewiesen, auf den d zeigt
(`*` = Dereferenzierungsoperator)

- ▶ `void *v;`

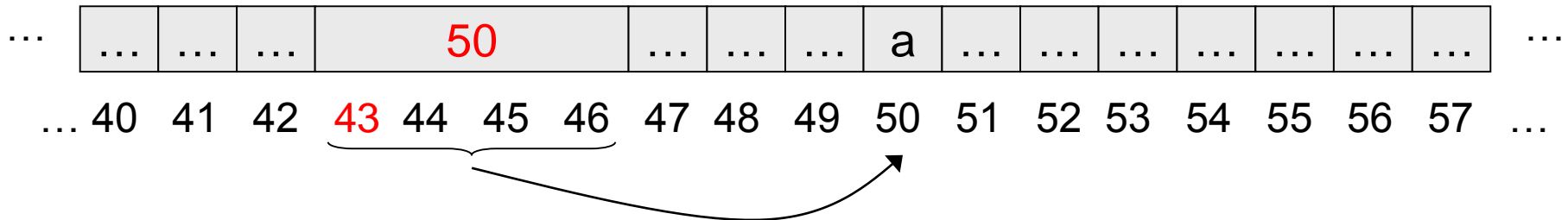
Generischer Pointer, kann jedem anderen Zeigertyp zugewiesen werden: `char *s = v;`

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



Der Inhalt der Variablen `c` wird z.B. in Speicherzelle 50 gespeichert.

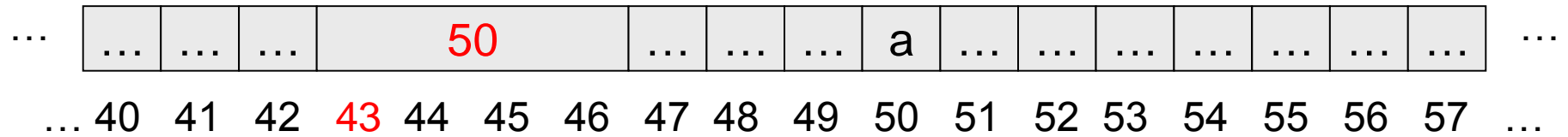
```
char *pc = &c;
```



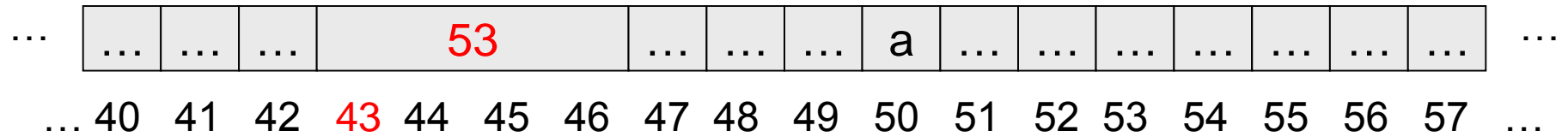
Der Inhalt der Variablen `pc` wird z.B. in Speicherzelle 43 gespeichert (hier: 4 Byte pro Adresse) und ist 50

- Mit Pointern kann man rechnen:

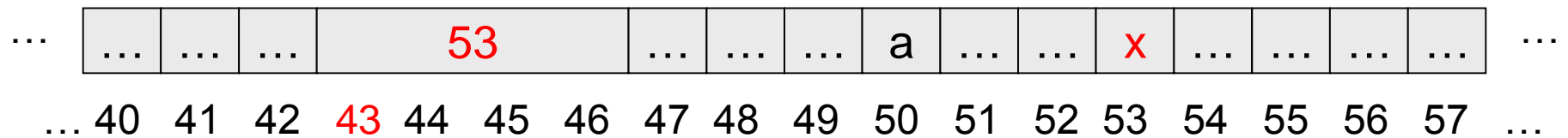
```
char *pc = &c;
```



```
pc += 3;
```

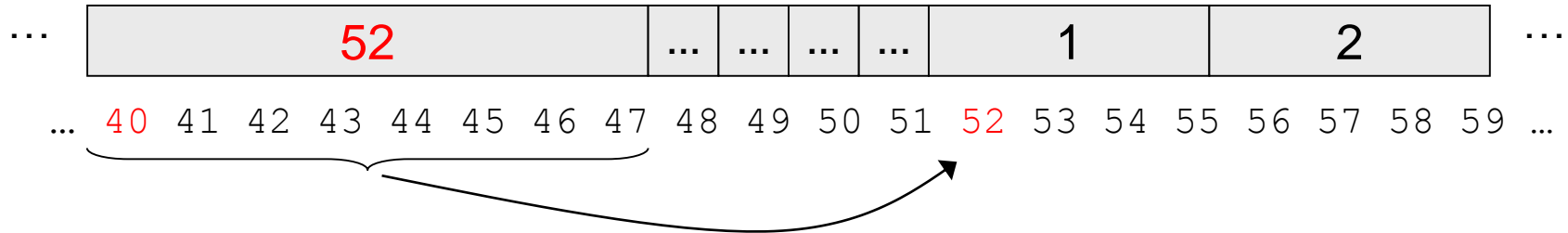


```
*pc = 'x';
```

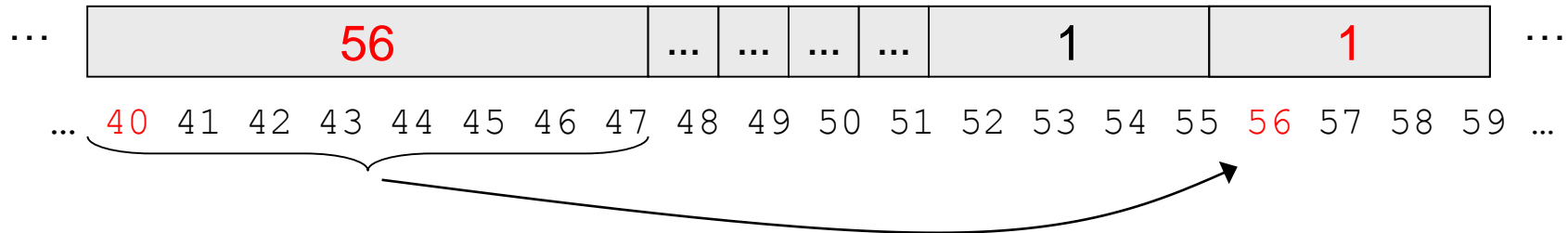


- Pointerarithmetik ist abhängig vom Zieltyp:

```
int32_t arr[] = {1, 2};  
int32_t *parr = &arr[0];
```



```
parr += 1;  
*parr = *(parr - 1);
```



- **Pointer sind mächtig, aber gefährlich!**
- **Zur Laufzeit wird nicht geprüft, auf was der Pointer zeigt!**
 - ▶ Auf `NULL` bzw. 0
 - ▶ Auf falsche Variablen
 - ▶ Auf Variablen vom falschen Typ
 - ▶ Mitten ins eigene Programm
 - ▶ Auf Speicherbereiche anderer Programme
- **Dies führt typischerweise frühestens zu Problemen, wenn sie dereferenziert werden...**

- 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 Zelle, auf die `pc` zeigt
- Auch möglich: *Zeiger auf Zeiger*
 - ▶ `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 Größe)
 - `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`

- Arrays in C liegen zusammenhängend im Speicher

- ▶ `int32_t array[16]` = 16*4 byte Speicher

- Arrays benehmen sich wie ein Pointer auf das erste Element:

- `array[0]` entspricht `*(array+0)` bzw. `*array`

- `array[3]` entspricht `*(array+3)`

- `int32_t *ap = array;`

- `*ap` entspricht `ap[0]`

- `*(ap+3)` entspricht `ap[3]`

- ▶ Arrays werden typischerweise als Pointer an Funktionen übergeben

- ▶ Wenn Arrays in Funktionsprototypen stehen **sind** es Pointer!

- Kein Unterschied:

```
int main(int argc, char **argv);
```

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

```
...
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 Arg.
        for (j = strlen(argv[i])-1; j >= 0; j--) { // über akt. Arg.
            *dest++ = argv[i][j];           // char in Puffer schreiben
        }
        *dest++ = ' ';                      // Wörter mit Leerz. trennen
    }
    ...
}
```

- Was bewirkt `*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
 - Rechts vor links (Suffix vor Prefix)

- **Strings in Java: Klasse mit automatischer Längenverwaltung**
- **In C: *kein eigener String-Typ*, sondern `char *`**
 - ▶ Speicherplatz für den String muss der Programmierer verwalten
 - ▶ Das Ende eines Strings wird mit einem Null-Byte markiert
- **Viele Bibliotheksfunktionen in `<string.h>`**
 - ▶ `strcpy, strncpy` Kopiert einen String
 - ▶ `strcmp, strncmp` Vergleicht zwei Strings
 - ▶ `strlen` Ermittelt Länge eines Strings
 - ▶ `atoi, atol` String in int/long konvertieren
 - ▶ `strcat, strncat` Zwei Strings aneinanderhängen
 - ▶ `strchr, strstr` In Strings suchen
 - ▶ ...

- *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!
- `malloc()` **und** `free()` **nur für Pointer!**
 - ▶ „Normale“ Variablen bekommen ihren Speicher automatisch

- **Aufruf**

`./backwards Betriebssysteme und Systemsoftware ist
eine tolle Vorlesung`

- **Warum stürzt das Programm ab?**

- ▶ Falls es abstürzt...

- Wenn man „Glück“ hat, läuft das Programm auch erst mal weiter
 - Bis es dann eventuell später knallt
 - Hängt davon ab, wie malloc() auf der jeweiligen Plattform umgesetzt ist und wie großzügig es Speicher allokiert
 - Konsequenz aus Murphy's Law:
Das Programm stürzt immer nur beim Kunden ab...

```
...  
buffer = malloc(BUFFER_SIZE); // Speicher reservieren  
...  
32 Byte!
```

- Die Eingabe „Betriebssysteme und Systemsoftware ist eine tolle Vorlesung“ ist (59+1) Byte lang und damit länger als der reservierte Puffer: *Speicherüberlauf* !
- In diesem Fall: *Heap Corruption* durch Überschreiben anderer Variablen → *Seiteneffekte*. Solche Fehler sind sehr schwer zu finden!
- Solch ein „*Buffer Overflow*“ kann Angriffe auf Rechner ermöglichen!

- **Konzepte bei Übergabe von Funktionsparametern**
 - ▶ C ist immer pass-by-value
 - ▶ Aber pass-by-reference über Pointer
 - ▶ Pointer selbst kann nicht geändert werden (pass-by-value), wohl aber der Wert, auf den er zeigt.

```
void add20 (int a) {  
    a += 20;  
}  
  
int main(void) {  
    int a = 5;  
    add20(a);  
    // a = 5;  
}
```

```
void add20 (int *a) {  
    *a += 20;  
}  
  
int main(void) {  
    int a = 5;  
    add20(&a);  
    // a = 25;  
}
```


- Auch möglich: *Function Pointer*

- ▶ `returnType (*ptrName) (arg1, arg2, ...);`

- ▶ Beispiel `int (*fp) (double x)`: Zeiger auf eine Funktion, die ein Integer zurückgibt

- Syntax:

- ▶ `int func(void);`

Funktion ohne Argumente mit einem Integer als Rückgabewert

- ▶ `int *func(void);`

Funktion ohne Argumente mit einem Pointer auf ein Integer als Rückgabewert

- ▶ `int (*func)(void);`

Pointer auf eine Funktion ohne Argumente mit einem Integer als Rückgabewert

- ▶ `int *(*func)(void);`

Pointer auf eine Funktion ohne Argumente mit einem Pointer auf einen Integer als Rückgabewert

```
#include <stdio.h>

void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
    myproc(10);           /* call myproc with parameter 10*/
    mycaller(myproc, 10); /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
    (*f)(param);          /* call function *f with param */
}

void myproc (int d){
    printf("%d\n", d);    /* Verarbeite d */
}
```

Wozu Function Pointer?

- **Abstraktion durch Indirektion**

- ▶ Austausch von Funktionalität ohne Änderung des restlichen Programms
- ▶ Auch zur Laufzeit möglich!
- ▶ Objektorientierung

- **Beispiele aus dem Linux-Kernel**

- ▶ Austausch von Algorithmen
 - Ratenadaption in WiFi
 - Staukontrolle in TCP (-> DatKom)
- ▶ Abstraktion von Implementierungen
 - “virtuelles Dateisystem” VFS
 - Implementiert allgemeine Konzepte
 - Dateisystemspezifische Umsetzung durch Zeiger auf Funktionen
 - Ähnlich zu Vererbung

```
void funca (void) {  
    printf("Ich bin a!\n");  
}  
  
void funcb (void) {  
    printf("Ich bin b!\n");  
}  
  
int main(void) {  
    void* (*fp) (void);  
    fp = &funca;  
    fp();  
    fp = &funcb;  
    fp();  
}
```

2.1 Die Programmiersprache C

- ▶ Motivation, Konzepte, Unterschiede zu Java

2.2 Aufbau eines C-Programms

- ▶ Grundelemente, Funktionen, Variablentypen, lokale/globale Variablen

2.3 Pointer

- ▶ Konzept, Arrays, Strings, Speicherverwaltung

2.4 Ausführen von C-Programmen

- ▶ Präprozessor, Compiler, Linker

- Ein C-Programm muss vor dem Starten in eine ausführbare Datei übersetzt (compiliert) werden:
 - ▶ gcc (Linux), MS Visual C++ (Windows), ...
- Schritte beim Compilieren:
 - ▶ *Präprozessor*
 - Verarbeitet alle Präprozessor-Direktiven im Sourcecode (Makros, ...)
 - ▶ *Übersetzer*
 - Übersetzt den vorverarbeiteten Sourcecode in Maschinsprache: Objektdatei(en)
 - ▶ *Binder (Linker)*
 - Bindet die Objektdateien und Bibliotheken zu einer ausführbaren Datei

- Alle drei Schritte (Präprozessor, übersetzen, binden) erledigt gcc in der Voreinstellung automatisch:
 - ▶ `$ gcc backwards.c`
 - Erzeugt aus `backwards.c` die ausführbare Datei `a.out`
 - ▶ `$ gcc -o backwards backwards.c`
 - Nennt die ausführbare Datei `backwards` (statt `a.out`)
- Fehler und Warnungen:
 - ▶ Ein kleiner Fehler kann viele Meldungen verursachen – den ersten Fehler beseitigen kann helfen!
 - ▶ `-Wall -Wextra -pedantic`:
mehr Warnungen einschalten. Warnungen nicht ignorieren!
 - ▶ Für mehr Informationen: `man gcc`

- Dateinamen und Optionen jedes Mal in der Kommandozeile anzugeben ist umständlich. Lösung: *Makefile*
- Bei uns:
 - ▶ `$ gcc -o backwards backwards.c`
- Mit Makefile:
 - ▶ `backwards: backwards.c backwards.h`
 - ▶ `gcc -o backwards backwards.c`
- Aufruf mit `make backwards`
 - <Tab>

- 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 "backwards.h"   /* eigene Definitionen */
```

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

- ▶ `stdio.h`: Ein-/Ausgabefunktionen
- ▶ `stdlib.h`: Standardfunktionen und Makros
- ▶ `string.h`: String-Funktionen
- ▶ ...


```
#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
    ...
}
```

Wie viel Speicher
wird reserviert?

- Mittels `#include` “...” werden (eigene) Header-Dateien (aus dem selben Verzeichnis) eingebunden
- Typischerweise gehört zu jeder .c-Datei eine entsprechende .h-Datei

```
#pragma once  
  
#define BUFFER_SIZE 32
```

```
#ifndef BACKWARDS_H  
#define BACKWARDS_H  
#define BUFFER_SIZE 32  
#endif
```

- **#define** definiert Konstanten und Makros
 - ▶ Jedes Vorkommen von „BUFFER_SIZE“ wird im Sourcecode vom Präprozessor durch „32“ ersetzt
 - ▶ Erst danach wird die so veränderte Datei übersetzt
- **#pragma once** bzw. **#ifndef/#define/#endif**
 - ▶ Sorgt dafür, dass **BUFFER_SIZE** nur einmal definiert wird, egal wie viele Dateien `backwards.h` per **#include** einbinden
- Funktionssignaturen usw. können ebenfalls hier deklariert werden

- **Sparsame Verwendung**

- ▶ Subtile Fehler
- ▶ Kaum sichtbar beim Debugging
- ▶ Code schwer zu lesen
- ▶ `#define square(a) a*a` – was ist das Ergebnis des Aufrufs `b = square(4+5) ?`
- ▶ Merke: Der Präprozessor ist ein simpler String-Matcher und –Ersetzer
- ▶ Mächtig, aber „dumm“

- **Es gibt für vieles Alternativen:**

- ▶ `#define INT16` → Typdefinition mit `typedef`
- ▶ `#define MAXLEN` → Konstante mit `const`
- ▶ `#define max(a,b)` → (inline) Funktionsdefinition

- Übersetzt C-Quelltext in Objektcode

- ▶ Nah am Maschinencode, aber noch nicht ausführbar

- C kompiliert single pass

- ▶ Code wird genau einmal durchlaufen
- ▶ Reihenfolge von Deklarationen wichtig!
- ▶ Eine Lösung: *forward declarations*
- ▶ Vor allem bei selbstdefinierten Datentypen und zirkulären Abhängigkeiten

```
int add (int a, int b);  
  
int main (void) {  
    return add(2,3);  
}  
  
int add (int a, int b) {  
    return a+b;  
}
```

```
forward_declaration.c: In function 'main':
```

```
forward_declaration.c:4: warning: implicit  
declaration of function 'add'
```

- **Nimmt eine odere mehrere Objektdateien**
 - ▶ Plus evtl. Libraries...
 - ▶ Wie z.B. die Standard-Libraries, gegen die immer gelinkt wird
 - ▶ Und deren Funktionen wir mit `#include <stdXXX.h>` nutzen
 - ▶ Verknüpft (linked) Funktionsaufrufe über Objektdateigrenzen hinweg
- **Erstellt ein Binary aus Maschinencode**
- **Wenn Verknüpfung fehlschlägt...**

```
/usr/lib/gcc/x86_64-linux-  
gnu/4.4.3/../../../../lib/crt1.o: In function  
`_start':  
  
(.text+0x20): undefined reference to `main'  
collect2: ld returned 1 exit status
```

- *Im Umgang mit Zeigern und Speicher vorsichtig sein!*
- `malloc()` **vermeiden!**
 - ▶ Oft nicht nötig
 - ▶ Vermeidung der Nutzung reduziert Fehlerpotential!
 - ▶ Keine „Magic Numbers“ verwenden!
- **Den (hoffentlich ;-)) sauberen Programmierstil aus Java nicht abgewöhnen!**

- **Der Klassiker**

- ▶ B. Kernighan, D. Ritchie: *Programmieren in C*. Hanser-Verlag, 1990.

- **Online-Ressourcen zu C**

- ▶ <https://en.cppreference.com/w/c>

- **C für Java-Programmierer**

- ▶ S. Simpson: *Learning C from Java*.
<http://www.lancs.ac.uk/~simpsons/java2c/>

- **Betriebssysteme und Systemsoftware**

- ▶ Betriebssysteme: Aufbau und Aufgaben
- ▶ **Shell-** und C-Programmierung
- ▶ Prozesse und Threads, Prozessverwaltung und -kommunikation
- ▶ CPU-Scheduling
- ▶ Prozesssynchronisation, Deadlocks
- ▶ Speicherverwaltung, virtueller Speicher
- ▶ Dateisystem, Zugriffsrechte und I/O-System
- ▶ Kommunikation, verteilte Systeme
- ▶ Virtualisierung