



**Hochschule
Augsburg** University of
Applied Sciences

Vorlesung: Betriebssysteme

Einführung

Prof. Dr. Lothar Braun, Dr.-Ing. Volodymyr Brovkov
Sommersemester 2024

Organisation

Einführung in Betriebssysteme

Geschichte der Betriebssysteme

Einführung Praktikum und Programmiersprache C

Organisation

Einführung in Betriebssysteme

Geschichte der Betriebssysteme

Einführung Praktikum und Programmiersprache C

- **Vorlesung**

- Vorlesung findet solange möglich in Präsenz
- Folien der Vorlesung über Moodle
- Moodle-Kurs: <https://moodle.hs-augsburg.de/course/view.php?id=8019>

- **Praktikum**

- Bereiten Sie die Aufgaben Zuhause vor
- (Kurze) Bearbeitungszeit und Besprechung während des Praktikums
- Termine und Blätter werden über Moodle bekannt gegeben

Vorlesung:

- Fr 11:40 - 13:10 im Raum W3.22 (falls kein Praktikum)
- Fr 14:00 - 15:30 im Raum W3.02

Praktikum:

- SG 1: Freitag, 09:50 - 11:20 Uhr im Raum M2.02 (wenn Praktikumswoche)
- SG 2: Freitag, 11:40 - 13:10 Uhr im Raum M2.03 (wenn Praktikumswoche)
- SG 3: Freitag, 09:50 - 11:20 Uhr im Raum M2.02 (eine Woche später)

- Das Praktikum findet ca. 14-tägig statt.
- Genaue Termine werden über Moodle bekannt gegeben.
- Melden Sie sich bitte im Voraus zur SG1, SG2 oder SG3 selbstständig über Moodle an.

- **Aufgaben dienen zur Vertiefung des Stoffs der Vorlesung**
 - **Keine Abgabe** und Korrektur von Aufgaben
 - Kein bestehen des Praktikums notwendig als Zulassung für Klausur
- **Inhalt des Praktikums ist Teil des Prüfungsstoffs**
 - Aufgaben bereiten auch auf die Klausur vor!
- **Ablauf: Aufgaben werden vor dem Praktikums-Termin veröffentlicht**
 - Vorbereitung der Aufgaben zu Hause ist notwendig
 - Ohne Vorbereitung sind Aufgaben im Praktikum nicht zu schaffen
 - Besprechung der Aufgaben während des Praktikum

- *Andrew S. Tanenbaum, Herbert Bos: **Moderne Betriebssysteme**, 4., aktualisierte Auflage, Pearson, 2016, ISBN: 978-3-86894-279-5 (Buch), 978-3-86326-776-6 (E-Book)*
- *William Stallings: **Operating Systems: Internals and Design Principles, Global Edition**, 9. Auflage, Pearson, 2017, ISBN: 9780134700069*

Organisation

Einführung in Betriebssysteme

Geschichte der Betriebssysteme

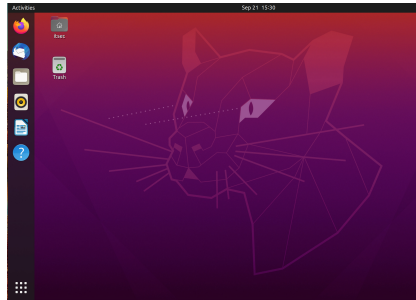
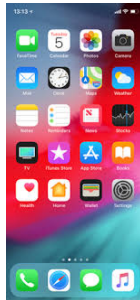
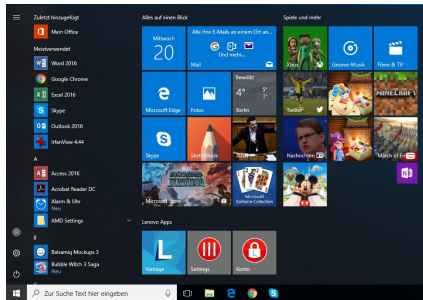
Einführung Praktikum und Programmiersprache C

- **Woran denkt man häufig, wenn man an Betriebssysteme denkt?**



Welche Betriebssysteme kennen Sie? Welche Betriebssysteme haben Sie im Einsatz?

Typische Sicht auf das Betriebssystem



- **Betriebssystem als *erweiterte Maschine***
 - Verwaltung der Hardware
 - Bereitstellen einer benutzerfreundlichen Schnittstelle
- **Betriebsmittelverwalter**
 - Einteilung von Betriebsmitteln
 - Bereitstellen von Ressourcen

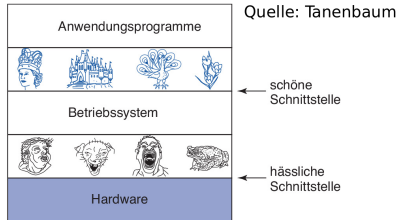


Abbildung 1.2: Betriebssysteme verwandeln die hässliche Hardware in wunderschöne Abstraktionen.

- **Hardware in Computern:**

- CPUs, Speicher, Festplatten, SSDs, Tastatur, Maus, Touchscreen

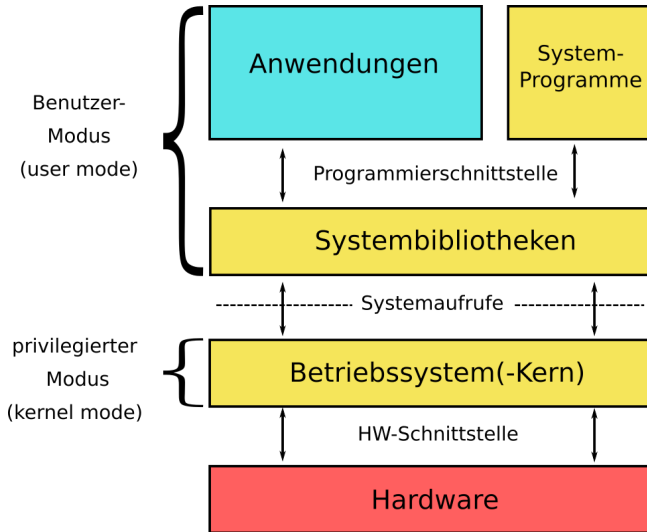
- **Gewünschte Dienstleistungen:**

- Starten von Prozessen
- Lesen / Schreiben von Dateien
- ...

- **Beispiel:**

- Hardware: Schreiben von Daten auf Festplatten wird blockweise adressiert
- Betriebssystem: Stellt Dateisysteme und Dateien zur Verfügung

- **Betriebsmittel und Ressourcen sind:**
 - Speicher, CPUs, Ein- und Ausgabegeräte, Daten
- **Aufteilung zwischen**
 - einzelnen Anwendungen (Prozessen)
 - einzelnen Benutzern des Systems
- **Anforderungen an die Aufteilung:**
 - *Gerechte* Verteilung der verfügbaren Ressourcen
 - Trennung der Umgebungen von Benutzern und Prozessen
 - Schutz vor unberechtigten Zugriffen
- **Das Betriebssystem muss dafür einige Mechanismen implementieren**
 - In dieser Vorlesung werden wir grundlegende Konzepte kennen lernen



- **Verständnis der grundlegenden Aufgaben und Konzepte der Betriebssysteme**
 - Kennenlernen des grundlegenden Aufbaus
 - Verstehen was passiert, wenn man mit einem Betriebssystem interagiert
- **Praktische Anwendung der Dienste des Betriebssystems**
 - Systemnahe Programmierung kennen lernen
 - Nebenläufige Anwendungen entwickeln
- **Wichtig: Beides ist prüfungsrelevant**

- **Grundstrukturen und Arbeitsweisen von Betriebssystemen**
- **Prozesse, Threads und Scheduling**
- **Synchronisation und Kommunikation**
- **Speicherverwaltung**
- **Ein-/Ausgabe**
- **Dateisysteme**

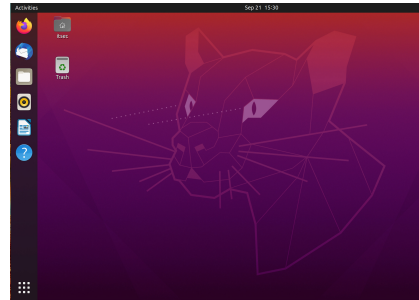
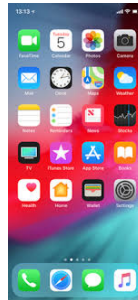
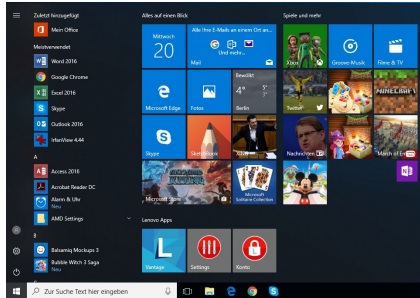
Organisation

Einführung in Betriebssysteme

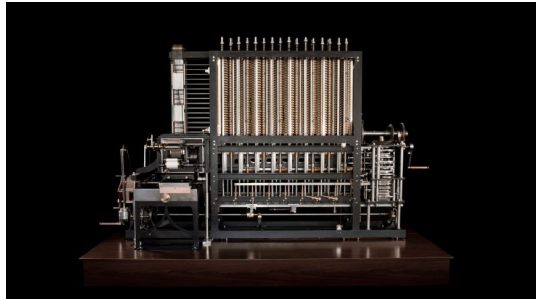
Geschichte der Betriebssysteme

Einführung Praktikum und Programmiersprache C

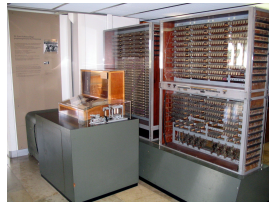
- **Heutige Sicht auf das Betriebssystem:**



- **Entwickelt von Charles Babbage (1791-1871)**
- **Rein mechanische Maschine zur Unterstützung von Berechnungen**
 - Nie von Babbage fertiggestellt
 - Besaß kein Betriebssystem
- **Wichtige Konzepte:**
 - Trennung von Steuer- und Rechenwerk
 - **Erkenntnis: Man benötigt Software**



- **Entwicklungen von Relais- und Röhrenrechnern in verschiedenen Ländern**
 - Colossus, Mark I, ENIAC, Z3
 - Primitive Rechenmaschinen
- **Programmierung durch Stecken von Verbindungen, Steckkarten, Lochkarten, usw.**
- **Keine Programmiersprachen oder Betriebssysteme**



Zuse Z3 (Quelle: Wikipedia)



MARK I (Quelle: IBM Archive)

- **Einführung von Transistoren**
 - Sehr teure Großrechner (Mainframes) → Auslastung wichtig
 - Programmiersprache meistens Fortran
- **Betriebssysteme:**
 - Typische Betriebssysteme: FMS (Fortran Monitoring System) oder IBSYS
 - Unterstützung für das Laden und Starten von Programmen (Jobs)

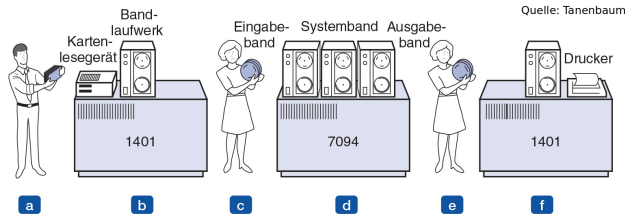
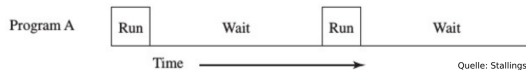
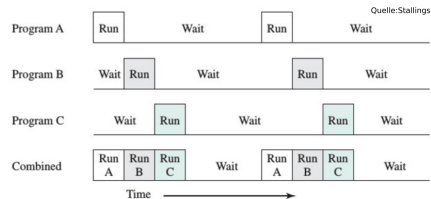
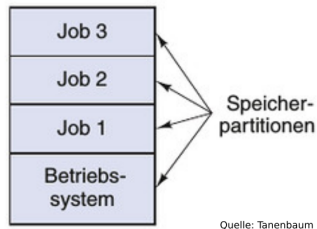


Abbildung 1.3: Ein frühes Stapelverarbeitungssystem. (a) Die Programmierer bringen die Stapel zur 1401. (b) Die 1401 liest den Stapel von Jobs auf ein Band. (c) Ein Operator trägt das Eingabeband zur 7094. (d) Die 7094 führt die Berechnung durch. (e) Ein Operator trägt das Ausgabeband zur 1401. (f) Die 1401 druckt die Ausgabe.

- **Begann mit der Einführung von integrierten Schaltkreisen**
 - Preis-Leistungs-Verhältnis wird besser
 - Familien kompatibler Rechner werden entwickelt
- **Schwächen der alten Generation:**
 - CPUs arbeiten nicht während Programme gelesen oder Ergebnisse geschrieben werden



- **Multiprogrammierung: Ausführung eines anderen Jobs während Wartezeiten**



Mehrere Jobs werden gleichzeitig im Speicher vorgehalten

- **Weitere wichtige Konzepte:**
 - Spooling: Einlesen mehrerer Programme in internen Speicher
 - Timesharing: Ausführen von mehreren Programmen (mehrerer Nutzer) gleichzeitig

- Entwicklung des MIT auf umgebauter IBM 7094 Hardware
- **Kontext-Wechsel** durch Ein/Auslagern einzelner Jobs nach **Timer-Interrupts**



Ein/Auslagern von Jobs in CTSS

- **Gemeinschaftsprojekt von MIT, Bell Labs und General Electric**
 - System mit hunderten Benutzern im Time-Sharing-Betrieb
 - Heute mit Thin-Clients und der Cloud wieder verteten
- **Wegweisende Konzepte für die Entwicklung von Betriebssystemen**
 - Prozesse
 - Virtuelle Speicherverwaltung
 - ...
- **Hohe Komplexität und hohe Kosten**
 - 1969 Ausstieg von Bell Labs und GE
 - Nur ca. 80 Installationen, aber bis in die 90er liefen einige davon noch
- **Viele Entwickler von MULTICS später an anderen Betriebssystemen beteiligt**

- **1969** Ken Thompson, ehemaliger MULTICS Entwickler, startet Implementierung von MULTICS in Assembler auf altem ausrangierten PDP-7-minicomputer
 - System wurde später in **UNIX** benannt
 - Ken Thompson auch bekannt für die Programmiersprache *B* (Vorgänger von *C*) oder die Sprache *go*
- **1974** Portierung auf Programmiersprache C und neue Hardware PDP-11.
- **nach 1974** weite Verbreitung an vielen Hochschulen und Firmen

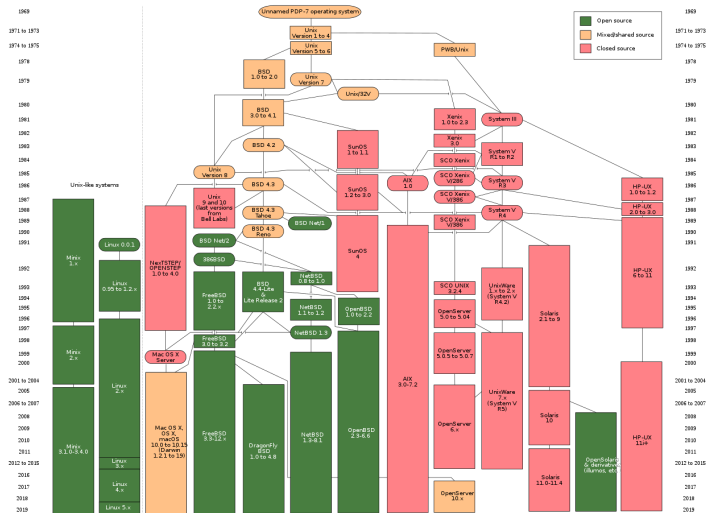


Ken Thompson (Quelle: [Wikipedia](#))



PDP-7 (Quelle: [Wikipedia](#))

Geschichte und Stammbaum von Unix



Quelle: [Wikipedia](#)

- **Ende der 1980er** Viele UNIX-Varianten mit inkompatiblen Schnittstellen
 - Programme für ein Unix können nicht auf anderen UNIXen ausgeführt werden
- **1985: POSIX-Standard**
 - Definierte Schnittstelle (API) zwischen Betriebssystem und Anwendung
 - Funktions-Aufrufe und -Parameter für Kernaufgaben des Betriebssystems
 - Definiert Verhalten von Betriebssystem-Funktionen
 - Aktueller Standard: POSIX.1-2017
- **POSIX heute:** Unterstützung von POSIX auch durch andere Systeme
 - Linux
 - Windows
 - VxWorks
 - ...

- **1974: CP/M (Control Program for Microcomputers)**
 - Intel 8080: 8-Bit-Allzweck CPU für Mikrocomputer
 - Entwickelt für Mikrocomputer mit Diskettenlaufwerk
- **1981: MS-DOS 1.0**
 - Microsoft Disk Operating System
 - Ausgeliefert mit IBM-PCs und BASIC als Programmiersprache
- **1986: MS-DOS 3.0**
 - IBM PC/AT (80286; 8 MHz, bis zu 16 MB RAM)
 - Keine Multiprogrammierung und nur erste Ansätze von virtueller Speicherverwaltung

```
Starten von MS-DOS...
```

```
HIMEM testet den erweiterten Speicher...beendet.
```

```
This driver is provided by Oak Technology, Inc..  
ATI-91X ATAPI CD-ROM device driver, Rev 091X0352  
(C)Copyright Oak Technology Inc. 1987-1997
```

```
Device Name      : CDROM  
Transfer Mode    : Programmed I/O  
Number of drives : 1
```

```
C:\>C:\DOS\SMARTDRV.EXE /X
```

```
MSCDEX Version 2.23
```

```
Copyright (C) Microsoft Corp. 1986-1993. Alle Rechte vorbehalten.
```

```
Laufwerk D: = Treiber CDROM Gerät 0
```

```
C:\>_
```

- **1960er Jahre:** Forschungsarbeiten am Stanford Research Institute
- **1983:** Apple Lisa
- **1984:** Apple Macintosh
- **1984:** X Window System
- **1987:** X11

Hinweis

- Graphische Schnittstellen gehören *prinzipiell* nicht zum Betriebssystem
- Heutige Betriebssysteme liefern jedoch meistens eine graphische Bedienoberfläche

- **1993:** Windows NT 3.1
 - Entwurf eines komplett neuen Betriebssystems
 - Hoffnung mit Windows NT 3.1 die MS-DOS-basierten Windows Systeme zu verdrängen
- **1996:** Windows NT 4.0
 - Neue Benutzeroberfläche mit Ähnlichkeit zu Windows 95
 - Großer Erfolg, insbesondere in Unternehmensnetzwerken
- **2000:** Windows 2000
 - Erweiterungen wie *Active Directory* und Dateisystem *NTFS*
 - Unterstützung neuer Hardware wie *USB* oder *FireWire* und Einführung von *Plug-and-Play*
- **2002:** Windows XP
- **2009:** Windows 7
- **2015:** Windows 10
- **2021:** Windows 11

- Mitte der 90er Jahre wurden PDA (Personal Digital Assistant) und Smart Phones eingeführt
- Einführung des iPhones 2007 sorgte für weite Verbreitung von Smart Phones
- iPad sorgte für weite Verbreitung von Tablets
- Heutige Betriebssysteme:
 - Android
 - iOS
- Implementieren gleiche Konzepte wie Betriebssysteme klassischer PCs



- **Geschichte der Computer und Betriebssysteme entwickelt sich seit den 1940er Jahren**
- **Aufgabe der Betriebssysteme ist es**
 - Hardware und Betriebsmittel zu verwalten
 - Betriebsmittel an Prozesse und Anwender gerecht zu verteilen
 - Eine einheitliche Schnittstelle für Anwendungen und Benutzer bereit zu stellen
- **Moderne Betriebssysteme bieten ähnliche Funktionen und Schnittstellen. Sie setzen ähnliche Konzepte zur Verwaltung von Betriebsmitteln um**
- **Im Rahmen der Vorlesung werden grundlegende Methoden und Konzepte moderner Betriebssysteme vorgestellt**

Organisation

Einführung in Betriebssysteme

Geschichte der Betriebssysteme

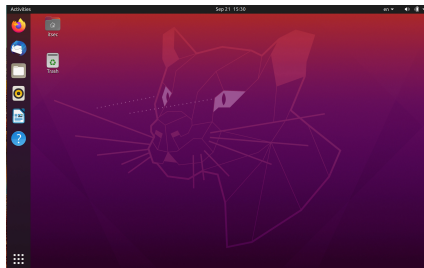
Einführung Praktikum und Programmiersprache C

- **Praktikum vertieft Themen der Vorlesung**

- Programmieraufgaben aus dem Bereich Betriebssystem
- Übungen zur Anwendung einzelner Algorithmen auf "Papier"

- **Umgebung: Linux**

- Virtuelle Maschine mit entsprechender Umgebung im Moodle bereitgestellt
- Erster Praktikumstermin: Einführung in Linux und aufsetzen der Umgebung



- **Assembler war lange Zeit die Sprache der Programmierung von Betriebssystemen**
 - Betriebssysteme wurden lange Zeit für eine bestimmte Computer-Hardware geschrieben
 - Bei Wechsel der Hardware musste das Betriebssystem portiert werden
- **C wurde als Hochsprache für Betriebssysteme in den 1970ern eingeführt**
 - Routinen für Systemstart werden in Assembler geschrieben
 - Übergeordnete Funktionalität wird in C geschrieben
 - Compiler übernehmen die Übersetzung in Binär-Code für unterschiedliche Prozessoren
- **C erlaubt direkten Zugriff und Manipulation auf Speicher**
 - Hardware-nahe Programmierung möglich
 - Erlaubt ebenfalls die Programmierung von Treibern

- **Sprachsyntax**

- Keine Klassen, keine Packages
- Kein Exception-Handling - Behandlung von Fehlern über Rückgabewerte von Funktionen

- **Speicherverwaltung**

- Speicher in Java wird über *Garbage Collection* automatisch freigegeben
- Speicher in C wird manuell allokiert und freigegeben

- **Direkter Speicherzugriff über Zeiger**

- Impliziter Zugriff auf Werte/Referenzen
- Explizit durch Angabe der Speicherstelle an der Objekt zu finden ist

Hello World in C

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

Übersetzen und Ausführen

```
$ gcc -o hello hello.c
$ ./hello
Hello world!
```


- **Deklaration:**

- `int i, *pi;`
 - Deklariert die Variable *i* als Integer
 - Deklariert den Zeiger *pi*, der auf Speicherbereich mit einem Integer-Wert zeigt

- **Adressen von Variablen in Zeigern speichern (referenzieren):**

- `pi = &i;`
 - *Pointer* zeigt nun auf die Adresse, an der der Inhalt der Variable *i* steht

- **Auf Speicherinhalte zugreifen (dereferenzieren):**

- `*pi = 7*7; // jetzt ist i == 49`
- `i = *pi * 3; // jetzt ist i == 147`

```
#include <stdio.h>

int func_call_by_value(int i)
{
    i = i+i;
    return i;
}

void func_call_by_pointer(int* pi)
{
    *pi = *pi + *pi;
    return;
}

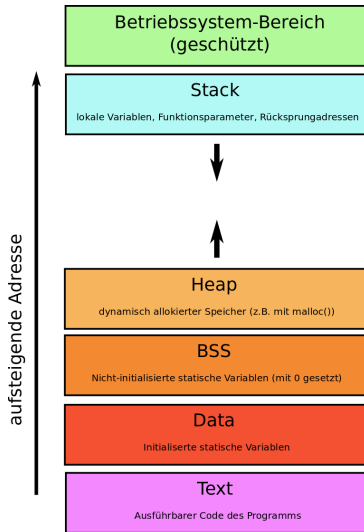
int main(int argc, char** argv)
{
    int n = 3;
    printf("%d\n", n);
    int j = func_call_by_value(n);
    printf("%d %d\n", j, n);
    func_call_by_pointer(&n);
    printf("%d\n", n);
}
```

• Zwei Funktionen

- *func_call_by_value* nimmt einen Integer als Parameter. Rückgabe der Berechnung über Rückgabewerte der Funktion
- *func_call_by_pointer* übernimmt Zeiger. Es werden direkt die Werte im Speicher manipuliert.

Ausführung des Programms

```
$ ./pointer-example
3
6 3
6
```



```
int func()
{
    // Deklariert ein Array bestehend aus 10 Integer-Werten
    // Das Array wird zu Beginn der Funktion allokiert und am Ende
    // der Funktion automatisch freigegeben
    int array[10];

    int i, *pi;

    // Zugriff auf Elemente des Arrays mittels Indizes in das Array
    // Indexierung beginnt bei 0
    for (i = 0; i < 10; ++i)
        array[i] = 1;

    // Zugriff auf die Elemente des Arrays durch Zeiger
    int* ptr = array;
    for (i = 0; i < 10; ++i) {
        *ptr = 2;
        ptr++;
    }

    //Zugriff auf das Array als Zeiger
    for (i = 0; i < 10; ++i)
        *(array + i) = 3;
}
```

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

int main(int argc, char** argv)
{
    int i;
    int *dyn_array, *pi;
    // Dynamische Allokation eines Arrays mit 10 Elementen

    dyn_array = (int*)malloc(10*sizeof(int));
    // Hat Allokation funktioniert?
    if (dyn_array == NULL) {
        perror("Failed to allocate array");
        exit(0);
    }

    // Indexierung des Arrays mit den gleich Methoden wie statische Arrays
    for (i = 0; i < 10; ++i) dyn_array[i] = 1;
    for (pi = dyn_array; pi < dyn_array + 10; ++pi) *pi = 2;

    free(dyn_array);
}
```

- **Primitive Datentypen**

- `int`, `short`, `long`, `float`, `double`, `char`
- Länge der Datentypen abhängig von Prozessor-Plattform
 - z.B. `long` auf 32-Bit Systemen oft 4 Bytes, auf 64-Bit Systemen oft 8 Bytes
- Datentypen mit positiven und negativen Zahlen
- Nur positive Zahlen: `unsigned int`, `unsigned short`, ...

- **Boolean**

- Kein eigener Boolean Datentyp!
- Tests auf Zahlenwerte werden für boolsche Tests verwendet: `0 == false`, alles andere `true`

- **Strings:**

- Strings werden in `char*`-Arrays gespeichert
- Ein Wert von 0 in einem Character zeigt das Ende des Strings an

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

#define STRLEN 100

struct mystruct {
    int somevalue;
    char* somestring;
};

int main(int argc, char** argv)
{
    struct mystruct first;

    first.somevalue = 20;
    first.somestring = (char*)malloc(STRLEN);
    strncpy(first.somestring, "Hello World", STRLEN);

    printf("Values: %d / %s\n", first.somevalue, first.somestring);

    free(first.somestring);
}
```

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

#define STRLEN 100

struct mystruct {
    int somevalue;
    char* somestring;
};

int main(int argc, char** argv)
{
    struct mystruct* second = (struct mystruct*)malloc(sizeof(struct mystruct));

    second->somevalue = 10;
    second->somestring = (char*)malloc(STRLEN);
    strncpy(second->somestring, "Hello World2", STRLEN);
    printf("Values: %d / %s\n", second->somevalue, second->somestring);

    free(second->somestring);
    free(second);
}
```


- **Kommentare**

- Einzelne Zeilen eingeleitet durch //
- Kommentare über mehrere Zeilen durch */* ... */*

- **Präprozessor-Makros**

- Präprozessor-Anweisungen definieren Konstanten, die vor Kompilierung ersetzt werden
- *#include<string.h> // fügt header string.h an stelle im Text ein*
- *#define FOO 1*
int i = FOO; // Initialisiert die Variable i mit dem Wert 1

- **Erlaubte Sprachkonstrukte abhängig von C-Standard**

- Mehrere Versionen von C wurden über die Jahre standardisiert. Aktuellste Version: C17
- Wichtigste Einschränkungen alter Versionen:
 - Deklarationen von Variablen nur zu Beginn der Funktion
 - Keine Inline-Kommentare über Zeichenkette //

- **Betrifft: C-Funktionen aus der Standard-Bibliothek und aus dem POSIX-Standard**
 - Dokumentation kann online gefunden werden
 - Auf Unix-Systemen oft lokal vorhanden: abfragbar über das Befehl *man*
 - Beispiel: *man strncpy*

```
STRCPY(3)                                Linux Programmer's Manual                                STRCPY(3)

NAME
    strcpy, strncpy - copy a string

SYNOPSIS
    #include <string.h>

    char *strcpy(char *dest, const char *src);
    char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
    The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. Beware of buffer overruns! (See BUGS.)

    The strncpy() function is similar, except that at most n bytes of src are copied. Warning: If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

    If the length of src is less than n, strncpy() writes additional null bytes to dest to ensure that a total of n bytes are written.

    A simple implementation of strncpy() might be:

    char *
    strncpy(char *dest, const char *src, size_t n)
    {
        size_t i;

        for (i = 0; i < n && src[i] != '\0'; i++)
            dest[i] = src[i];
        for (; i < n; i++)
            dest[i] = '\0';
    }

Manual page strncpy(3) line 1 (press h for help or q to quit)
```

- **Übersetzung des Programms in zwei Schritten**
- **Compiler**
 - Übersetzt den Programm-Code in Objekt-Dateien
- **Linker**
 - Fügt Objekt-Dateien und Bibliotheken zu ausführbarem Programm zusammen
 - Verwendete Bibliotheken (außer Standardbibliothek) müssen angegeben werden
- **Beispiel-Kompilierung:**
 - `gcc -Wall -Werror -O0 -g -lpthread -o generated_file source_file.c`
 - Generiert Programm *generated_file* aus *source_file.c* und bindet Bibliothek *libpthread* ein

- GCC ist ein Open-Source Compiler → Verwenden wir für C-Kompilierung
- Compiler-Schalter, die wir immer benutzen:
 - *-Wall* - Alle Warnungen einschalten
 - *-Werror* - Warnungen wie Fehler behandeln
 - *-O0* - Optimierungen des Compilers ausschalten
 - *-g* - Debug-Informationen hinzufügen
- Beispiel: `gcc -Wall -Werror -O0 -g -o test test.c`

- *make* ist ein Werkzeug, um den Compile&Link-Prozess zu vereinfachen
- *make* benötigt dazu Instruktionen im sogenannten *Makefile*
- Entweder mit *-f <filename>* angeben, oder (besser) *make* sucht eine Datei namens *makefile* oder *Makefile* im aktuellen Verzeichnis
- *Makefiles* bestehen aus Regeln (rules), die angeben was zu tun ist, wenn sich bestimmte Dateien ändern
- *Makefiles* haben meist mehrere Regeln

Makefile

```
target: prerequisites ...  
    recipe  
    ...
```

- target: Output
- prerequisites: Inputs
- recipe: was zu tun ist (Achtung: Es muss ein *TAB* vor receipe sein, sonst Fehler)

Makefile

```
foo.o: foo.c defs.h  
    gcc -Wall -Werror -o foo.o foo.c
```

- *make* ist intelligent und führt Regeln nur aus, wenn es auch nötig ist
- Im Beispiel: wenn *foo.c* und *defs.h* nicht geändert wurden, seit *foo.o* das letzte Mal gebaut wurde, wird die Regel *foo.o* nicht ausgeführt
 - Kompilieren nur wenn *foo.o* nicht existiert, oder *foo.o* älter als *foo.c* oder *defs.h* (also letzte Modifizierung) ist
- **Ausführen mit**
 - *make*
 - oder *make -f <filename>*
 - oder *make target*

- Bei phony targets, ist das target nicht wirklich der Name einer Datei
- Gängige phony targets sind *clean*, *all*, ...

Funktioniert nicht wenn Datei Namens clean existiert

```
clean:
    rm *.o temp
```

PHONY sorgt dafür dass clean immer ausgeführt wird

```
.PHONY: clean
clean:
    rm *.o temp
```


- Wir sind alle fehlbar ... darum gibt es Tools!
- Aber: “A fool with a tool is still a fool!”, sprich man muss trotzdem wissen was man tut (und was das Tool tut und auch nicht)
- Zwei wichtige Werkzeuge sind statische Code-Analyse und dynamisch (Laufzeit-)Analyse
 - *cppcheck* (statisch)
 - *valgrind* (dynamisch)

- Überprüft Quellcode auf Fehler
- Aber: *cppcheck* findet nicht alle Fehler!
- Verwendung: *cppcheck code.c*
- Output:
 - Checking code.c...

code.c:4 : (error) Array 'a[10]' index 10 out of bounds

code.c

```
int main() {  
    char a[10];  
    a[10] = 0;  
    return 0;  
}
```

- Überprüft Programm zur Laufzeit
- *valgrind* bietet viele mögliche Analysen
 - Wir beschränken uns auf Speicher
- Benutzung: *valgrind ./prog*
- *valgrind* Output:
 - ...
 - ==5470== Use of uninitialised value of size 8
 - ==5470== at 0x4004F1: main (code.c:3)
 - ...

code.c

```
int main() {  
    int* a;  
    *a = 1;  
    return 0;  
}
```

- **Inhalt:**
 - Kurz-Einführung in Linux-Umgebung der VM
 - Wiederholung / Einführung in C unter Linux
- **Vorbereitung:**
 - Laden Sie die virtuelle Maschine
 - Vor dem Termin sollten Sie in der Lage sein die Maschine zu starten und sich einzuloggen

Fragen?