Informatik 7bi Schuljahr 2017/2018 als PDF exportieren

Informatik 7. Klasse - Schuljahr 2017/18

Lehrinhalte

- Lehrplaninhalte

Materialien

- Mappe mit 4 Trennblätter (pro Lehrstoffbereich)
- (Optional) Weitere Trennblätter für Kapitel

Remote-Zugriff auf Schulserver

Kapitel

- 1) Betriebssysteme
- 2) Virtualisierung
- 3) C++

Leistungsbeurteilung

- **Schularbeit (SA)**
  - 1x SA (2h) pro Semester
- **Mitarbeit (MA)**
  - Aktive Mitarbeit im Unterricht (aMA)
  - Mündliche Stundenwiederholungen (mMA)
  - Schriftliche Stundenwiederholungen (sMA)
- **Praktische Arbeiten (PA)**
  - 1x praktischer Arbeitsauftrag pro Woche
- Aktueller Leistungsstand

Schularbeitsstoff für die 1. SA in Informatik – 7BI – 24.11.2017

- Hardware
  - Unterschied HW vs. SW
  - EVA-Prinzip (engl. IPO-Model)
  - Von-Neumann-Architektur (VNA)
    - Komponenten

        - Prinzipien
- Betriebssysteme
    - Allgemeines
        - Definition, Ziele, Aufgaben
        - Schichtenmodell, OS als Schnittstelle
        - Betriebssystemarten
        - Historische Entwicklung von Computersystemen
        - Partitionierung & Formatierung
            - 3 Stufen der Formatierung
                - Physikalische Formatierung (Grundbausteine, Adressierungsarten)
                - Partitionierung (MBR, Partitionsarten, Gründe)
                - Logische Formatierung (Schnell/Normalformatierung, Dateisysteme)
            - Dateisysteme (Windows, Linux)
            - Planung eines Systems
            - Betriebssysteme
        - Windows
            - MS-DOS
            - Editionen
            - Dateiberechtigungen
            - Benutzer & Gruppen
            - Sicherheit und Wartung (Automatische Sicherung, Systemabbild erstellen, Systemwiederherstellung, Live CD)
            - Batch-File erstellen
        - Linux
            - Betriebssystemaufbau, Distribution, Kernel
            - Grafische Oberflächen
            - Benutzertypen
            - Verzeichnisstruktur
            - Linux Prompt
            - Dateiarten
            - Dateirechte & Sonderrechte
            - Benutzer & Gruppenverwaltung
            - Inodes
            - Links (Hard- & und Softlinks)
            - BASH / Shell / Terminal - Befehle
            - Pipes
            - Shell-Scripts

Was wird in der 7. Klasse gemacht?

Informatiksysteme - Hardware, Betriebssysteme, Vernetzung

Hardware

- Wiederholung der wichtigsten Komponenten

Betriebssysteme

- Von der Hardware zum Betriebssystem
- Rechnerarchitekturen
- Ziele des Betriebssystems
- Aufgaben des Betriebssystems
- Entwicklung des Betriebssystems
- Betriebssystemarten
- Planung eines Systems
- Formatierung/Partitionierung
- Windows
- Linux

Virtualisierung

- Definition
- Abgrenzung zur Emulation
- Anwendungsbereiche von Virtualisierung (Hardware, Software, Netzwerk)
- Terminologie (Reale Maschine, Virtuelle Maschine, Host-OS, Gast-OS, VM Monitor, Emulation)
- Partitionierung/Ressourcenaufteilung
- Vorteile/Nachteile
- Varianten der Virtualisierung (Typ 1, Typ 2, Paravirtualisierung)

Netzwerke

- Grundlagen (Größenordnungen, Architekturen, Topologien..)
- Übertragungsmedien
- Netzwerkgeräte
- Netztypen (Peer-to-Peer, Server-Client)
- Netzwerkbefehle
- Netzwerkadressierung (Netzklassen, IP-Adresse, Subnetz,Ports,...)
- OSI Modell
- Netzwerkdienste/Protokolle

Angewandte Informatik, Datenbanksysteme und Internet

PHP

- PHP Definition
- PHP Grundlagen
- Variablen und Datentypen
- Formulare
- Kontrollstrukturen
- Schleifen
- Arrays (indiziert, assoziativ, mehrdimensional)
- Dateimanagement
- Sessions & Cookies
- Funktionen

Datenbanksystem - MySQL

- Definition und Anwendung
- Eigenschaften eines DBS/ Unterschied zu Tabellenkalkulation
- Datenbank planen (ER-Diagramm)
- SQL als Datenbanksprache
- PHP Anbindung

Wiederholung Textverarbeitung

- Alternativen zu MS Word
- Grundlagen
- Spalten und Tabelllen
- Kopf- und Fußzeile, Seitenzahl
- Formatvorlagen
- Abbildungsverzeichnis, Inhaltsverzeichnis
- Serienbriefe

Algorithmik und Programmierung

HTML & CSS

- Seitenaufbau planen
- Navigationsleisten erstellen mit HTML und CSS
- Bereiche definieren und positionieren
- Dateimanagement
- Dateistrukturen
- Klassen und Objekte
- Grafikelemente (Widgets)

Javascript

- Hintergründe
- Einfache Befehle
- Grundlegende Sprachelemente
- Kontrollstrukturen
- Funktionen
- Objekte
- Vordefinierte Objekte
- Objektmodell
- Zugriff auf HTML-Dokumente
- Event-Handler
- Unsere Scripts

C++

- Pointer
- Dateimanagement
- Dateistrukturen
- Klassen und Objekte
- Grafikelemente (Widgets)

Gesellschaftliche Aspekte der Informationstechnologie

Bedeutung der Informatik in der Gesellschaft

- Datensicherheit
- Datenschutz
- Urheberrecht

```
7. Klasse (3 Stunden,  je eine 2h Schularbeit pro Semester)
```

Kapitel Betriebssyteme als PDF exportieren

Betriebssysteme

Einführung Betriebssysteme

Wiederholung Hardware

Wortherkunft

„Hardware“ kommt ursprünglich aus dem Englischen und bedeutet übersetzt Eisenwaren.

Hardware vs. Software

- Hardware = sind alle greifbaren/sichtbaren Elemente eines PCs
- Software = Programme und Daten, also nicht greifbare Elemente eines PCs

Komponenten

- **Grundbestandteile**
  - Motherboard/Mainboard - Hauptplatine
  - Central Processing Unit (CPU) - zentrale Recheneinheit (Prozessor)
  - Random Access Memory (RAM) - Arbeitsspeicher
- **Massenspeicher**
  - Festplatte (SSD, SATA)
  - Laufwerke (CD, DVD, Band,..)
- **Erweiterungskarten** (optional)
  - Grafikkarte
  - Soundkarte
  - Netzwerkkarte
- **Peripheriegeräte**
  - Eingabegeräte
    - Maus & Tastatur
    - Scanner
  - Ausgabegeräte
    - Bildschirm
    - Drucker

EVA/IPO Prinzip

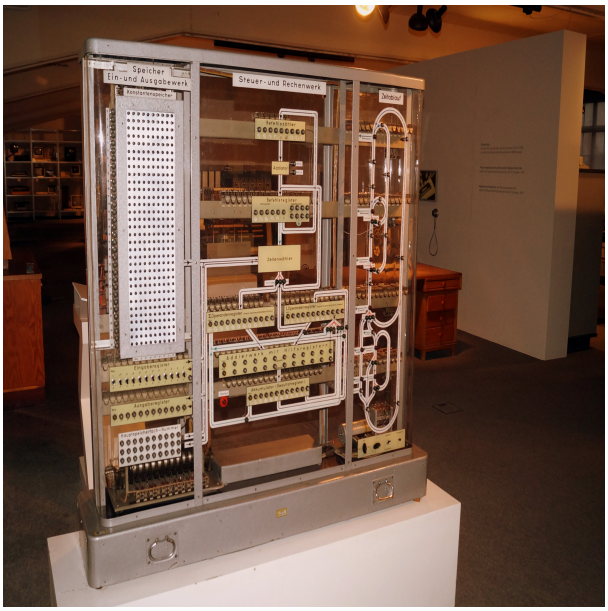
Das **EVA**-Prinzip ist ein Grundprinzip für die Datenverarbeitung und beschreibt die Reihenfolge in der Daten verarbeitet werden.

Eingabe (**I**nput) - **V**erarbeitung (**P**rocess) - **A**usgabe (**O**utput)

EVA Prinzip

Von-Neumann-Architektur (VNA)

Die **V**on-**N**eumann-**A**rchitektur ist ein Modell für Computer und bietet die Grundlage für alle heutigen Computer. Das Modell wurde von Johann von Neumann im Jahr 1945 entwickelt. Heute ist Johann von Neumann unter seinem amerikanischen Namen John von Neumann bekannt.

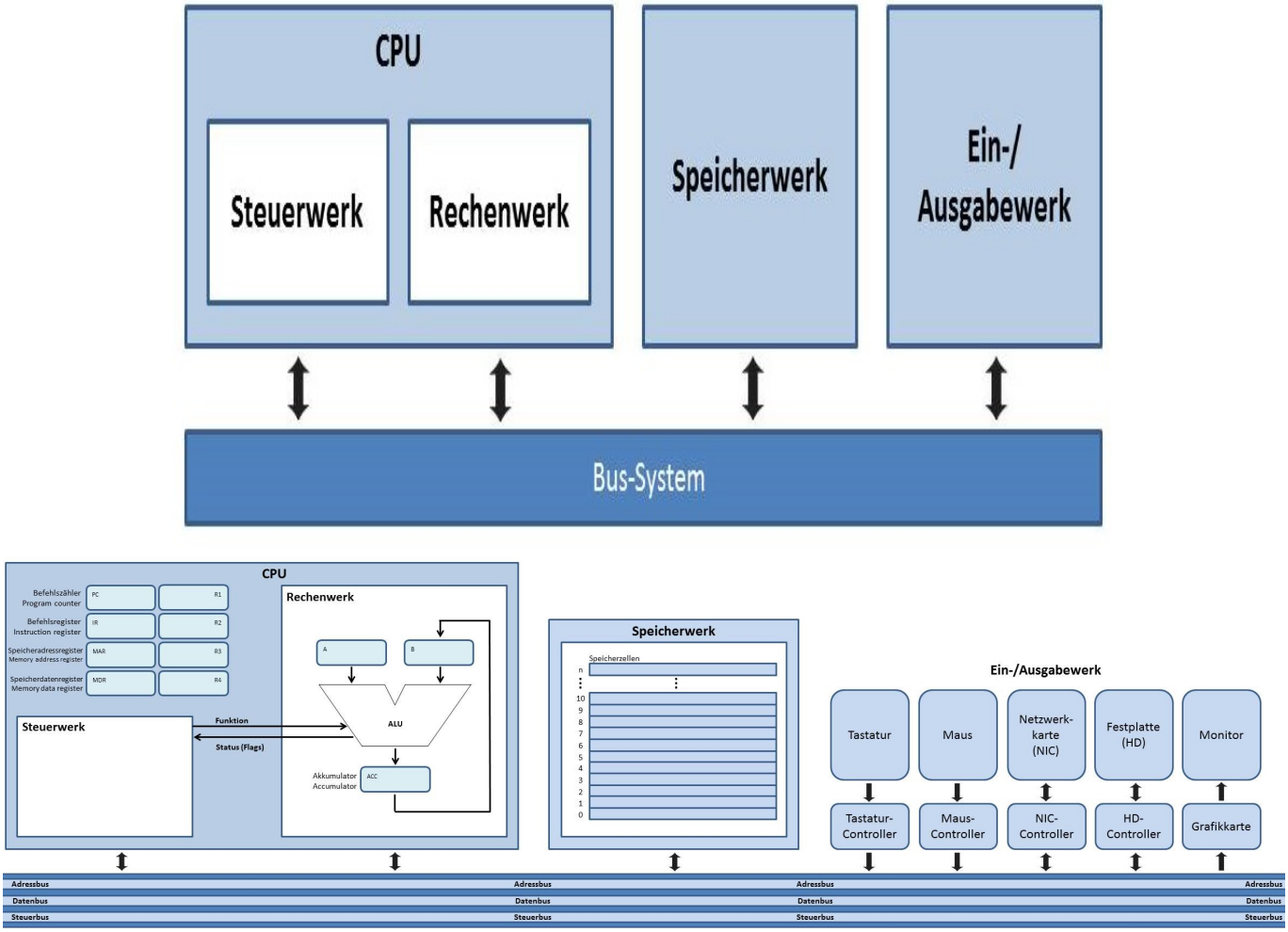


Er revolutionierte die bisherigen Computer, da durch seine Architektur verschiedene Programme auf derselben Hardware laufen konnten.

Komponenten

Ein Von-Neumann-Rechner beruht auf folgenden Komponenten, die bis heute in Computern verwendet werden:

- ALU (Arithmetic Logic Unit) – Rechenwerk, selten auch Zentraleinheit oder Prozessor genannt, führt Rechenoperationen und logische Verknüpfungen durch. (Die Begriffe Zentraleinheit und Prozessor werden im Allgemeinen in anderer Bedeutung verwendet.)
- Control Unit – Steuerwerk oder Leitwerk, interpretiert die Anweisungen eines Programms und verschaltet dementsprechend Datenquelle, -senke und notwendige ALU-Komponenten; das Steuerwerk regelt auch die Befehlsabfolge.
- Bussystem: Dient zur Kommunikation zwischen den einzelnen Komponenten (Steuerbus, Adressbus, Datenbus)
- Memory – Speicherwerk speichert sowohl Programme als auch Daten, welche für das Rechenwerk zugänglich sind.
- I/O Unit – Eingabe-/Ausgabewerk steuert die Ein- und Ausgabe von Daten, zum Anwender (Tastatur, Bildschirm) oder zu anderen Systemen (Schnittstellen).



Register

- **Befehlszähler (Program Counter - PC)**

Enthält die Speicheradresse vom Speicherwerk des aktuellen Befehls. (Startadresse = 0x0000). Wird nach jedem Befehl um 1 erhöht.

- **Befehlsregister (Instruction Register - IR)**

Speichert den vom Speicherwerk zurückbekommenen Befehl.

- **Speicheradressregister (Memory Address Register - MAR)**

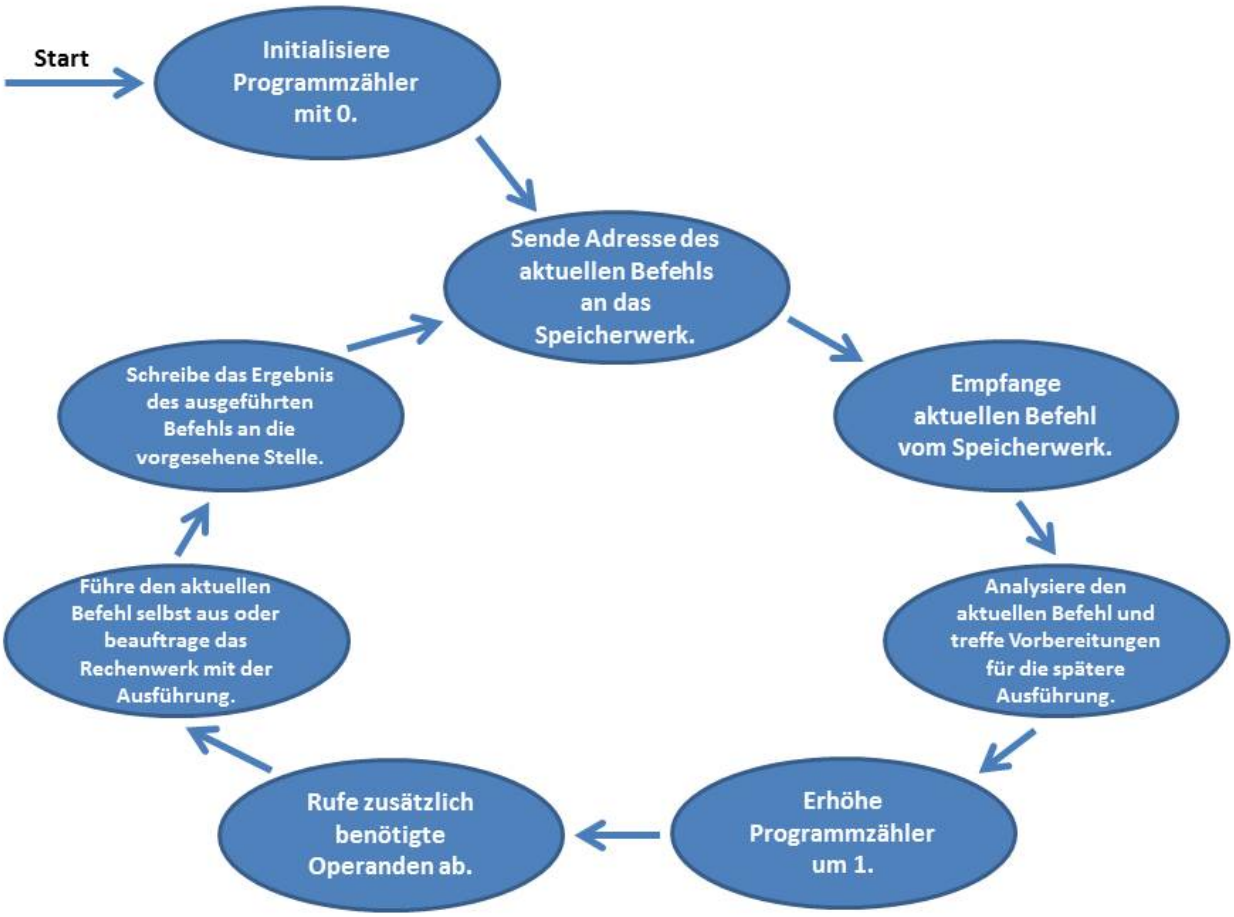
Ausschließlich für die Kommunikation zwischen Steuerwerk und Rechenwerk. Im MAR legt das Steuerwerk jeweils die Adresse ab, welche im Speicherwerk angesprochen werden soll.

- **Speicherdatenregister (Memory Data Register - MDR)**

Ausschließlich für die Kommunikation zwischen Steuerwerk und Rechenwerk. Bei einem Lesezugriff auf die Speicherzelle wird der vom Speicherwerk über den Datenbus bereitgestellte Wert im MDR abgelegt und kann von hier aus weiter verarbeitet werden. Bei einem Schreibzugriff muss sich im MDR der zu schreibende Wert befinden, so dass er über den Datenbus an das Speicherwerk übermittelt werden kann.

Prozesszyklus

1. Initialisiere das Befehlszählerregister (Program Counter- PC) mit 0 (Start)
2. Sende Adresse des aktuellen Befehls zum Speicherwerk
3. Empfange aktuellen Befehl vom Speicherwerk und speichere diesen in das Befehlsregister (Instruction Register - IR)
4. Analysiere aktuellen Befehl und treffe Vorbereitungen für die spätere Ausführung (Welcher Befehl und was ist dazu notwendig?)
5. Erhöhe den Befehlszähler (PC) um 1
6. Rufe zusätzlich benötigte Operanden ab (z.B.: Befehl ADD OP1 OP2)
7. Führe den Befehl selbst (Steuerwerk) aus oder beauftrage das Rechenwerk für die Ausführung
8. Schreibe das Ergebnis des ausgeführten Befehls an die vorgesehene Stelle



Arbeitsweise des Steuer- und

Rechenwerks

Arbeitsweise des Speicherwerks

Befehlszähler und Befehlsregister im Zusammenspiel mit dem Bus-System

Arbeitsweise des Steuerwerks

Die 7 Prinzipien der Von-Neumann-Architektur

- Rechner besteht aus fünf Funktionseinheiten
- Struktur des Rechners ist unabhängig vom zu bearbeitenden Problem. Zur Lösung eines Problems muss Programm im Speicher abgelegt werden.
- Programme, Daten und Ergebnisse werden im selben Speicher abgelegt.
- Der Speicher ist in fortlaufenden nummerierten Zellen unterteilt. Über die Adresse einer Speicherzelle kann auf den Inhalt zugegriffen werden.
- Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgende Speicherzellen abgelegt.
- Durch Sprungbefehle kann von der gespeicherten Befehlsreihenfolge abgewichen werden.
- Es gibt zumindest
    - arithmetische Befehle (Addition, Subtraktion, Multiplikation)
    - logische Befehle (EQUAL, NOR, AND, OR)
    - Transportbefehle, z.B. von Speicher zu Rechenwerk und für Ein- und Ausgabe
- Alle Daten (Befehle, Adressen, usw.) werden binär kodiert

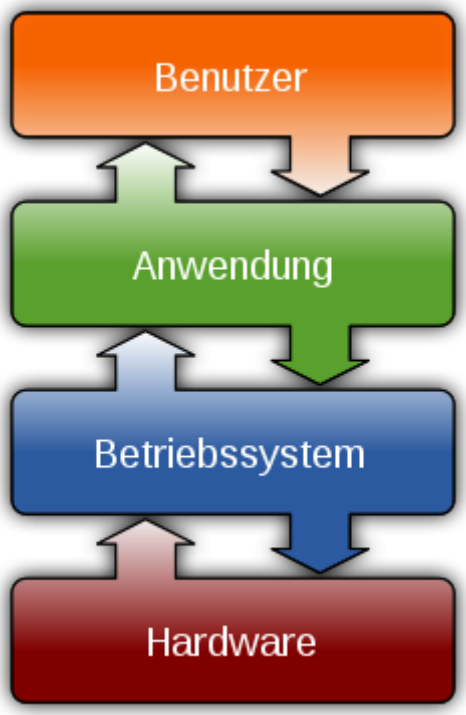
Zusammenhang Hardware - Betriebssysteme

Grundlegendes

Ein Rechner besteht zunächst aus den sichtbaren und greifbaren Komponenten der Hardware. Damit aber mit dem Rechner auch gearbeitet werden kann, benötigt er noch geeignete Programme. Nach dem Von-Neumannschen Prinzipien ist ein Rechner als Universalrechner konstruiert. Das heißt die Erledigung unterschiedlicher Aufgaben wird ausschließlich über Programme (auch Software genannt) gesteuert.

<steps> Da sich aber nicht jeder Softwareentwickler um die Details zur Veraltung der einzelnen Hardwarekomponenten (z.B.: Steuerungsinformationen für Plattenspeicher, Speicherverwaltung, Prozessverwaltung etc.) kümmern sollte, erscheint es sinnvoll diese Komponenten zentral und damit allgemein verfügbar bereitzustellen. Andernfalls würde jeder Entwickler viel Zeit verschwenden.

<step> Somit entwickelte man eine Ebene (=Betriebssystemebene) auf der verschiedenste Verwaltungsprogramme bereits laufen und allen Anwendern bzw. Anwendungsprogrammen eine vereinfachte Sicht auf die Hardware zur Verfügung stellen.



<step> Folgendes gilt, dass das Betriebssystem quasi der Mittler (=Übersetzer) zwischen Hardware und Benutzer ist. Dadurch kann die Komplexität der darunterliegenden Systemarchitektur verborgen werden und dem Anwender wird eine einfache und verständliche Schnittstelle (Interface) angeboten. Er kann sich somit voll und ganz auf die Implementierung seiner Funktion konzentrieren und muss sich nicht mit komplexen Maschinenbefehlen herumschlagen.

<step> Des Weiteren verwaltet das Betriebssystem die Ressourcen des Rechners, also alle physikalischen Geräte (Prozessoren, Speicher, Grafikkarte,..). Es teilt also die Ressourcen des Gesamtsystems gerecht an die konkurrierenden Anwenderprogramme auf. Ohne einer solchen

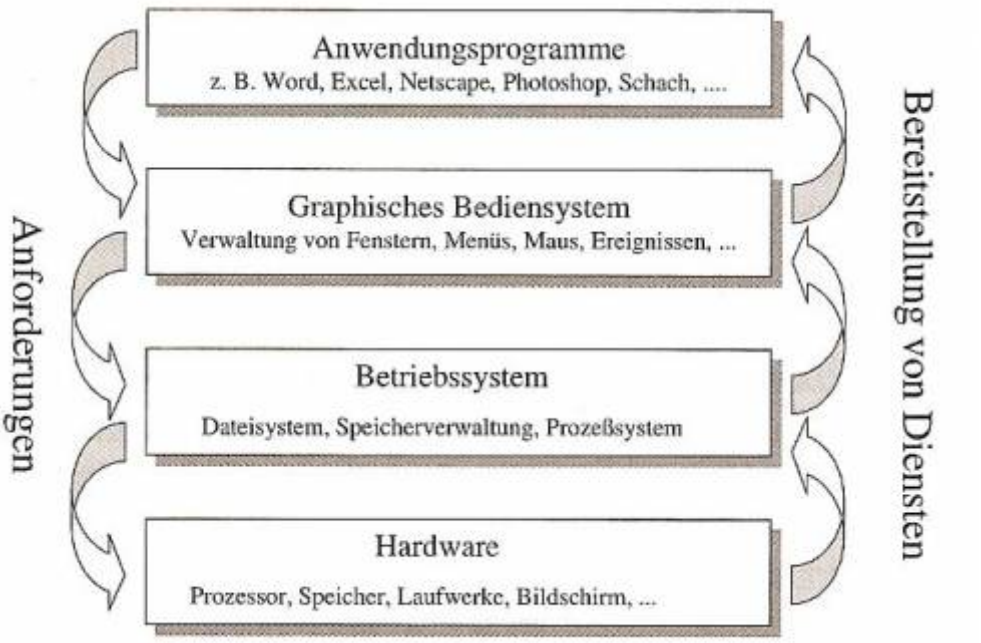
Aufteilung wäre eine sinnvolle Benützung des PCs nicht möglich.

<step> Als normaler Computeranwender befinden wir uns auf der **Benutzerebene**, wo der Rechner als Arbeitsgerät auch für technische Laien dienen muss.

<step> Zwischen dem vom Anwender intuitiv zu bedienenden Rechner und den Fähigkeiten der Hardware klafft eine riesige Lücke, die vom

- Betriebssystem (Datei-, Prozess- und Speicherverwaltung) und dem
- grafischen Bediensystem (Menüs, Fenster, Maus)

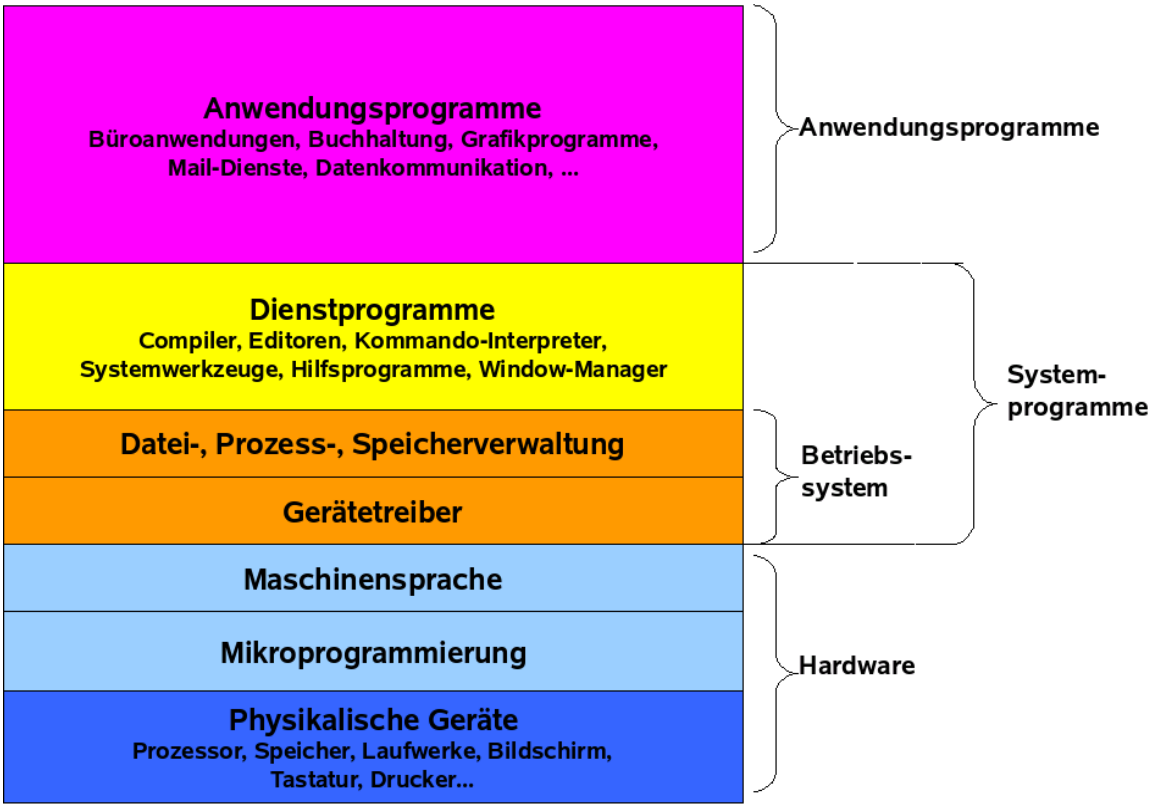
überbrückt wird.



<step> Jede Schicht fordert von der niedrigeren Schicht Dienste an. Diese wiederum benötigt zur Erfüllung der Anforderungen selber Dienste von der nächsten Schicht. Auf diese Weise setzen sich die Anforderungen in die tiefen Schichten fort, bis die Hardware zu geeingeten Aktionen veranlasst wird.

Anders betrachtet, bietet jede Schicht der jeweils höherliegenden Schicht Dienste an. </steps>

Schichten eines Computersystems



Physikalische Geräte

Die unterste Schichte enthält die physikalischen Geräte. Sie besteht aus integrierten Schaltungen, Drähten, Stromversorgung usw.

Mikroprogrammierung/Mikrocode

Aufgabe des Mikrocodes ist die direkte Steuerung der physikalischen Geräte. In Microcode entworfene Programme werden von einem Interpreter in entsprechende Steueranweisungen übersetzt und an die Geräte übermittelt. Die Instruktionen der Maschinensprache (z.B.: ADD, MOVE, JUMP,...) werden in einzelne kleine Microcodeprogramme übersetzt.

Maschinensprache

Die Menge von Instruktionen die das Mikroprogramm ausführen kann wird als Maschinensprache bezeichnet. Maschinensprache besteht aus zahlreichen elementaren Maschinenbefehlen die im Prozessor in Microcode zu einem Programm übersetzt und ausgeführt werden. Wichtig ist, dass jeder Maschinenbefehl durch ein festgelegtes Mikroprogramm implementiert bzw. fest verdrahtet ist.

Da die Maschinensprache hardwarespezifisch ist, wird sie noch der Hardware zugeordnet.

Betriebssystem

Das Betriebssystem vermindert die Komplexität der Hardware und deren Verwaltung um dem Programmierer eine angemessene Menge an Instruktionen zur Verfügung zu stellen, mit denen er effizienter arbeiten kann.

Systemprogramme / Dienstprogramme

Das sind vom Betriebssystemhersteller mitgelieferte Programme wie z.B.: Window-Manager, Compiler für das Kompilieren von C-Programmen, Editoren, Systemwerkzeuge wie z.B.: Taskmanager, Explorer ...

Anwenderprogramme

Die oberste Schicht stellt den Benutzern des Systems Anwendersoftware zur Verfügung. Diese Programme setzen entweder auf den Dienstprogrammen oder unmittelbar auf dem Betriebssystem auf. Beispiele dafür sind:

- Browser
- Microsoft Office
- ....

2 Hauptaufgaben von Betriebssystemen

Übersicht

- **Abstraktion (Vereinfachung) der Hardware** –> abstrakte Programmierschnittstelle
- **Verwaltung von Systemressourcen** (CPU, Speicher, Netzwerk, Drucker, Platten..)
  - **Prozessverwaltung**
  - **Speicherverwaltung**
  - **Dateiverwaltung**

Der Rechner mit seinen Peripheriegeräten stellt eine Fülle von Ressourcen zu Verfügung, wie CPU, Hauptspeicher, Plattenspeicher, externe Geräte. Die Verwaltung dieser Ressourcen ist Aufgabe des Betriebssystems.

Der Aufruf eines Programms führt oft zu vielen gleichzeitig und unabhängig voneinander ablaufenden Teilprogrammen, auch **Prozesse** genannt.

Ein Prozess ist also ein eigenständiges Programm mit eigenem Speicherbereich, der vor dem Zugriff durch andere Prozesse geschützt ist. Allerdings ist es erlaubt, dass verschiedene Prozesse untereinander *kommunizieren* , also Daten untereinander austauschen.

Dabei hat das Betriebssystem die Aufgabe, die *Kommunikation* zwischen diesen Prozessen zu verwalten, damit sich die gleichzeitig aktiven Prozesse nicht untereinander beeinträchtigen oder gar zerstören. Das Betriebssystem verwaltet unter anderem also gleichzeitig aktive Prozesse, so dass einerseits keiner benachteiligt wird, andererseits aber kritische Prozesse mit Priorität behandelt werden.

Selbstverständlich können Prozesse aber nicht wirklich gleichzeitig ablaufen. Das Betriebssystem erzeugt aber eine scheinbare Parallelität dadurch, dass jeder Prozess immer wieder eine kurze Zeitspanne (wenige Millisekunden) an die Reihe kommt, dann unterbrochen wird, während andere Prozesse bedient werden. Nach kurzer Zeit ist der unterbrochene Prozess wieder an der Reihe und setzt seine Arbeit fort. Diesen Vorgang nennt man **Multitasking**.

Ähnlich verhält es sich mit der Verwaltung des Hauptspeichers, in dem nicht nur der Programmcode, sondern auch die Daten der vielen Prozesse gespeichert werden. Neuen Prozessen muss freier Hauptspeicher zugeteilt werden, der Speicher beendeter Prozesse muss wiederverwendet werden.

Die dritte wichtige Aufgabe ist die **Dateiverwaltung**. Damit ein Benutzer sich nicht darum kümmern muss, in welchen Sektoren auf welchen Spuren noch Platz ist um den gerade geschriebenen Text zu speichern, stellt das Betreibssystem das Konzept „Datei“ als Behälter für Daten aller Art zur Verfügung. Die Übersetzung von Dateien und ihren Namen in bestimmte Spuren, Sektoren und Köpfe der Festplatte nimmt das *Dateisystem* als Bestandteil des Betriebssystem vor.

Eigenschaften von Betriebssystemen

- Betriebssystem muss **änderbar** sein (durch neue Hardware, Software bzw. bei Sicherheitslücken)
- **modular** und **klar strukturiert** aufgebaut

- gut **dokumentiert** –> Schnittstellenbeschreibung!!

Ziele von Betriebssystemen

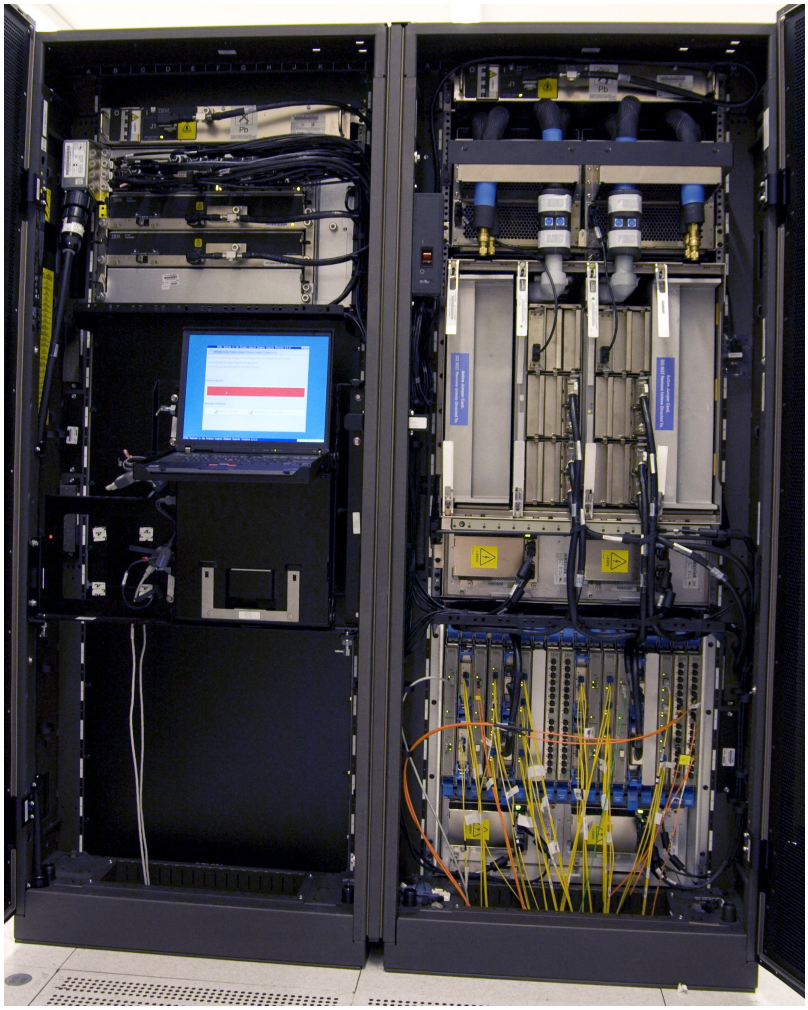
- **Bequemlichkeit** - aus Sicht des Benutzers
- **Effizienz** - Nutzung der Ressourcen
- **Entwicklungsfähigkeit** - Neue Dienste

Betriebssystemarten

Man unterscheidet mehrere Arten von Betriebssystemen, die im Laufe der Zeit entstanden sind.

Mainframe-Betriebssysteme

- Betriebssysteme für Großrechner
- Einsatz: Webserver, Datenbankserver, E-Commerce, Business-to-Business (B2B)
- Viele Prozesse gleichzeitig mit hohem Bedarf an schneller Eingabe/Ausgabe
- Beispiele sind MVS, OS/390, z/OS 2.1 –> zumeist IBM-Großrechner



Server-Betriebssysteme

- Betriebssysteme die auf Servern laufen
- Bieten verschiedene Dienste im Netzwerk an (Dateidienste, Webdienste,...)
- Dienste stehen vielen Benutzern im Netz zur Verfügung z.B.: Netzlaufwerk
- Aktuelle Beispiele sind Windows Server 2016 bzw. Red Hat Enterprise Linux 7.2

PC(Desktop)-Betriebssysteme

- Betriebssystem für Personalcomputer
- Meist nur 1 oder wenige Benutzer (über Netzwerk)
- Einsatz: Programmierung, Textverarbeitung, Spiele, Internetzugriff
- Mehrere Programme pro Benutzer → quasi-parallel
- Aufteilung der Prozesse auf vorhandene Kerne
- Zuteilung der Systemressourcen
- Beispiele: Linux, Windows, Mac OS X

Echtzeit-Betriebssysteme

- Einhaltung von harten Zeitbedingungen

- Einsatz: Steuerung von maschinellen Fertigungsanlagen, Steuerung von Ampeln, Robotorsteuerung...
- Aktion in einem fest vorgegebenen Zeitintervall

Eingebettete Systeme / Embedded Systems

- = Computer die man nicht unbedingt gleich sieht
- Einsatz: Fernseher, Handy, Auto,...
- Meist Echtzeitanforderungen
- Wenig Ressourcen zur Verfügung (geringer Stromverbrauch, wenig Arbeitsspeicher)
- Beispiele: iOS, Android, Windows Phone

Historische Entwicklung von Betriebssystemen

Arbeitsauftrag

Recherchiere im Internet hinsichtlich den historischen Entwicklungen von Betriebssystemen.

Beschreibe und erkläre die folgenden Entwicklungsstadien (= 4 Generationen) und gib die ungefähre Zeitspanne an:

1. Generation - Serielle Systeme
2. Generation - Einfache Stapelvearbeitungssysteme
3. Generation - ICS, Multiprogramming, Timesharing
4. Generation - Personal Computer

Das Ergebnis soll ein Word-Dokument (gegliedert in den 4 Generationen) im Umfang von 2-4 Seiten exklusive Titelblatt (Download unter

Titelblatt

- vor der Abgabe bitte den Namen im Titelblatt und den Dateinamen auf Betriebssysteme_NACHNAME.docx ändern) sein (d.h. mindestens 3 Seiten).

Letzter Abgabetermin ist der 20.09.2017 @ 23:59.

Die bis dahin abgegebenen Arbeitsaufträge werden bewertet und zählen zur Teilnote Praktische Arbeiten (PA)!!

Nicht abgegebene Arbeitsaufträge werden negativ beurteilt.

Abgabe unter

Google Classroom

Formatierung von Datenträgern

Stufen der Formatierung

Wenn man von Formatierung spricht, unterscheidet man dabei mehrere Stufen:

1. **Low-Level-Formatierung** = physikalische Einteilung eines Speichermediums
2. **Partitionierung** = logische Einteilung des Speichermediums in zusammenhängende Strukturen
3. **High-Level-Formatierung** = die logische Einteilung der Partitionsstruktur mit einem Dateisystem durch eine Software (meist durch das Betriebssystem).

Soweit durch die Beschaffenheit, industrielle Standards oder durch spezielle Verwendung die physikalische Einteilung des Mediums als Norm feststeht, ist eine separate Low-Level-Formatierung nicht erforderlich. In diesen Fällen können beide Vorgänge gleichzeitig vorgenommen werden, so zum Beispiel bei Disketten, CD-ROM, CD-RW oder DVD-ROM/RW.

1) Physikalische Formatierung (Low-Level-Formatierung)

Damit eine Festplatte logisch formatiert werden kann, muss sie physikalisch formatiert sein.

Die physikalische Formatierung eines Festplattenlaufwerks (auch als Zwei-Stufen-Formatierung bezeichnet) wird in der Regel vom Hersteller durchgeführt.

Durch die physikalische Formatierung wird eine Festplatte in ihre physikalischen Grundbausteine unterteilt: **Spuren**, **Sektoren** und **Zylinder**. Durch diese Elemente wird die Art und Weise vorgegeben, mit der Daten physikalisch auf der Festplatte aufgezeichnet und von dort gelesen werden können.

Spuren

Die **Spuren** sind konzentrische Kreispfade, die auf jede Scheibenseite geschrieben werden, wie auf einer Schallplatte oder einer CD. Die Spuren werden mit einer Nummer identifiziert, die mit Spur 0 am äußeren Rand einsetzt.

Zylinder

Der Spurensatz, der auf allen Seiten der Platte im gleichen Abstand von der Mitte angelegt ist, wird als "**Zylinder**" bezeichnet. Hardware und Software arbeiten häufig mit diesen Zylindern.

Sektoren

Die Spuren sind in Bereiche, sogenannte "**Sektoren**" oder „**Blöcke**" unterteilt, in denen eine

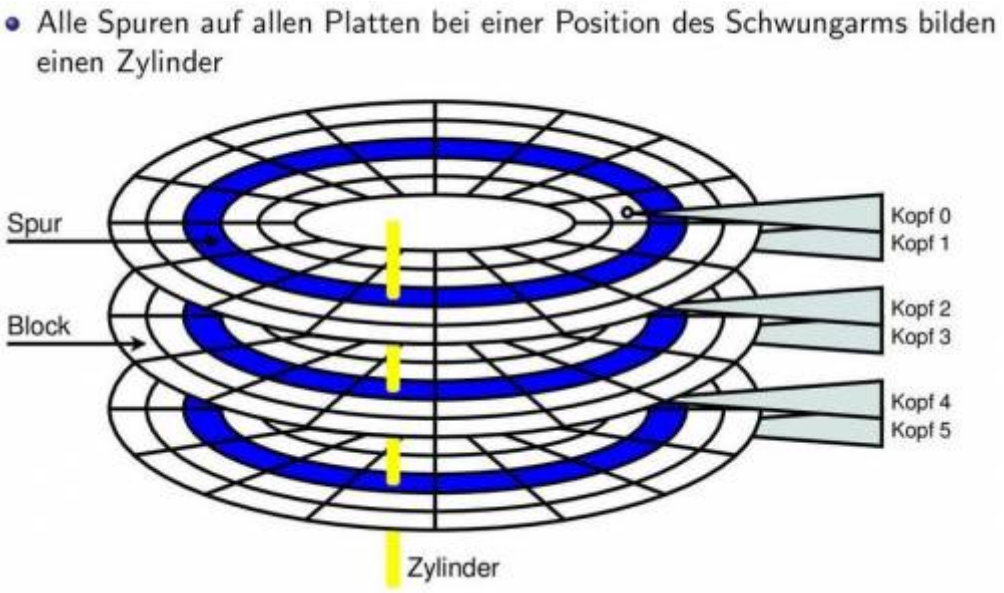
festgeschriebene Datenmenge gespeichert wird. Die Sektoren werden in der Regel für eine Speicherkapazität von 512 Byte (ein Byte besteht aus 8 Bit) formatiert.

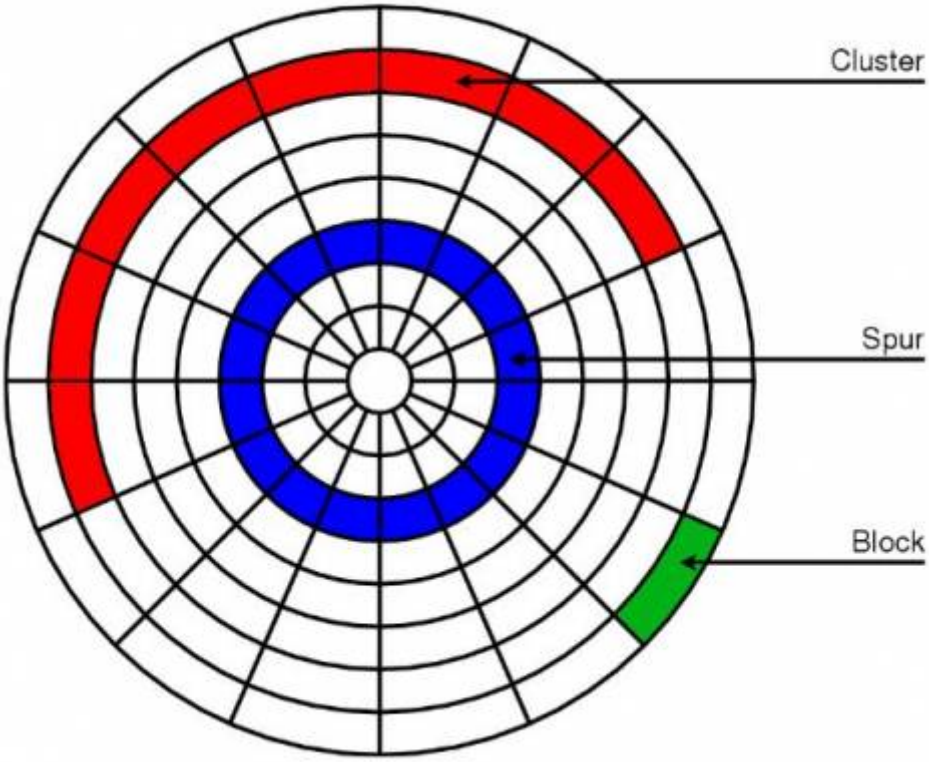
Fehlerhafte Sektoren

Nachdem ein Festplattenlaufwerk physikalisch formatiert wurde, kommt es gelegentlich vor, dass die magnetischen Eigenschaften der Eisenoxydschicht an einigen Stellen auf der Festplatte nach und nach an Wirksamkeit verlieren. Dies hat zur Folge, dass es für die Schreib-/Leseköpfe der Festplatte in zunehmenden Maße schwieriger wird, ein Bit-Muster auf der Festplatte zu speichern, das später von der Festplatte gelesen werden kann.

Wenn dieser Prozess einsetzt, bezeichnet man die Sektoren, in denen Daten nicht mehr zuverlässig gespeichert werden können, als “fehlerhafte Sektoren”. Glücklicherweise ist die Qualität moderner Festplatten so hoch, dass solche “fehlerhaften Sektoren” nur sehr selten auftreten.

Darüber hinaus sind die modernen Computer normalerweise in der Lage, einen fehlerhaften Sektor zu identifizieren und ihn als solchen zu kennzeichnen, damit er nicht mehr genutzt wird; es wird dann ein alternativer Sektor benutzt.





Adressierung der Daten auf einer (magnetischen) Festplatte

CHS - Adressierung

Bei Festplatten mit einer Kapazit< 8 GB werden die einzelnen Sektoren mit der sogenannten "**Zylinder-Kopf-Sektor-Adressierung(Cylinder-Head-Sector-Adressierung)**" durchgeführt.

Jeder Sektor lässt sich mit **CHS** klar lokalisieren und adressieren.

Einschränkungen durch das BIOS

- Zylinder = 10 Bits
- Kopf = 4 Bits –> später 8 Bits (erweitertes CHS)
- Sektor/Spur = 8 Bits

Maximaler Speicherplatz früher: 1.024 Zylinder * 16 Köpfe * 63 Sektoren/Spur * 512 Byte/Sektor = 528.482.304 Byte = 504 MB

Maximaler Speicherplatz heute: 1.024 Zylinder * 255 Köpfe * 63 Sektoren/Spur * 512 Byte/Sektor = 8.422.686.720 Byte = 7,844 GB

- Größe einer Spur berechnen?
- Größe einer Scheibe berechnen?

LBA- Adressierung

Seit der Einführung der logischen Blockadressierung (**Logical Block Addressing (LBA)**) 1995 werden alle Sektoren auf der Festplatte von 0 beginnend durchnummeriert. Durch LBA wurde das

BIOS - und CHS-Korsett umgangen und es kann damit jede derzeit gängige Festplatte voll adressiert werden. Die Zahlen zu Zylinder, Köpfen und Sektoren auf der Festplatte haben heute nichts mehr mit der tatächlichen Lage der Sektoren auf der Festplatte zu tun. Es handelt sich um eine logische Plattengeometrie, die nur aus Kompatibilitätsgründen existiert

2) Partitionierung von Festplatten

Eine Partition ist ein zusammenhängender Bereich einer Festplatte, der in der Regel ein Dateisystem enthält.
Wenn man einen PC oder ein Notebook mit vorinstalliertem Windows kaufen, enthält die Festplatte zumeist zwei Partitionen: eine winzige Partition mit Windows-Boot-Dateien und eine zweite Partition, die den Rest der Festplatte füllt und Windows enthält. Unter Umständen kann es auch weitere Partitionen geben, die beispielsweise ein Recovery-System enthalten.

Um mehrere Betriebssysteme (Windows, Linux etc.) auf einem Rechner zu installieren, benötigt man mehrere Partitionen. Für jedes Betriebssystem ist mindestens eine Partition erforderlich; für Linux sind sogar mehrere Partitionen sinnvoll.

Master-Boot-Record (MBR)

Der Master-Boot-Record ist der **erste Sektor** auf der Festplatte. Er beinhaltet zum einen den „**Bootstrap**“. Dies ist ein Programm, das von dem BIOS aufgerufen wird, um das eigentliche Betriebssystem zu laden. Zum anderen enthält dieser Sektor auch eine Beschreibung, ob / wie die Festplatte in unterschiedliche Bereiche (Partitionen) unterteilt ist. Diese Beschreibung erfolgt in der sogenannten „Partitionstabelle“. Sie enthält für jede Partition einen Eintrag. Dieser besteht aus der Lage der Partition auf der Festplatte und dem „Typ“ dieser Partition.
Die Master-Boot-Record Partitionstabelle ist eine Tabelle, welche die Unterteilung der Festplatte in Partitionen beschreibt. Sie ist seit der Verbreitung von Festplatten im Jahr 1980 fest definiert und wird von allen Betriebssystemen zwingend vorausgesetzt.

Aus historischen Gründen kann diese Partitionstabelle nur vier Einträge (primäre Partitionen oder erweiterte Partitionen) aufnehmen. Das Format dieses Master-Boot-Records (Bootstrap / Partitionstabelle) ist fest definiert und wird von allen Betriebssystemen zwingend vorausgesetzt.

Partitionsarten

Es gibt im wesentlichen zwei Partitionsarten: **Primärpartitionen** und **erweiterte Partitionen**. Darüber hinaus lassen sich erweiterte Partitionen noch weiter in **logische Partitionen** unterteilen. Sie können bis zu vier Hauptpartitionen auf dem Festplattenlaufwerk definieren, wovon eine als erweiterte Partition definiert werden kann, d.h. maximal vier Primärpartitionen oder drei Primärpartitionen und eine erweiterte Partition.

a) Primärpartitionen

In einer Primärpartition können Sie jedes beliebige Betriebssystem sowie Datendateien wie z.B. Programm- und Benutzerdateien speichern. Eine Primärpartition ist logisch formatiert, damit sich ein Dateisystem darauf definieren lässt, das mit dem auf der Partition installierten Betriebssystem kompatibel ist.

Wenn Sie mehrere Betriebssysteme auf Ihrem Festplattenlaufwerk installieren müssen, ist es in der Regel notwendig, mehrere Primärpartitionen zu erstellen, da die meisten Betriebssysteme (vor allem Windows) nur von einer Primärpartition gebootet werden können. Pro Festplatte ist es möglich 4 primäre Partitionen einzurichten, ohne den Bootsektor der Festplatte anzupassen.

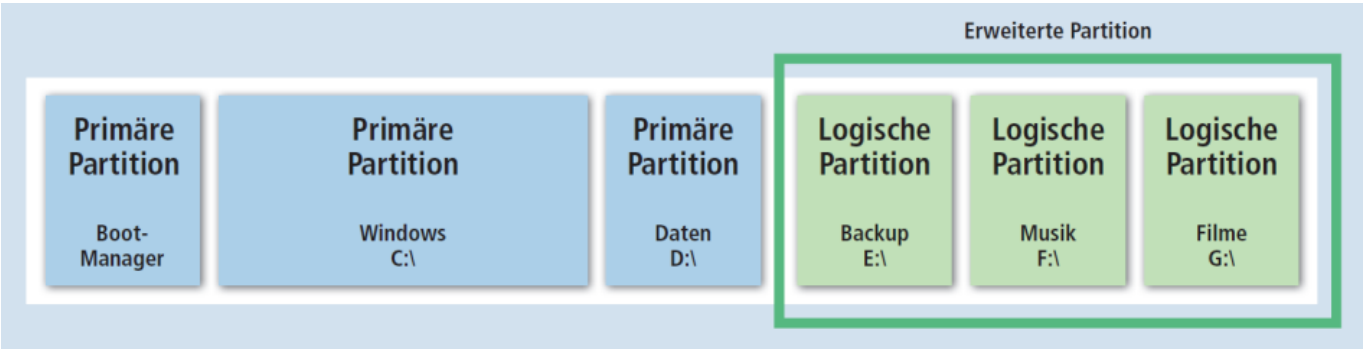
b) Erweiterte Partitionen

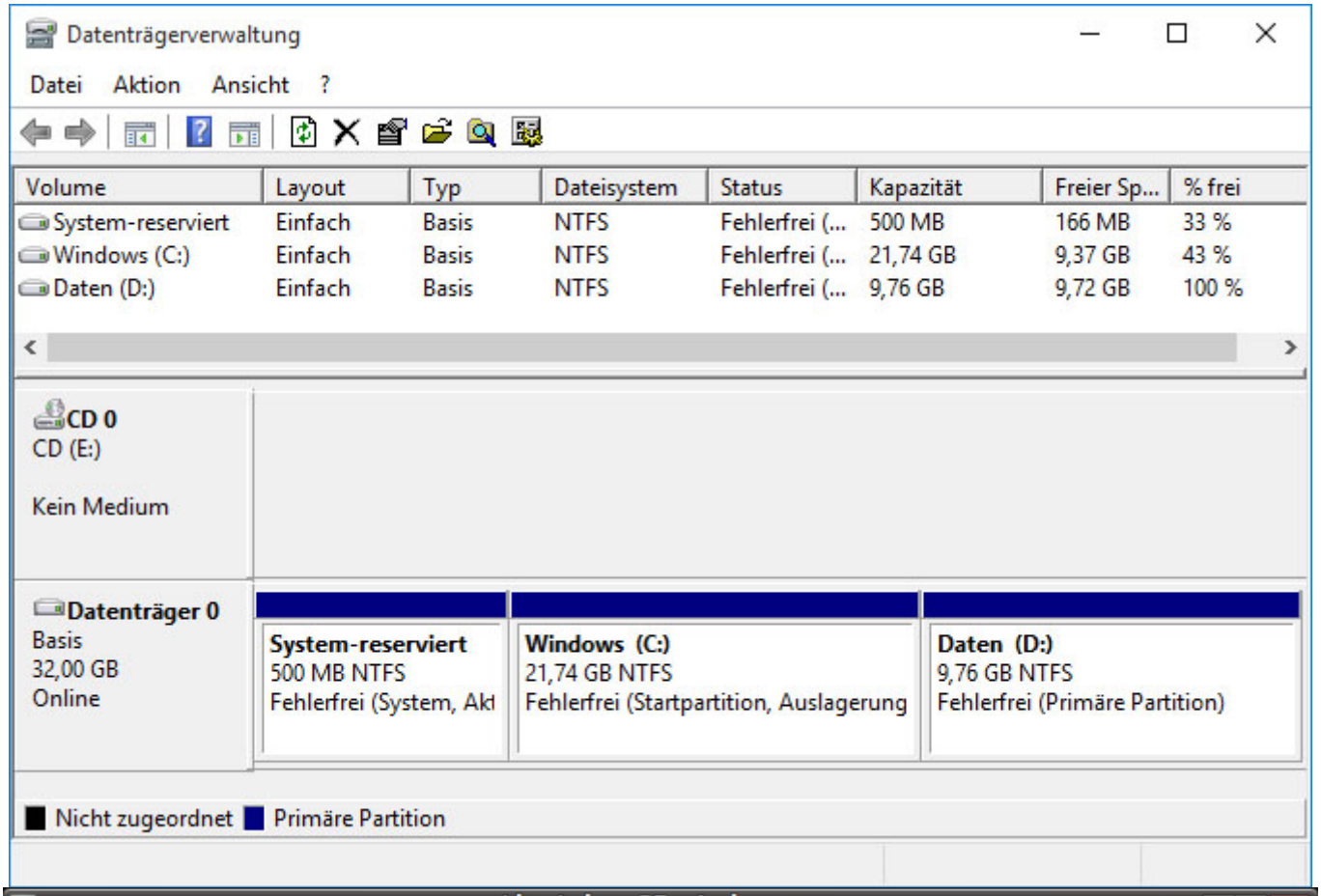
Die erweiterte Partition wurde entwickelt, um die willkürliche 4-Partitionen-Begrenzung zu umgehen. Es handelt sich im Prinzip um einen Bereich, in dem Sie den Datenträgerspeicher weiter physikalisch unterteilen können, indem Sie eine unbegrenzte Anzahl logischer Partitionen definieren (weitere physikalische Unterteilungen des Datenträgerspeichers).

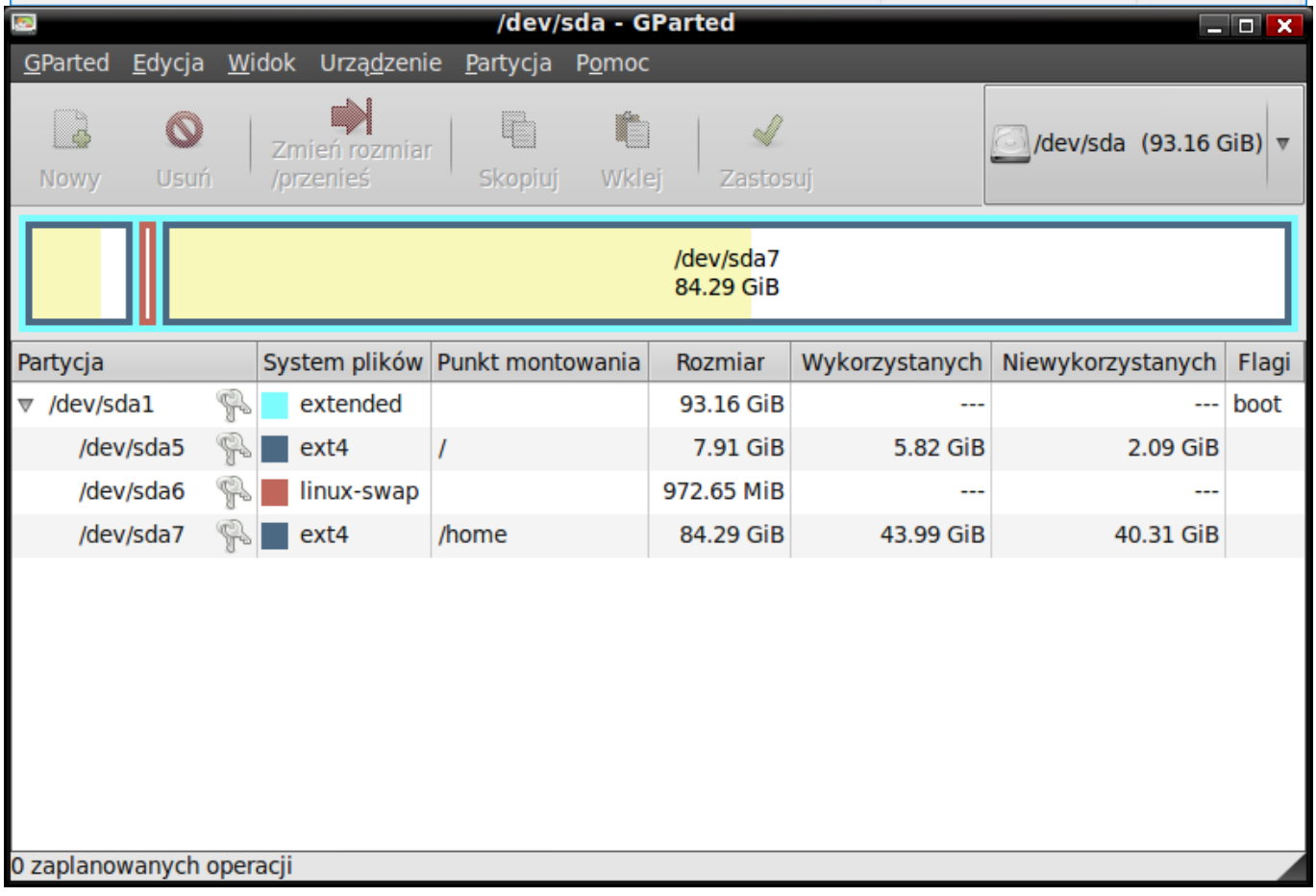
Eine erweiterte Partition dient nur indirekt der Datenspeicherung. Der Benutzer muss in der erweiterten Partition logische Partitionen definieren, in denen die Daten gespeichert werden. Logische Partitionen müssen logisch formatiert sein; auf jeder logischen Partition kann ein anderes Dateisystem definiert sein. Nach der logischen Formatierung gilt jede logische Partition als separater Datenträger.

c) Logische Partitionen

Logische Partitionen können nur innerhalb einer erweiterten Partition definiert werden und sind nur für die Speicherung von Datendateien und Betriebssystemen vorgesehen, die von einer logischen Partition gebootet werden können (z.B. Linux und Windows NT). Betriebssysteme, die von einer logischen Partition gebootet werden können, sollten gewöhnlich in einer logischen Partition installiert werden; damit können Primärpartitionen für andere Zwecke freigehalten werden.







Frage: Wie finde ich die aktuelle Partitionierung von meinem PC?

Gründe für Partitionen

Das Problem früher waren die Festplatten-Controller die nicht in der Lage waren, einen größeren Adressbereich anzusprechen. Und, der technische Fortschritt und die höheren Kapazitäten von Festplatten wurden schneller eingeführt als neue und bessere Dateisysteme. Vor allem unter Windows-Betriebssystemen war das FAT-Dateisystem (File Allocation Table) lange führend. FAT ermöglichte durch die Zusammenführung mehrerer Blöcke zu einer logischen Ansprecheinheit (Cluster), um die Adressierungsbeschränkung zu umgehen. Es hatte den Nachteil, dass es Festplatten nur bis zu einer bestimmten Kapazität verwalten und die Dateien nicht besonders platzsparend speichern konnte.

Beschränkungen der Partitionsgröße vom Dateisystem:

- FAT16 ⇒ max. Partitionsgröße = 2 GByte
- Nachfolger FAT32 ⇒ 2 TByte
- Über 32 GByte verwendet man üblicherweise das Dateisystem NTFS.

Heutige Gründe für Partitionierungen:

- Installation mehrerer Betriebssysteme
- Einrichten verschiedener Dateisysteme
- Reservierung für eine Windows- oder Linux-Auslagerungsdatei (SWAP-Partition)
- Installationsdateien
- sehr großen Speicher verwalten
- Trennung von Programmen und Daten

3) Logische Formatierung (High-Level-Formatierung)

Ein physikalisch formatiertes Festplattenlaufwerk muss noch logisch formatiert werden. Durch die logische Formatierung wird ein **Dateisystem** auf der Festplatte definiert. Erst ein Dateisystem gestattet es einem Betriebssystem wie z.B. DOS, OS/2, Windows 95/98, Windows NT/Windows 2000/WindowsXP/Windows7/Windows8 oder Linux), den verfügbaren Speicher für das Speichern und Laden von Dateien zu nutzen.

Vor der logischen Formatierung kann eine Festplatte in **Partitionen** unterteilt werden. Auf jeder Partition kann ein Dateisystem (logisches Format) definiert werden. Eine Festplattenpartition, die bereits logisch formatiert ist, wird als Datenträger (Volume) bezeichnet. Als Teil des Formatierungsvorgangs werden Sie durch das Formatierungsprogramm aufgefordert, der Partition einen Namen zuzuweisen, die sogenannte “Datenträgerbezeichnung”. Mit diesem Namen können Sie den Datenträger (die Partition) später identifizieren.

Normal- und Schnellformatierung

Es gibt zwei unterschiedliche Methoden, den Datenträger high-level zu formatieren:

Die Schnell- und die Normalformatierung.

Sie unterscheiden sich in folgenden Punkten:

Normalformatierung – Es wird eine Suche nach fehlerhaften Sektoren durchgeführt. Diese Suche nimmt den Großteil der Zeit in Anspruch. Anschließend erfolgt das Schreiben der Dateisystem-Metadaten und somit die Löschung der vorhandenen Dateien.

Schnellformatierung – Es werden zwar die Dateien aus dem Inhaltsverzeichnis des Datenträgers beziehungsweise der betreffenden Partition entfernt, die Suche nach fehlerhaften Sektoren wird jedoch ausgelassen.

Bei einer High-Level-Formatierung sind in dem neuen Dateisystem die alten Daten nicht verfügbar, da sie nicht mehr durch entsprechende Verweise im Dateisystem referenziert werden. Sie werden jedoch nicht notwendigerweise gelöscht. Meistens verbleiben sie rein physikalisch auf der Festplatte, bis sie mit neuen Daten überschrieben werden. Solange die verwendeten Datenblöcke nicht erneut beschrieben werden, ist mit entsprechender Software eine weitgehende Wiederherstellung noch möglich, gestaltet sich jedoch schwerer, als wenn die Dateien einfach nur gelöscht würden. Nicht selten können daher via Software nur die einzelnen Dateien, nicht aber die Ordnerstruktur wiederhergestellt werden.

Dateisysteme

Jeder Computer benötigt ein funktionsfähiges Betriebssystem, auch wenn es lediglich aus den DOS- oder Windows 98-Bootdateien besteht. Verschiedene Betriebssysteme verwenden unterschiedliche Dateisysteme, um auf Datenträger zuzugreifen und die Speicherorte von Daten zu ermitteln. Das Dateisystem bestimmt hierbei die Art der Verwaltung und des Zugriffs auf Dateien auf einem Datenträger.

Die Methoden und Datenstrukturen, die ein Betriebssystem verwendet, um Speicherorte von Dateien auf einer Partition zu ermitteln, werden also als das **Dateisystem** definiert. Bei den hier besprochenen Dateisystemen handelt es sich um FAT, FAT32, HPFS, NTFS, ReFS, Linux Ext2 und Linux ReiserFS.

Windows

FAT16

Das FAT16-Dateisystem ist das ursprünglich von MS-DOS für die Dateiverwaltung verwendete System. Das FAT (File Allocation Table) ist eine Datenstruktur, die MS-DOS beim Formatieren eines Datenträgers auf demselben erstellt. FAT16-Partitionen können bis zu 2 GB groß sein. Die Größe einer Partition bestimmt die Größe der Cluster, in denen Daten gespeichert werden. Bei einer Partitionsgröße von über einem GB kann dies zu einem erheblichen Speicherverlust führen. FAT16-Partitionen werden bei den meisten Versionen von DOS und Windows 95 eingesetzt und von den folgenden Betriebssystemen unterstützt: Windows 95, Windows 98, Windows NT, Windows 2000, DOS (inklusive MS-DOS, PC-DOS und DR-DOS), OS/2, Linux und viele andere Betriebssysteme.

FAT32

FAT32 ist ein Dateisystem, das von Microsoft für Windows 95, Version B (OSR2), entwickelt wurde. Es verfügt über die Fähigkeit, 32-Bit-Clusteradressen zu verwalten und unterstützt Partitionen von bis zu 8 GB bei einer Clustergröße von nur 4 KB. Partitionen von bis zu 16 GB können bei einer Clustergröße von 8 KB verwaltet werden, was den unnötigen Verbrauch von Festplattenspeicher aufgrund ineffizienter Clusternutzung deutlich reduzieren kann. FAT32-Partitionen kommen ausschließlich unter Windows 95B oder höher, Windows 98, Windows 2000 und unter neueren Versionen von Linux zum Einsatz.

NTFS

NTFS (New Technology File System) wurde von Microsoft speziell für die Verwendung mit Windows NT, Windows 2000 und Windows XP entwickelt und unterstützt die verbesserten Sicherheitsmerkmale, die über dieses Betriebssystem zur Verfügung stehen. NTFS unterstützt vollständige Zugriffskontrolle, Dateisystemwiederherstellung und besonders große Datenträger. NTFS-Partitionen werden ausschließlich unter Windows NT (Workstation und Server) sowie unter Windows 2000, Windows XP, Windows 7 und Windows 8 eingesetzt.

ReFS

ReFS (Resilient File System) ist ein Dateisystem von Microsoft, wobei engl. „resilient“ für „elastisch“, „belastbar“ oder gar „unverwüstlich“ steht. Es wurde mit den Betriebssystemen Windows 8 und Windows Server 2012 als Ergänzung zum NTFS-Dateisystem eingeführt.

Linux

Ext2

Das Ext2-Dateisystem war bis ca. 2002 das Standardsystem für Linux. Ist außerordentlich stabil und sicher, aber leider muss nach einem Systemabsturz (z.B. Stromausfall) eine sehr zeitaufwändige Überprüfung des gesamten Dateisystems durchgeführt werden.

Ext3

Das Ext3-Dateisystem ist z.Z. das Standardsystem für Linux. Weitgehend kompatibel zu Ext2, hat aber den Vorteil das es über eine **Journaling-Funktion** verfügt. D.h. nach einem Stromausfall entfällt eine langwierige Überprüfung des gesamten Dateisystems.

Ext4

Ext4 (Fourth Extended File System): weiterentwickelte Variante von Ext3, unter anderem mit

erweiterten Limits.

ReiserFS

Das ReiserFS ist eine Weiterentwicklung des Ext2-Dateisystem ist das Standardsystem für Linux (Verbesserte Dateisystemwiederherstellung nach Unterbrechungen im laufenden Betrieb). Linux ReiserFS-Partitionen werden ausschließlich unter Linux-Installationen verwendet.

MAC

HFS

- HFS

Weiterführende Informationen

Festplatten Fakten

Planung eines Systems

Man sollte sich vor der Installation der Betriebssysteme (oder eines Bootmanagers) einige Gedanken über den Aufbau des Systems machen: Hierbei ist am wichtigsten, sich zuerst überlegen, welche Btriebssysteme eingesetzt werden sollen.
Als nächstes ist abzuklären, wie viel Festplattenplatz für welches Betriebssystem benötigt wird(Größe der Festplatte). Hierzu macht der Hersteller des Betriebssystem bereits einen Vorschlag, der als Minimum eingehalten werde sollte.
Unter Umständen kann es sinnvoll sein, einen Teil der Festplatte (eine Partition) ausschließlich dazu zu verwenden, Daten unter allen, oder zumindest unter mehreren Betriebssystemen, zur Verfügung zu stellen. Am besten lässt man einen Teil der Festplatte unbenutzt (falls diese groß genug ist), damit dieser später für Erweiterungen genutzt werden kann.

Besonderheiten und Probleme der einzelnen Betriebssysteme

Probleme mit DOS / Windows 95/98/ME

Bei der Verwendung von FAT 16 durfte die Partition nicht größer als 2 GB sein.
Von der 2. Festplatte kann nur gebootet werden, wenn auf der 1. Festplatte keine primäre Partition sichtbar ist.

Probleme mit Windows NT/2000/XP

- Windows 2000/XP: Das Setup-Programm von Windows 2000/XP überschreibt während der

Installation den Master-Boot-Record.

- Unter Windows 2000/XP kann eine einmal angelegte Partition nicht mehr verkleinert werden.

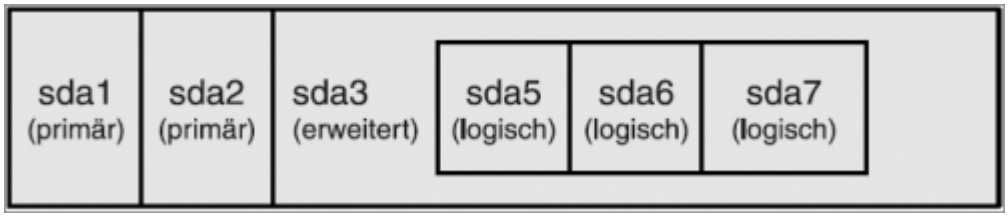
Windows 7, Windows 8

- ab Windows 7 können Partitionen auch verkleinert werden:
    - Datenträgerverwaltung - Rechtsklick auf die Partition - Volume verkleinern

Linux

Linux kann auf jeder Festplatte maximal 15 Partitionen ansprechen, davon maximal 11 logische Partitionen.
Unter Linux erfolgt der interne Zugriff auf Festplatten bzw. deren Partitionen über so genannte Device-Dateien: Die Festplatten erhalten der Reihe nach die Bezeichnungen /dev/sda, /dev/sdb, /dev/sdc etc.

Um eine einzelne Partition und nicht die ganze Festplatte anzusprechen, wird der Name um die Partitionsnummer ergänzt. Die Zahlen 1 bis 4 sind für primäre und erweiterte Partitionen reserviert. Logische Partitionen beginnen mit der Nummer 5 – auch dann, wenn es weniger als vier primäre oder erweiterte Partitionen gibt. Die folgende Abbildung veranschaulicht die Nummerierung: Auf der Festplatte gibt es zwei primäre Partitionen und eine erweiterte Partition, die drei logische Partitionen enthält.



Die maximale Partitionsgröße beträgt 2 TByte. Da es mittlerweile Festplatten mit mehr als 2 TByte Speichervolumen gibt, ist eine sinnvolle Nutzung von Festplatten mit mehr als 2 TByte nur noch mit GPT-Partitionstabellen möglich.

Anzahl und Größe der Linux-Partitionen

Leider gibt es auf die Frage, wie viele Linux-Partitionen ein System haben sollte, keine allgemein gültige Antwort.

Die **Systempartition** ist die einzige Partition, die man unbedingt benötigt. Sie nimmt das Linux-System mit all seinen Programmen auf. Diese Partition bekommt immer den Namen /. Dabei handelt es sich genau genommen um den Punkt, an dem die Partition in das Dateisystem eingebunden wird (den mount-Punkt).
Eine vernünftige Größe für die Installation und den Betrieb einer gängigen Distribution liegt bei 10 bis 20 GByte.

Mit einer **Datenpartition** trennt man den Speicherort für die Systemdateien und für die eigenen Dateien. Das hat einen wesentlichen Vorteil: Man kann später problemlos eine neue Distribution in die Systempartition installieren, ohne die davon getrennte Datenpartition mit Ihren eigenen Daten zu gefährden. Bei der Datenpartition wird **/home** als Name bzw. mount-Punkt verwendet, weswegen oft auch von einer Home-Partition die Rede ist. Die Größe hängt von den eigenen Bedürfnissen ab.

Die **Swap-Partition** ist das Gegenstück zur Auslagerungsdatei von Windows: Wenn Linux zu wenig RAM hat, lagert es Teile des gerade nicht benötigten RAM-Inhalts dorthin aus. Im Gegensatz zu den anderen Partitionen bekommt die Swap-Partition keinen Namen (keinen mount-Punkt). Die Größe sollte das ein- bis zweifache des RAMs betragen.

Mehrere Betriebssysteme verwalten

Es gibt verschiedene Möglichkeiten, mehrere Betriebssysteme zu verwalten:

- Mit Hilfe von Bootmanagementprogrammen, wie beispielsweise BootMagic von PowerQuest, Bootstar von Star-Tools, Acronis von SWSoft oder GRUB von SuSE-Linux
- Mit Hilfe einer von einem Betriebssystem verwalteten doppelten Bootkonfiguration, wie

beispielsweise der Boot Loader von Windows NT oder GRUB (bzw. mit dem älteren) LILO unter Linux.

- Manuell, indem Sie ein Betriebssystem als „aktiv“ einstellen. Nehmen Sie diese Einstellung über ein Dienstprogramm wie PQBoot vor, oder bearbeiten Sie den Masterbootdatensatz von Hand (nicht zu empfehlen!).

Windows-Betriebssysteme

- MS-DOS / DOS
- Windows 3.11
- Windows 95
- Windows 98
- Windows NT
- Windows 2000
- Windows XP
- Windows Vista
- Windows 7
- Windows 8
- Windows 10

Linux-Betriebssysteme

- OpenSuSE Linux
  - Open Suse für Profis
- SuSE Linux

Ubuntu

- Ubuntu
- www.ubuntu.com
  - Ubuntu Austria
  - Ubuntu Server
- Kubuntu
- Edubuntu
- Xubuntu
- Gobuntu
- Knoppix
- RedHat Linux
- Debian Linux
- Xubuntu von USB-Stick: Pendrive Linux

Macintosh-Betriebssysteme

- Leopard
- Snow Leopard

Betriebssystem-Virtualisierung

- Virtualisierung

Online-Betriebssysteme

- Ein Online-Desktop von eyeOS

Portable-Betriebssysteme

- PC am Datenstick
- Linux Advanced

Windows

- MS-DOS
- Benutzerverwaltung
- Dateimanagement
- Sicherung & Wiederherstellung

Das Betriebssystem MSDOS

MSDOS (DOS = Disk Operating System) ist ein Betriebssystem, das von der Firma Microsoft in den USA entwickelt wurde und früher zu den am meisten verbreiteten Betriebssystemen zählte. Die Entwicklungsgeschichte reicht sehr lange zurück, die erste Version gab es 1981.

Mittlerweile ist MSDOS weitgehend duch die neuen Windows-Betriebssysteme abgelöst worden - trotzdem wird DOS für bestimmte Aufgaben auch heute noch verwendet, wie z.B. zur Erstellung von Batch-Dateien. Die Kenntnisse über DOS könnten auch insbesondere bei einem Crash nützlich sein, wenn Windows z.B. nicht mehr startet oder man ein Uralt-DOS-PC-Spiel noch mal spielen will.

Arbeiten mit DOS auf der Systemkonsole

Speicher zuvor folgenden Ordner auf deinem Verzeichnis: dos-befehle.zip

Start der Systemkonsole

- Start - Ausführen: cmd
- Start - Programme - Zubehör - Eingabeaufforderung

- Vollbildmodus: ALT + ENTER
- Beenden: exit

Einige wichtige DOS-Befehle

Allgemeine Befehle

Befehl	**Beschreibung**
help	zeigt eine Liste aller MS-DOS-Befehle und eine kurze Beschreibung
help *Befehl*	man erhält Informationen zu einem bestimmten Befehl
dir	zeigt Inhaltsverzeichnis des aktuellen Verzeichnisses an
dir i:	zeigt den Inhalt des Laufwerks i
dir *.exe	Stern ist eine „Wild Card“. Er steht für eine beliebige Zeichenkombination, es werden somit alle .EXE-Dateien ausgegeben
dir m??er.doc	Fragezeichen ist ebenfalls eine „Wild Card“, jedoch für ein einzelnes Zeichen
whoami	Gibt den Usernamen des ausführenden Benutzers aus
hostname	Gibt den Rechnernamen aus

Befehle für Verzeichnisse

Die folgende Befehle sind zum Anlegen, Wechseln und Löschen neuer Verzeichnisse. Das Verzeichnis, welches im aktuellen Eingabebereitschaftszeichen aufscheint, wird meist als **aktuelles Verzeichnis** oder **Arbeitsverzeichnis** bezeichnet.

Befehl	Beschreibung
cd	change directory, Bedeutung von Syntax abhängig
cd \..	geht zum root (oberste Verzeichnisebene)
cd \programme	wechselt in das Verzeichnis Programme ausgehend vom root
cd programme	wechselt - ausgehend vom derzeitigen Arbeitsverzeichnis - in das Unterverzeichnis `programme`
cd .	Zeiger auf aktuelles Verzeichnis
cd ..	Zeiger auf übergeordnetes Verzeichnis
md	make directory - **erzeugt Unterverzeichnis**
rd	remove directory - **löscht Unterverzeichnis**
tree	zeigt die Verzeichnisstruktur auf dem aktuellen Laufwerk an

Befehle für Dateien

Wichtiger Operationen mit Dateien sind das Kopieren, das Löschen und das Umbenennen.

Befehl	Beschreibung
copy	benötigt zwei Parameter: 1. Quelldatei, die kopiert werden soll, 2. Zieldatei
copy x.dat x.old	kopiert die Datei x.dat auf x.old (dabei wird x.dat nicht gelöscht!). Existiert bereits eine Datei x.old, so wird diese gelöscht und mit den Daten von x.dat gefüllt.
copy *.* a:	kopiert alle Dateien im Arbeitsverzeichnis nach A:
copy adam.dat a:	kopiert adam.dat **nach** a:adam.dat
copy *.* c:\xy\z	kopiert alle Dateien des aktuellen Ordners auf die Festplatte c: ins Unterverzeichnis Z des Verzeichnisses xy
del x.dat	löscht die Datei `x.dat` im aktuellen Verzeichnis
del *.*	löscht alle Dateien im aktuellen Verzeichnis
ren x.dat y.dat	rename, benennt x.dat in y.dat um
move muster.doc windows	verschiebt `muster.doc` ind das Verzeichnis `windows`
type *Dateiname*	zeigt den Inhalt der angegebenen Datei am Bildschirm an
edit *Dateiname*	ruft einen Editor auf, um den Inhalt einer Datei verändern zu können
print *Dateiname*	gibt den Inhalt einer Datei auf dem Drucker aus
xcopy /E	Kopiert ganze Verzeichnisse
rmdir /S	Löscht Verzeichnisse trotz Inhalt

DOS-Übung

Speichere den Ordner „DOS“ dos.rar in dein Verzeichnis, bevor du mit der Übung beginnst!

1. Erzeuge ein Verzeichnis `uebungen` in deinem homedirectory.
2. Wechsle in dieses Verzeichnis.
3. Erstelle eine Datei `test1.txt` und `test2.txt` mit beliebigen Inhalt.
4. Kopiere die Datei `test1.txt` in die Datei `test21.txt`.
5. Erstelle ein Verzeichnis `aufgabe`.
6. Benenne `test2.txt` um in `test12.txt`.
7. Kopiere die Dateien test1?.txt in das Verzeichnis `aufgabe`. Welche Dateien wurden kopiert?
8. Kopiere dir alle Dateien aus dem Ordner`DOS` in das Verzeichnis `aufgabe`. Wie viele Dateien

befinden sich nun in diesem Verzeichnsi?
9. Lasse dir alle Dateien mit der Endung `*.txt` ausgeben.
10. Lösche alle Dateien mit der Endung `*.dat`.
11. Benenne alle Dateien mit der Endung `*.bak` um in `*.txt`.
12. Lasse dir alle Dateien sortiert nach der Dateigröße anzeigen. Finde den Befehl selber heraus.
13. Gib den Inhalt der Datei `hallo.txt` aus.
14. Lasse dir alle Dateien anzeigen, bei denen im Dateinamen die Zahl 1 vorkommt.
15. Lösche diese Dateien.
16. Verschiebe alle Dateien aus dem Verzeichnis Aufgabe in das Verzeichnis uebungen.
17. Lösche das Verzeichnis Aufgabe.
18. Gib den Befehl `attrib +h *.*` ein.
19. Lasse dir alle Dateien anzeigen. Was ist passiert?
20. Mache dir über den Befehl `attrib` schlau und gib einen Befehl ein, sodass alle Dateien wieder angezeigt werden.

BATCH (.bat) Files

Befehl	**Beschreibung**
echo	Gibt Text aus
@echo off	Schaltet die Autobefehlsanzeige aus
title	Definiert den Titel des BATCH-Files
set VAR=Text	Speichert Text in die Variable VAR
set /P VAR=	Fordert den User auf eine Zeichenfolge einzugeben
set/A C=%A%+%B%	/A gibt an, dass die Zeichenfolge rechts vom = ein numerischer Ausdruck ist, der ausgewertet wird. in C wird das Ergebnis von A+B gespeichert
echo %VAR%	Ausgabe einer Variable → Text
echo %USERNAME%	Gibt den Usernamen des ausführenden Benutzers aus
REM	Definiert einen Kommentar

Benutzerverwaltung

Microsoft Windows ist ein Multi-User Betriebssystem. Was ist ein Benutzer? Was ist eine Gruppe? Welche Rolle spielt die Sicherheitsrichtlinie?

BENUTZER

SID

Jeder Benutzer wird durch eine Nummer, einer SID (Security Identifier) eindeutig gekennzeichnet und erhält ein eigenes Profil unter c:\users\%username%.

Mithilfe von SIDs ist es u.a. möglich Benutzer umzubenennen, ohne Veränderung seiner Zugriffsrechte. Selbst wenn das Administratorkonto umbenannt wird, so hat dieser immer die SID mit der Endung 500. Welcher User welche SID verwendet kann in der CMD oder mit PowerShell eingesehen werden:

Befehl für die PowerShell:

```
PS C:\> get-wmiobject win32_useraccount


AccountType : 512
Caption     : SERVER01\Administrator
Domain      : SERVER01
SID         : S-1-5-21-140281148-68646805-4244902722-500
FullName    :
Name        : Administrator

AccountType : 512
Caption     : SERVER01\Gast
Domain      : SERVER01
SID         : S-1-5-21-140281148-68646805-4244902722-501
FullName    :
Name        : Gast
```

Befehl für die CMD:

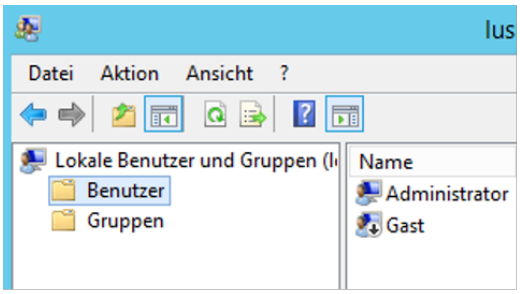
```
wmic useraccount get name,sid
```

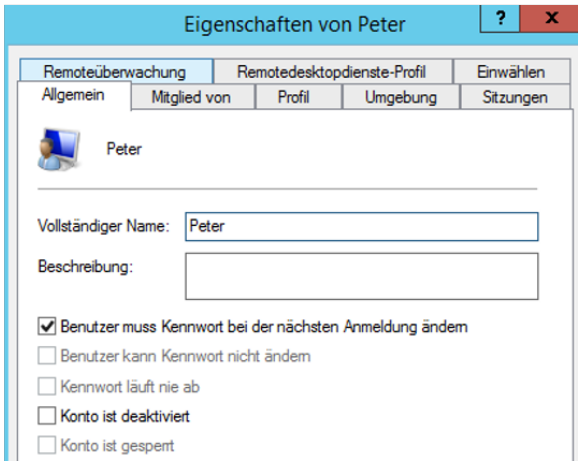
Kennwörter/Passwörter

Kennwörter werden in der SAM (Security Account Manager) Datenbank gespeichert. Diese Datenbank kann im laufenden Betrieb (mit Bordmitteln) nicht eingesehen werden. Die Kennwörter werden als Hashwert gespeichert, d.h. nicht im Klartext.

Anlegen von Lokalen Benutzern

Dies ist grafisch mit lusrmgr.msc oder mithilfe des Befehls net user möglich.

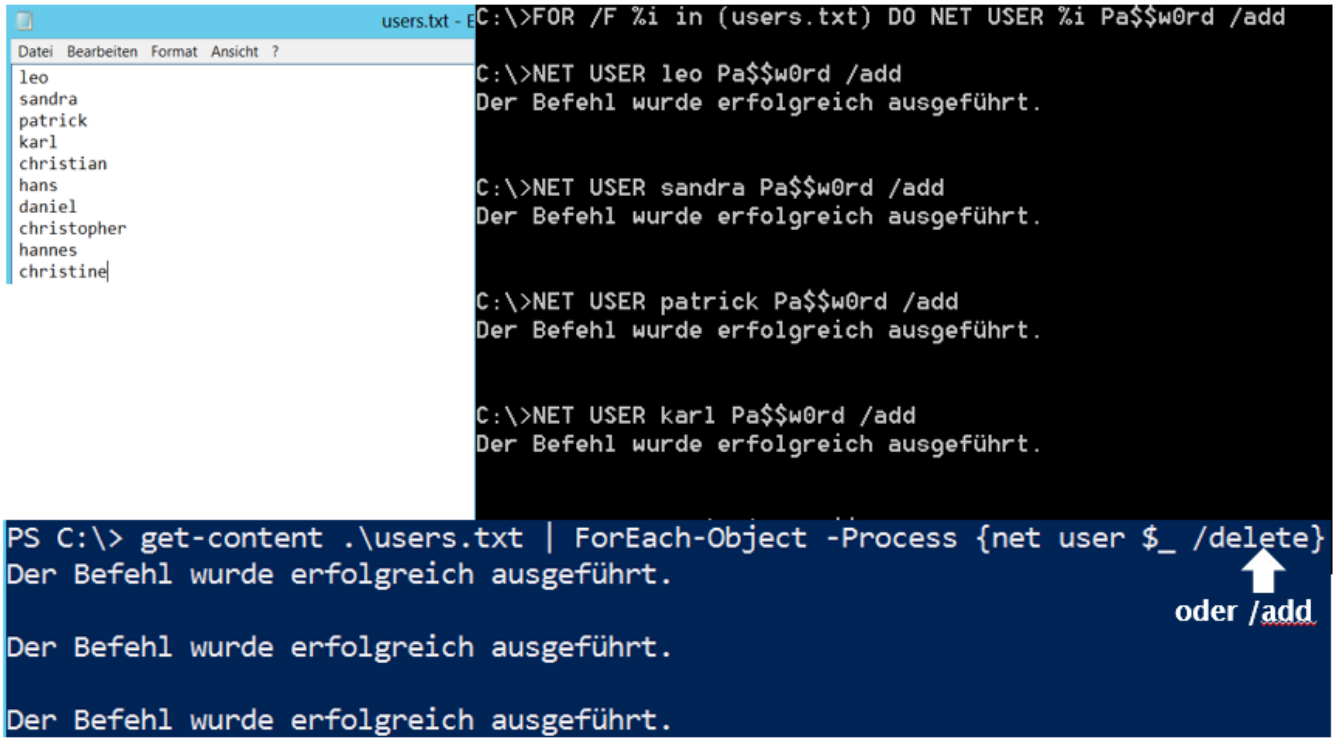





```
net user Max 'Pa$$w0rd' /add
```

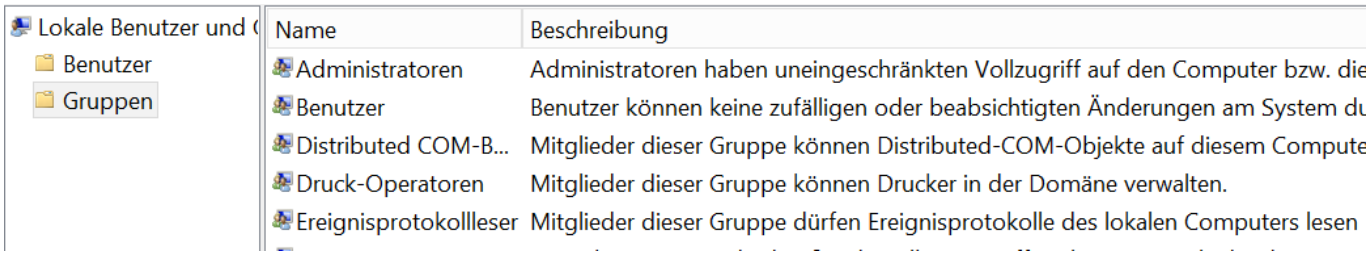
Massenimport von Usern

Hier am Beispiel einer Textdatei users.txt mit einer Auflistung von Vornamen:



GRUPPEN

Bei einer Gruppe handelt es sich um eine Sammlung von Benutzer- und Computerkonten, Kontakten sowie weiteren Gruppen, die als einzelne Einheit verwaltet werden können. Gruppen findet man ebenfalls in lusrmgr.msc bzw. auch mit net localgroup.

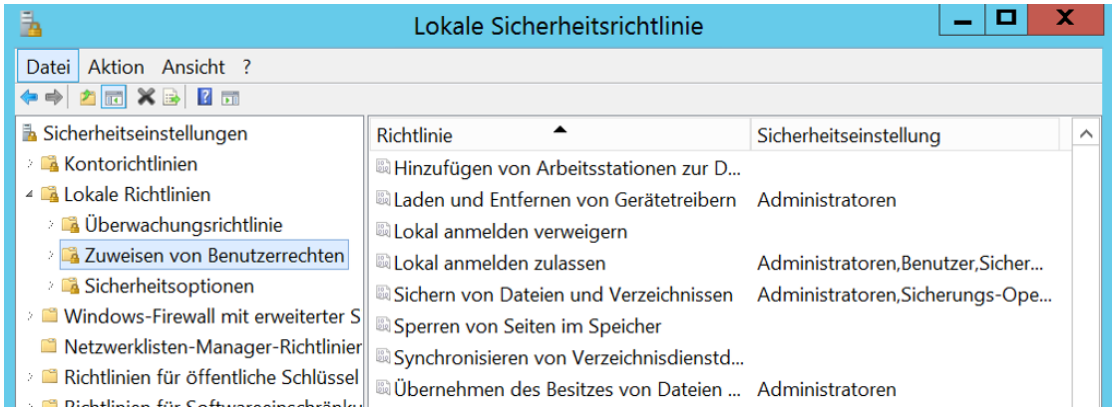


Hier wird der Benutzer admin01 der Gruppe Administratoren hinzugefügt:

```
net localgroup administratoren admin01 /add
```

SICHERHEITSRICHTLINIEN

Wozu das Ganze? Gruppen, Benutzer? Nun, es ist wichtig zu unterscheiden, wer, was, wann, wo am System tun darf oder nicht. Und dies regelt die Sicherheitsrichtlinie, welche unter secpol.msc zu finden ist.



Fügen Sie einen Benutzer in die Gruppe der Administratoren hinzu, so darf dieser laut der Liste Gerätetreiber installieren oder auch Dateien sichern. Würde es keine Gruppen(-richtlinien) geben, so müsste man bei jedem einzelnen Benutzer jedes einzelne Recht hinzufügen oder entfernen. Mithilfe von Gruppen braucht man dies nur einmal für die Gruppe und kann später alle gewünschten Benutzer in diese Gruppe hinzufügen. Somit spart man sehr viel Zeit als IT-Administrator.

Dateimanagement

Eine Hauptaufgabe des Betriebssystems ist die Verwaltung der Daten. Dabei werden Daten in sogenannten **Dateien** zusammengefasst, die auf Festplatten und anderen Speichermedien abgespeichert werden können. Eine Datei lässt sich mit dem Inhalt einer Karteikarte vergleichen.

Mehrere Dateien („Karteikarten“) werden in einem Verzeichnis (directory) abgelegt. Ein Verzeichnis ist vergleichbar mit einer Schachtel, in der verschiedene Karteikarten untergebracht werden können. Selbstverständlich ist es auch möglich, in einer Schachtel mehrere kleinere Schachteln unterzubringen (Unterverzeichnisse - subdirectory) usw.

Klarerweise entsteht dadurch ein baumartiges System, weshalb man dieses auch als „Tree“ bezeichnet. Die größte Schachtel, die alle anderen enthält bezeichnet man als **Hauptverzeichnis** oder **Root** (Wurzel).

Dateiformate

Die Art und Weise, wie Informationen innerhalb einer Datei gespeichert werden, und wie diese zu interpretieren sind, wird als Dateiformat bezeichnet. Um den Inhalt einer Datei wieder benutzen oder weiterverwenden zu können, muss bekannt sein, auf welche Weise und in welcher Reihenfolge die Informationen in der Datei abgelegt wurden.

Für unterschiedliche Inhalte und Einsatzzwecke gibt es zahlreiche Methoden zur Speicherung in Dateien, die unter dem Oberbegriff **Dateiformate** zusammengefasst werden. Um also z.B. den Inhalt eines einfachen Briefes in einer Datei abzulegen, wird ein auf Textspeicherung ausgerichtetes Format verwendet, während bei der Speicherung eines Bildes ein entsprechendes Grafikformat verwendet wird.

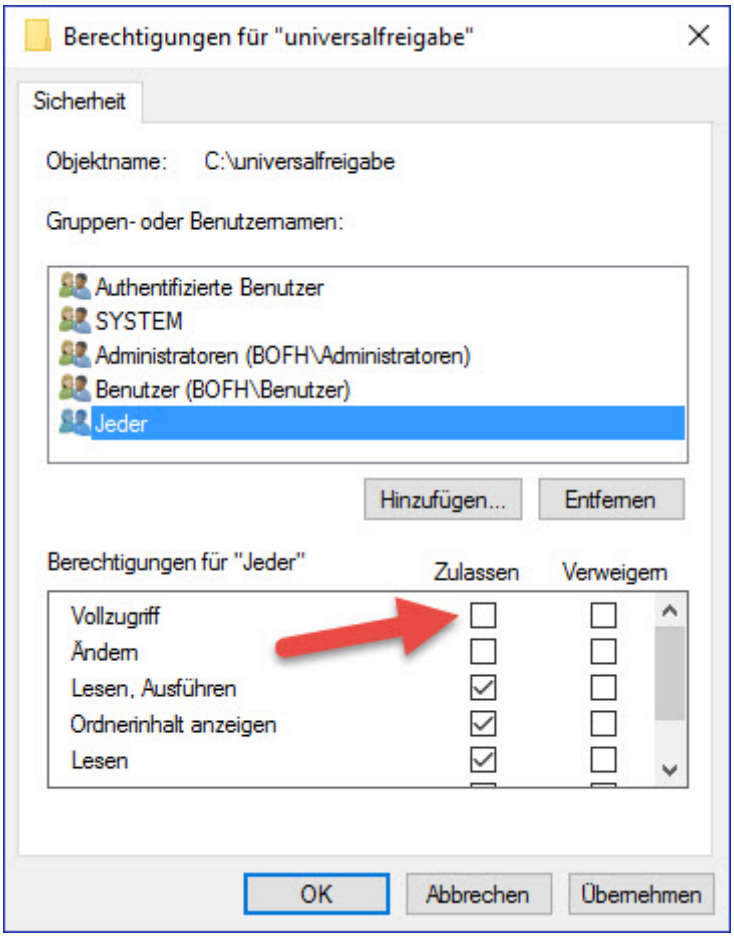
Vor allem bei Microsoft-Betriebssystemen hat es sich eingebürgert, einen Hinweis auf das verwendete Dateiformat durch einen Zusatz am jeweiligen Namen der Datei anzubringen. Diesen Hinweis nennt man **Dateiendung** und wird durch einen **Punkt** vom Dateinamen getrennt.

Übersicht über einige Standarderweiterungen

für ausführbare Dateien	
EXE	ausführbare Datei
BAT	Batch-File
programmspezifische Erweiterungen	
DOC	formatiertes Dokument
TXT	Textdatei ohne Sonderzeichen
SYS	System(Programm)datei
DBF	Datenbankfile
…	…

Dateirechte

Da nicht jeder Benutzer immer alle Rechte auf einem Rechner haben darf/soll/muss, gibt es im Dateisystem (z.B. NTFS) die Rechte für die jeweiligen Benutzer bzw. Gruppen zu definieren. Diese Dateirechte sind die Sicherheitseinstellungen, die man auf einer NTFS-Partition einem Dateiobjekt vergibt. Diese werden üblicherweise im Kontextmenü des Objektes unter Eigenschaften > Sicherheitseinstellungen vergeben:



Es gibt die folgenden Rechte in der Registerkarte Sicherheit bei einem Dateiobjekt

Berechtigung	**Rechte**
Vollzugriff / Full Control	Alle
Ändern / Modify	Alle, außer Berechtigungen ändern und Besitzerrechte übernehmen
Lesen & Ausführen / Read & Execute	Datei öffnen und lesen, ausführbare Dateien und Batchdateien starten
Ordnerinhalt auflisten / List Folder Contents	Nur bei Ordner: Lesen und Lesen & Ausführen
Lesen / Read	Datei öffnen und lesen
Schreiben / Read	Datei ändern oder neu erzeugen
Spezielle Berechtigungen	Siehe unten

Kopieren / Verschieben von Dateien

Wenn eine Datei kopiert wird, wird sie am Zielort neu erstellt und erbt daher die Berechtigungen des Ordners, in den sie kopiert wurde. Beim Verschieben innerhalb einer Partition behält die Datei Ihre Ursprungsrechte.

Will man Dateien verschieben und Berechtigungen des Zielordners annehmen lassen, kopiert man also am Besten und löscht dann die Ursprungsdaten.

Sichern und Wiederherstellen in Windows

Ein Windows 10 Backup könnt ihr mit integrierten Backup-Tools im Handumdrehen erstellen, eure persönlichen Daten durch eine Sicherung schützen und ein komplettes Windows-Systemabbild mit Bordmitteln erstellen.

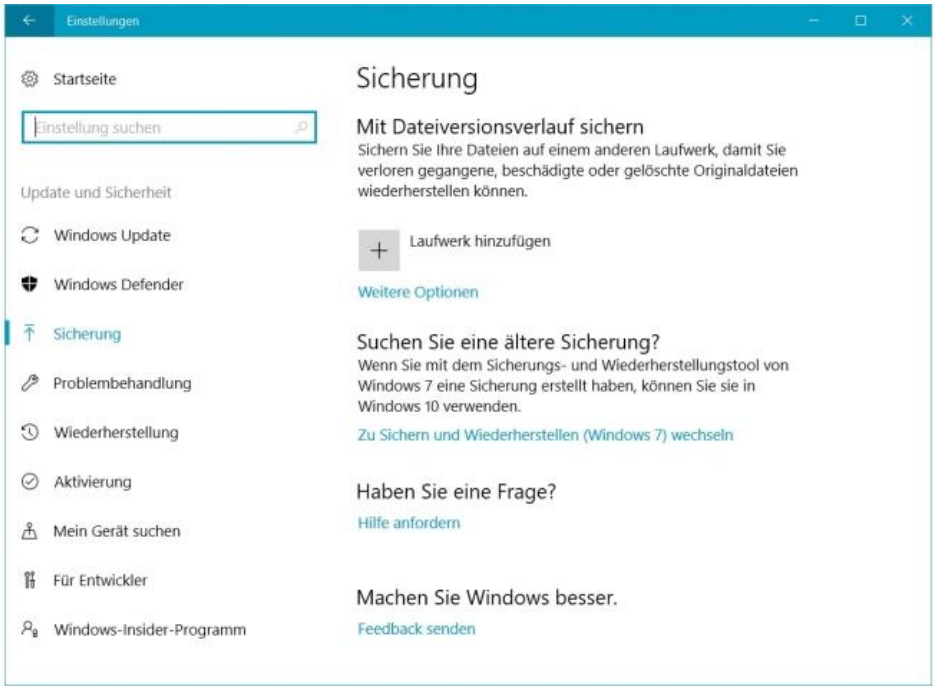
Mit den Windows 10-Bordmitteln könnt ihr bei einem Defekt der Festplatte nicht nur einzelne Daten, sondern euer komplettes Windows-System wiederherstellen. Außerdem könnt ihr mit den Windows-Systemprogrammen eure Daten automatisch sichern.

Hier sind die einzelnen Varianten beschrieben:

1) Backup erstellen und einzelne Dateien sichern

Mit den folgenden Schritten könnt ihr einzelne Ordner und Dateien, etwa Bilder, Videos und Dokumente, automatisch sichern und einzelne Dateien im Fehlerfall wiederherstellen:

- Öffnet unten links das vierkachelige Windows-Startmenü und klickt auf den Reiter „Einstellungen“.
- Wählt im neuen Fenster das Feld „Update und Sicherheit“ und klickt anschließend auf die Kategorie „Sicherheit“.
- Klickt auf „Laufwerk hinzufügen“ und Windows listet euch alle verfügbaren Datenträger aus. Wählt die Festplatte oder den USB-Stick aus, auf dem eure Daten gesichert werden sollen.
- Klickt danach auf „Weitere Optionen“, wo ihr über „Ordner hinzufügen“ weitere Ordner und Daten dem Backup hinzufügen könnt. Über das Drop-Down-Menü bei „Meine Daten sichern“ könnt ihr außerdem auswählen, ob und wie oft eure Daten automatisch gesichert werden sollen.
- Schießt die Datensicherung mit einem Klick auf „Jetzt sichern“ ab.

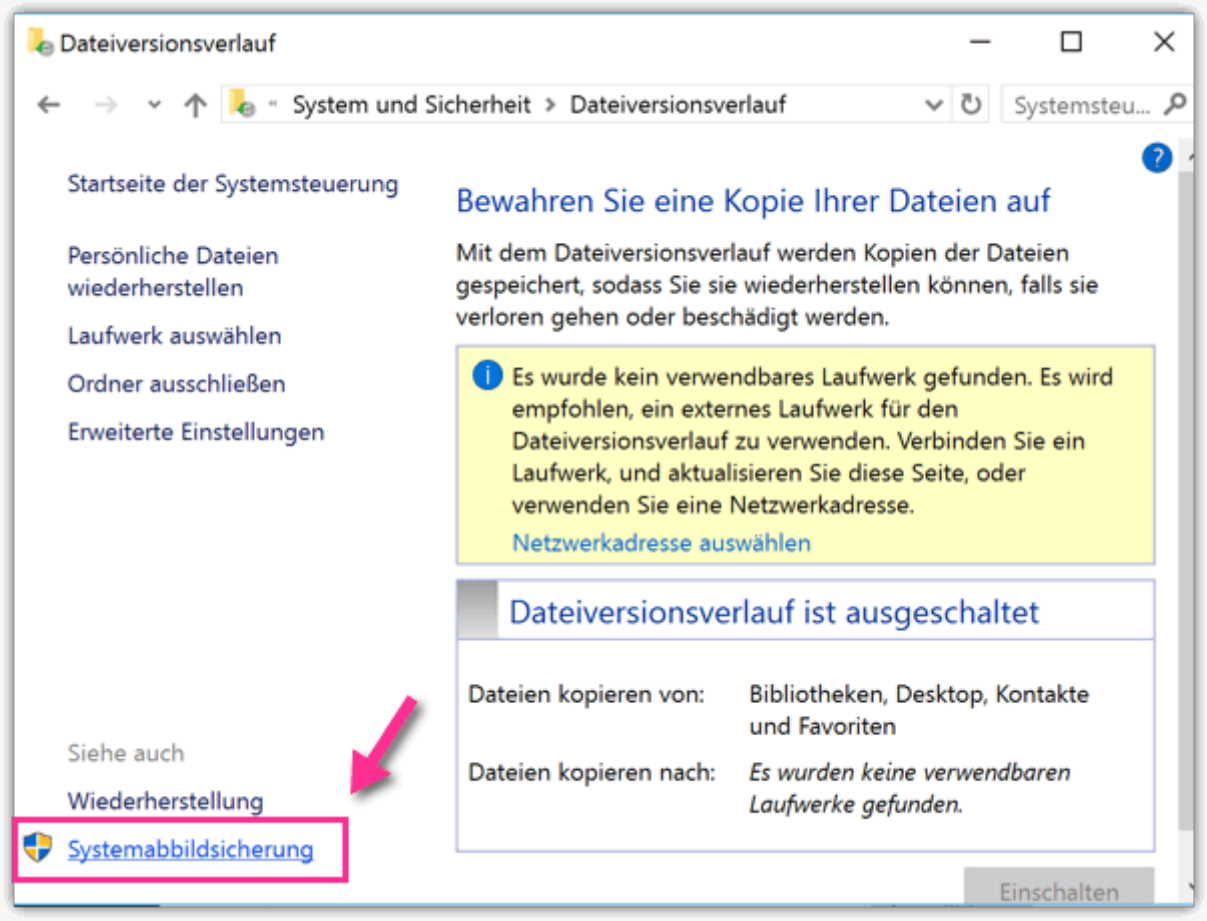


2) Komplettes Systemabbild mit Bordmitteln erstellen

Wollt ihr aber euer komplettes System absichern und ein Windows 10-Systemabbild erstellen, so sind folgende Schritte notwendig. Damit kann man nicht nur einzelne Dateien sondern auch das komplette Betriebssystem im Fehlerfall wiederherstellen.

Achtung!!: Speichert das Backup nie auf das Systemlaufwerk, denn im Falle eines Festplattendefekts hilft das beste Backup nichts!! Als Speicherort eignet sich am besten eine externe Festplatte oder ein Serverlaufwerk mit genügende Speicherplatz.

- Navigiert, wie oben beschrieben, erneut in das „Weitere Optionen“-Menü.
- Scrollt in den Sicherungsoptionen bis an das Ende der Seite und klickt auf „Siehe erweiterte Einstellungen“.
- Wählt im neu geöffneten Fenster unten links „Systemabbildsicherung“ und anschließend „Systemabbild erstellen“ aus.
- Wählt die Festplatte aus, auf dem das Systemabbild erstellt werden soll und bestätigt mit „Weiter“.
- Windows zeigt euch an, welche Laufwerke mit dem Backup gesichert werden. Schließt den Vorgang mit „Sicherung starten“ ab



3) Alternative Backup-Methoden

Neben den Windows-Bordmitteln könnt ihr für eure Datensicherung auch auf verschiedene Software-Lösungen zurückgreifen.

- Acronis
- Duplicati

LINUX

Linux ist ein freies Multiplattform-Mehrbenutzer-Betriebssystem, das den Linux-Kernel enthält. Im praktischen Einsatz werden meist sogenannte Linux-Distributionen genutzt, in denen der Linux-Kernel und verschiedene Software zu einem fertigen Paket zusammengestellt sind.

Das wichtigste gleich vorweg, in Linux ist alles eine Datei –> Everything is a file.

Everything is a file beschreibt eine der definierenden Eigenschaften von Unix und seinen Abkömmlingen, demnach Ein-/Ausgabe-Ressourcen wie Dateien, Verzeichnisse, Geräte (z. B. Festplatten, Tastaturen, Drucker) und sogar Interprozess- und Netzwerk-Verbindungen als einfache Byteströme via Dateisystem verfügbar sind

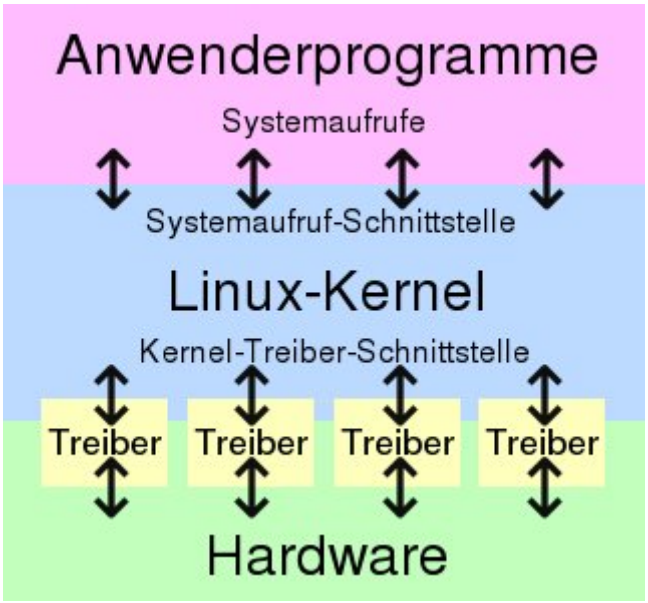
- Aufbau des Betriebssystems
- Benutzer
- Dateimanagement
- Dateirechte
- Inodes
- Distributionen und Desktops
- Konsole/Bash/Terminal
- Shell Scripts

Aufbau des Betriebssystems

Das zentrale Kernstück des Betriebssystems, der Linux-Kernel (meist nur Kernel genannt) bildet eine Trennschicht zwischen Hardware und Anwenderprogrammen. Das heißt, wenn ein Programm auf ein Stück Hardware zugreifen will, so kann es niemals direkt darauf zugreifen, sondern nur über das Betriebssystem.

Dazu bedient sich das Programm der **Systemaufrufe**. Über den Systemaufruf teilt das Anwenderprogramm dem Betriebssystem mit, dass es etwas zu tun gibt. Will etwa ein Programm eine Zeile Text auf dem Bildschirm ausgeben, so wird ein Systemaufruf gestartet, dem der Text übergeben wird. Das Betriebssystem erst schreibt ihn auf den Bildschirm.

Auf der anderen Seite muss das Betriebssystem die Möglichkeit haben, mit den einzelnen Hardware-Komponenten zu sprechen. Mittels seiner **Treiberschnittstelle** spricht es spezielle Geräte-Treiber an. Erst die Treiber kommunizieren dann direkt mit den Geräten.



Zu den **Anwenderprogrammen** zählen alle von uns gestarteten Programme (Videoplayer, Webbrowser ...), wie auch die grafische Oberfäche des Betriebssystems, das Desktop-Environment. Letzteres ist nicht ein Programm, sondern eine Sammlung von Programmen, die zusammen die gewohnten Funktionalitäten beisteuern.

Ein ganz spezielles Anwenderprogramm ist die Shell - die „Benutzeroberfläche“. Es existieren viele verschiedene Shells - wir werden hier mit der Bash (Bourne again shell) arbeiten. Diese ist die Standardshell auf Linuxsystemen. Alle Shells stellen dem Benutzer eine Kommandozeile zur Verfügung, mit der Befehle eingegeben werden können, die direkt als Systemaufrufe an das Betriebssystem weitergeleitet werden.

Linux ist ein **Multitasking-Betriebssystem**: das heißt, es können mehrere Prozesse - so nennt man Programme, sobald sie in den Speicher geladen sind und laufen - gleichzeitig laufen. Das bedingt, dass das System die verfügbare Rechenzeit des Prozessors in kleine Zeitscheiben aufteilt (im Millisekundenbereich), die dann den jeweiligen Prozessen zur Verfügung stehen. Diese Aufgabe übernimmt eine übergeordnete Instanz - der Scheduler. Dieser verwaltet die Zuteilung der Zeitscheiben an die verschiedenen Prozesse.

Daher kann kein Prozess die ganze Rechenleistung für sich beanspruchen und auch ein „hängender“ Prozess kann nicht das ganze System lahmlegen.

Benutzer

Linux ist auch ein **Multiuser**-System, das heißt, es können mehrere Benutzer an verschieden Terminals auf dem selben Rechner arbeiten. Dazu ist es natürlich notwendig, dass jeder Benutzer eindeutig identifiziert ist. Die User (engl., Benutzer) werden zwar mit ihren Namen verwaltet, intern arbeitet ein Unix-System aber mit Usernummern. Jeder Benutzer hat also eine Nummer welche UserID oder kurz UID genannt wird. Jeder Benutzer ist auch Mitglied mindestens einer Gruppe. Es kann beliebig viele Gruppen in einem System geben und auch sie haben intern Nummern (GroupID oder GID). Im Prinzip sind Gruppen nur eine Möglichkeit, noch detailiertere Einstellungsmöglichkeiten zu haben, wer was darf.

Eine spezielle Rolle hat der Benutzer mit der UserID 0 - er ist Root (engl., Wurzel). Root steht außerhalb aller Sicherheitseinrichtungen des Systems - kurz - er darf alles. Er kann mit einem Befehl das ganze System zerstören, er kann die Arbeit von Wochen und Monaten löschen usw. Aus diesem Grund meldet sich auch der Systemverwalter im Normalfall als normaler Benutzer an - zum Root-Benutzer wird er nur dann, wenn er Systemverwaltungsarbeiten abwickelt, die diese Identität benötigen.

Benutzertypen

root

Der Benutzer root ist mit allen Rechten ausgestattet, die ihm die Administration (bei Unachtsamkeit natürlich auch die Beschädigung!) des Systems erlauben. Diesem auch als Superuser bezeichneten Benutzer ist immer die UID 0 zugeordnet.

Systembenutzer

Je nach System kann eine Vielzahl von Prozessen und Diensten erwünscht sein, die bereits beim Hochfahren des Systems verfügbar sein sollen. Nicht jeder dieser Prozesse benötigt jedoch die volle Rechteausstattung des Superusers. Man möchte natürlich so wenige Prozesse wie nur möglich unter einer root Kennung starten, da die weitreichenden Rechte solcher Prozesse unnötige Möglichkeiten für Missbrauch und Beschädigung des Systems liefern.

Ein Systembenutzerkonto ist in diesem Sinne ein Benutzerkonto, das jedoch (nahezu) ausschließlich zur Ausführung von Programmen unter einer speziellen Benutzerkennung verwendet wird. Kein menschlicher Benutzer meldet sich normalerweise unter einem solchen Konto an.

Standardbenutzerkonto

Dies ist das normale Benutzerkonto, unter welchem jeder üblicherweise arbeiten sollte.

Die zentralen Benutzerdateien

Die Dateien zur Benutzerverwaltung finden Sie unter Linux im Verzeichnis /etc. Es handelt sich dabei um die Dateien **/etc/passwd**, **/etc/shadow** und **/etc/group**.

/etc/passwd

Die Datei /etc/passwd ist die zentrale Benutzerdatenbank.

Mit cat /etc/passwd können Sie einen Blick in diese zentrale Benutzerdatei werfen. Hier werden alle Benutzer des Systems aufgelistet. Zu beachten ist, dass alle Benutzertypen eingetragen sind, also sowohl der Superuser root als auch die Standard- und Systembenutzer.

Ein Benutzerkonto in der Datei /etc/passwd hat generell folgende Syntax:

```
Benutzername : Passwort : UID : GID : Info : Heimatverzeichnis : Shell
```

Spalte	**Erklärung**
Benutzername	Dies ist der Benutzername in druckbare Zeichen, meistens in Kleinbuchstaben.
Passwort	Hier steht verschlüsselt das Passwort des Benutzers (bei alten Systemen). Meist finden Sie dort ein x. Dies bedeutet, dass das Passwort verschlüsselt in der Datei /etc/shadow steht. Es ist auch möglich, den Eintrag leer zu lassen. Dann erfolgt die Anmeldung ohne Passwortabfrage (in der Datei /etc/shadow muss dann an Stelle des verschlüsselten Passwortes ein * stehen).
UID	Die Benutzer-ID des Benutzers. Die Zahl hier sollte größer als 100 sein, weil die Zahlen unter 100 für Systembenutzer vorgesehen sind. Weiterhin muss die Zahl aus technischen Gründen kleiner als 64000 sein.
GID	Die Gruppen-ID des Benutzers. Auch hier muss die Zahl wie bei der UID kleiner als 64000 sein.
Info	Hier kann weitere Information vermerkt werden, wie z.B. der vollständige Name des Benutzers und persönliche Angaben (Telefonnummer, Abteilung, Gruppenzugehörigkeit u.ä.).
Heimatverzeichnis	Das Heimatverzeichnis des Benutzers bzw. das Startverzeichnis nach dem Login.
Shell	Die Shell, die nach der Anmeldung gestartet werden soll. Bleibt dieses Feld frei, dann wird die Standardshell /bin/sh gestartet.

Hier ein Beispiel für einen Systembenutzer:

```
uucp:x:10:14:Unix-to-Unix CoPy system:/etc/uucp:/bin/bash
```

Der Benutzer heißt uucp, das Passwort ist in der Datei /etc/shadow gespeichert (x), die UID ist 10, die GID 14, als erläuternde Bezeichnung trägt der Benutzer den Namen „Unix-to-Unix CoPy system“, das Startverzeichnis nach der Anmeldung ist /etc/uucp, und die vorgeschlagene Shell ist die bash.

An dieser Stelle sei nochmals darauf hingewiesen, dass die meisten Linux-Distributionen komfortable Werkzeuge zur Benutzerverwaltung mitliefern und es auch eine Reihe von Befehlen gibt, die für die Benutzerverwaltung verwendet werden können

/etc/shadow

Bei früheren Versionen von Linux speicherte man die die Passwörter direkt in die passwd-Datei. Allerdings war dies durch einen sogenannten Wörterbuchangriff und der beispielsweise mit Hilfe des Programmes crypt möglich, diese Passwörter in vielen Fällen zu entschlüsseln und auszulesen. Deshalb hat man die Datei /etc/shadow eingeführt, in der die Angaben über die Passwörter durch ein spezielles System besser geschützt werden.

Der Eintrag in diese Datei erfolgt nach einem ähnlichen Schema, wie in der Datei /etc/passwd:

```
Benutzername : Passwort : DOC : MinD : MaxD : Warn : Exp : Dis : Res
```

Benutzername	**Dies ist der Benutzername in druckbaren Zeichen, meistens in Kleinbuchstaben.**
Passwort	Hier steht verschlüsselt das Passwort des Benutzers. Wenn hier ein * oder ! steht, dann bedeutet dies, dass kein Passwort vorhanden bzw. eingetragen ist.
DOC	Day of last change: der Tag, an dem das Passwort zuletzt geändert wurde. Besonderheit hier: Der Tag wird als Integer-Zahl in Tagen seit dem 1.1.1970 angegeben.
MinD	Minimale Anzahl der Tage, die das Passwort gültig ist.
MaxD	Maximale Anzahl der Tage, die das Passwort gültig ist.
Warn	Die Anzahl der Tage vor Ablauf der Lebensdauer, ab der vor dem Verfall des Passwortes zu warnen ist.
Exp	Hier wird festgelegt, wieviele Tage das Passwort trotz Ablauf der MaxD noch gültig ist.
Dis	Bis zu diesem Tag (auch hier wird ab dem 1.1.1970 gezählt) ist das Benutzerkonto gesperrt
Res	Reserve, dieses Feld hat momentan keine Bedeutung.

Es folgt wieder ein Beispiel:

```
selflinux:/heSIGnYDr6MI:11995:1:99999:14:::
```

Der Benutzer heißt selflinux, das Passwort lautet verschlüsselt „/heSIGnYDr6MI“. Es wurde zuletzt geändert, als 11995 Tage seit dem 1.1.1970 vergangen waren. Das Passwort ist minimal einen Tag gültig, maximal 99999 Tage (was man als immer deuten kann - 99999 Tage sind ca. 274 Jahre). Es soll ab 14 Tage vor Ablauf des Passwortes gewarnt werden. Die anderen Werte sind vom Administrator nicht definiert und bleiben daher leer.

/etc/group

In dieser Datei finden Sie die Benutzergruppen und ihre Mitglieder. In der Datei /etc/passwd wird mit der GID eigentlich schon eine Standardgruppe für den Benutzer festgelegt. In der /etc/group können Sie weitere Gruppenzugehörigkeiten definieren. Das hat in der Praxis vor allem in Netzwerken eine große Bedeutung, weil Sie so in der Lage sind, z.B. Gruppen für Projekte oder Verwaltungseinheiten zu bilden. Für diese Gruppen kann man dann entsprechend die Zugriffsrechte einstellen. Dies hat dann wiederum den Vorteil, dass man die Daten gegen eine unbefugte Benutzung absichern kann.

Der Eintrag einer Gruppe in die Datei sieht so aus:

```
Gruppenname : Passwort : GID : Benutzernamen (Mitgliederliste)
```

Gruppenname	**Der Name der Gruppe in druckbare Zeichen, auch hier meistens Kleinbuchstaben.**
Passwort	Die Besonderheit hier ist folgende: Wenn das Passwort eingerichtet ist, können auch Nichtmitglieder der Gruppe Zugang zu den Daten der Gruppe erhalten, wenn ihnen das Passwort bekannt ist. Ein x sagt hier aus, das das Passwort in /etc/gshadow abgelegt ist. Der Eintrag kann auch entfallen, dann ist die Gruppe nicht durch ein Passwort geschützt. In diesem Fall kann jedoch auch kein Benutzer in die Gruppe wechseln, der nicht in diese Gruppe eingetragen ist.
GID	Gruppen-ID der Gruppe
Benutzernamen	hier werden die Mitglieder der Gruppe eingetragen. Diese sind durch ein einfaches Komma getrennt.

Für einen korrekten Eintrag in die /etc/group reicht eigentlich der Gruppenname und die GID aus. Damit ist die Gruppe dem System bekannt gemacht. Die Felder für das Passwort und die Benutzernamen können frei bleiben.

Soll der Benutzer nur in seiner Standardgruppe bleiben, ist kein Eintrag in die /etc/group notwendig. Hier reicht der Eintrag in die /etc/passwd völlig aus, weil dort die Standardgruppe schon mit angegeben wird. Nur wenn der Benutzer in weiteren bzw. mehreren Gruppen Mitglied sein soll, muss dies in die /etc/group-Datei eingetragen werden. Für Passwörter gilt das oben in der Tabelle Gesagte.

Hier sehen Sie ein Beispiel für einen Eintrag:

```
dialout:x:16:root,tatiana,steuer,selflinux
```

Sie sehen eine Gruppe mit der GID „16“ und den Namen dialout. (Zur Information: dialout erlaubt es normalen Benutzern einen ppp-Verbindungsaufbau zu starten, normalerweise hat nur root dieses Recht). Das x bedeutet hier, dass das Passwort in /etc/shadow abgelegt ist. Da in /etc/gshadow hier bei Passwort ein * steht, ist also kein Passwort für die Gruppe vorhanden (Das bedeutet wiederum, das nur die eingetragenen Mitglieder Zugang zu dieser Gruppe haben). Mitglieder der Gruppe sind: root, tatiana, steuer, selflinux.

Benutzerklassen: user, group und others

Aus der Sicht des Systems existieren drei Benutzerklassen, wenn entschieden werden soll, ob die Berechtigung für einen Dateizugriff existiert oder nicht. Soll beispielsweise eine Datei gelöscht werden, so muss das System ermitteln, ob der Benutzer, welcher die Datei löschen möchte, das erforderliche Recht besitzt:

user@linux ~$ rm testdatei rm: Entfernen (unlink) von „testdatei“ nicht möglich: Keine Berechtigung

In diesem Fall wurde dem rm Kommando der beabsichtigte löschende Zugriff auf die Datei verwehrt - der ausführende Benutzer hatte nicht das Recht, die Datei zu löschen. Um diese Entscheidung zu treffen, verwendet das System das Konzept der Benutzerklassen. Drei Benutzerklassen werden unterschieden: user, group und others. Jede dieser Benutzerklassen ist wiederum in ein Lese-, Schreib- und Ausführrecht unterteilt. Diese werden im Folgenden als Berechtigungsklassen bezeichnet. Somit ergibt sich folgende Körnung für die einfachen Zugriffsrechte einer Datei:

Recht	Beschreibung
user-read	Leserecht für Dateieigentümer
user-write	Schreibrecht für Dateieigentümer
user-execute	Ausführrecht für Dateieigentümer
group-read	Leserecht für Gruppe des Dateieigentümers
group-write	Schreibrecht für Gruppe des Dateieigentümers
group-execute	Ausführrecht für Gruppe des Dateieigentümers
other-read	Leserecht für alle anderen Benutzer
other-write	Schreibrecht für alle anderen Benutzer
other-execute	Ausführrecht für alle anderen Benutzer

Benutzerklassen sind also eng mit der Eigentümerschaft von Dateien verbunden. Jede Datei und jedes Verzeichnis ist sowohl einem Benutzer (einer UID) als auch einer Gruppe (einer GID) zugeordnet. UID und GID gehören zur elementaren Verwaltungsinformation von Dateien und Verzeichnissen und werden in der sogenannten Inode gespeichert.

Beim Zugriff auf eine Datei werden nun UID und GID des zugreifenden Prozesses mit UID und GID der Datei verglichen. Ist other-read gesetzt, darf jeder Benutzer lesend zugreifen und ein weiterer Vergleich erübrigt sich. Ist lediglich group-read gesetzt, muss der Zugreifende mindestens der Gruppe des Dateieigentümers angehören, d.h. eine identische GID aufweisen. Ist ausschließlich user-read gesetzt, so darf nur der Eigentümer selbst die Datei lesen. root ist von dieser Einschränkung freilich ausgenommen. („Ich bin root, ich darf das!“). Von sehr speziellen Ausnahmen abgesehen, die sich außerhalb der hier besprochenen Rechteklassen bewegen, ist root in seinen Aktionen in keinerlei Weise eingeschränkt.

Dateimanagement

Nachdem unter Linux das Prinzip **Everything is a file** gilt, werden hier die Besonderheiten von Dateien beschrieben.

Dateien

Datei- und Verzeichnisnamen können bis zu 255 Zeichen lang sein. Dabei wird in jedem Fall zwischen Groß- und Kleinschreibung unterschieden. Die Dateinamen

- DATEI
- datei
- Datei

bezeichnen drei unterschiedliche Dateien. Ein Dateiname darf beliebig viele Punkte enthalten, also zum Beispiel auch Datei.Teil.1.txt. Ein Punkt gilt als normales Zeichen in einem Dateinamen. Dateien, die mit Punkt beginnen, gelten als versteckt und werden normalerweise nicht angezeigt - zum Beispiel .datei. Das Zeichen zum Trennen von Verzeichnis- und Dateinamen ist der Slash („/“) statt dem Backslash („\“) bei Windows.

Es gibt verschiedene Dateiarten: (in Klammer die offizielle Darstellung, wie sie symbolisiert werden)

- Normale Dateien (-)
- Verzeichnisse (d)
- Symbolische Links (l)
- Blockorientierte Geräte (b)
- Zeichenorientierte Geräte (c)
- Named Pipes (p)

Wir sehen hier schon, dass auch Verszeichnisse bloß eine bestimmte Dateiart sind. Eine spezielle nämlich, in der andere Dateien aufgelistet sind. Mit einem Dateibrowser (von Windows kennen wir „Explorer“, bei Apple den „Finder“) sehen wir uns immer nur genau diese Verzeichnisse an, sofern wir nicht mittels verschiedener Plugins die Dateien selbst auswerten und Textdokumente, Bilder anzeigen oder Videos und Musik wiedergeben.

In einem Unix-Dateisystem hat jede einzelne Datei jeweils einen Eigentümer und eine Gruppenzugehörigkeit. Neben diesen beiden Angaben besitzt jede Datei noch einen Satz Attribute, die bestimmen, wer die Datei wie benutzen darf. Diese Attribute werden dargestellt als „rwx“. Dabei steht r für lesen (read), w für schreiben (write) und x für ausführen (execute).

Das Dateisystem

Das Dateisystem ist die Ablageorganisation auf einem Datenträger eines Computers. Um die Funktionsweise zu verstehen, betrachten wir einen Datenträger, die Festplatte, näher:

Die Festplatte besteht aus mehreren Scheiben mit einer magnetisierbaren Oberfläche, auf die die

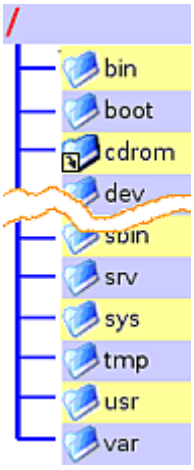
Schreibköpfe unsere Daten als Einsen (ein) und Nullen (aus) abspeichern. Um diese aber vernünftig adressieren zu können, benutzen wir Dateisysteme. Ein solches teilt die Festplatte (eigentlich die „Partition“, denn die Festplatte wird häufig in mehrere Partitionen aufgeteilt, die dann unabhängig formatiert werden können) in kleine Einheiten, die „Blöcke“, welche aus Performancegründen häufig noch zu „Clustern“ zusammengefasst werden.

Der Block (oder Cluster) ist dann die kleinste Einheit, in die eine Datei geschrieben wird, jede Datei benötigt dadurch immer diesen Speicherplatz (oder ein vielfaches) auf der Festplatte.

Von Windows kennen wir NTFS und FAT32, bzw. Apple-Benutzer werden schon von HFS+ gehört haben. Unter Linux werden meist ext2, ext3 oder ext4 (Second, Third bzw. Fourth Extented File System) verwendet. „ext3“ unterscheidet sich von „ext2“ nur dadurch, dass zusätzlich ein „Journal“ geschrieben wird, welches bei Systemabstürzen eine zuverlässige Wiederherstellung möglich macht. „ext4“ ist eine performantere Weiterentwicklung von „ext3“ und heute Standard. Daneben gibt es gelegentlich noch ReiserFS, XFS oder JFS, aber die Wahl des Dateisystems bestimmt tatsächlich immer das Abwägen zwischen höherer Sicherheit und schnellerer Schreibgeschwindigkeit - mit oder ohne Journal.

Verzeichnisstruktur

Das Dateisystem beginnt mit einem Wurzelverzeichnis (auch Rootverzeichnis genannt - /). Es enthält im Regelfall keine Dateien, sondern nur die folgenden Verzeichnisse (Ubuntu):



/bin

Von: binaries (Programme); muss bei Systemstart vorhanden sein; enthält für Linux unverzichtbare Programme; diese Programme können im Gegensatz zu /sbin von allen Benutzern ausgeführt werden; /bin darf keine Unterverzeichnisse enthalten.

/boot

Muss bei Systemstart vorhanden sein; Enthält zum Booten benötigte Dateien.

/dev

Von devices (Geräte); muss bei Systemstart vorhanden sein; enthält alle Gerätedateien, über die die Hardware im Betrieb angesprochen wird

/etc

Von: et cetera („alles übrige“), später auch: editable text configuration (änderbare Text Konfiguration); muss bei Systemstart vorhanden sein; enthält Konfigurations- und Informationsdateien des Basissystems.

- /etc/init.d: dort liegen alle Start- und Stopskripte
- /etc/opt: Verzeichnisse und Konfigurationsdateien für Programme in /opt
- /etc/network: Verzeichnisse und Konfigurationsdateien des Netzwerkes
- ….

/home

Von: home-directory (Heimatverzeichnis); enthält pro Benutzer ein Unterverzeichnis; jedes Verzeichnis wird nach dem Anmeldenamen benannt

/lib

Von: libraries (Bibliotheken); muss bei Systemstart vorhanden sein; enthält unverzichtbare Bibliotheken fürs Booten und die dynamisch gelinkten Programme des Basissystems;

/lost+found

(verloren und gefunden); Dateien und Dateifragmente, die beim Versuch, ein defektes Dateisystem zu reparieren, übrig geblieben sind.

/media

Für (Speicher-)Medien. Enthält Unterverzeichnisse, welche als mount- oder Einhängepunkte für transportable Medien wie z.B. externe Festplatten, USB-Sticks, CD-ROMs, DVDs und andere Datenträger dienen. Ubuntu legt hier auch die Einhängepunkte für Partitionen an. Unterverzeichnisse sind u.a.:

- /media/floppy: Einhängepunkt für Disketten
- /media/cdrom0: Einhängepunkt für CD-ROMs

/mnt

Von: mount (eingehängt); normalerweise leer; kann für temporär eingehängte Partitionen verwendet werden. Für Datenträger, die hier eingehängt werden, wird im Gegensatz zu /media kein Link auf dem Desktop angelegt

/opt

Von: optional; ist für die manuelle Installation von Programmen gedacht, die ihre eigenen Bibliotheken mitbringen und nicht zur Distribution gehören;

/proc

Von: processes (laufende Programme); muss bei Systemstart vorhanden sein; enthält Schnittstellen zum aktuell geladenen Kernel und seinen Prozeduren; Dateien lassen sich mittels cat auslesen; Beispiele: version (Kernelversion), swaps (Swapspeicherinformationen), cpuinfo, interrupts, usw.;

/root

Ist das Homeverzeichnis des Superusers (root). Der Grund, wieso sich das /root-Verzeichnis im Wurzelverzeichnis und nicht im Verzeichnis /home befindet, ist, dass das Homeverzeichnis von Root immer erreichbar sein muss, selbst wenn die Home-Partition aus irgendeinem Grund (Rettungs-Modus, Wartungsarbeiten) mal nicht eingehängt ist.

/sbin

Von: system binaries (Systemprogramme); muss bei Systemstart vorhanden sein; enthält alle Programme für essentielle Aufgaben der Systemverwaltung; Programme können nur vom Systemadministrator (root) oder mit Superuserrechten ausgeführt werden

/srv

Von: services (Dienste); Verzeichnisstruktur noch nicht genau spezifiziert; soll Daten der Dienste enthalten; unter Ubuntu in der Regel leer

/sys

Von: system; im FHS noch nicht spezifiziert; erst ab Kernel 2.6. im Verzeichnisbaum enthalten; besteht ebenso wie /proc hauptsächlich aus Kernelschnittstellen

/tmp

Von: temporary (temporär); enthält temporäre Dateien von Programmen; Verzeichnis soll laut FHS beim Booten geleert werden.

/usr

Von: user (siehe: Herkunft); enthält die meisten Systemtools, Bibliotheken und installierten Programme; der Name ist historisch bedingt - früher, als es /home noch nicht gab, befanden sich hier auch die Benutzerverzeichnisse;

- /usr/bin : Anwenderprogramme; Hier liegen die Desktopumgebungen und die dazu gehörigen Programme, aber auch im Nachhinein über die Paketverwaltung installierte Programme, wie Audacity

/var

Von variable (variabel); enthält nur Verzeichnisse; Dateien in den Verzeichnissen werden von den Programmen je nach Bedarf geändert (im Gegensatz zu /etc); Beispiele: Log-Dateien, Spielstände, Druckerwarteschlange

- /var/log : Alle Log-Dateien der Systemprogramme; Beispiele: Xorg.0.log (Log-Datei des XServer), kern.log (Logdatei des Kernels), dmesg (letzte Kernelmeldungen), messages (Systemmeldungen); Siehe auch Logdateien

DATEIRECHTE

UNIX-Systeme wie Linux verwalten ihre Dateien in einem virtuellen Dateisystem (VFS, Virtual File System). Dieses ordnet jeder Datei über eindeutig identifizierbare Inodes unter anderem folgende Eigenschaften zu:

- Dateityp (einfache Datei, Verzeichnis, Link, …)
- Zugriffsrechte (Eigentümer-, Gruppen- und sonstige Rechte)
- Größe
- Zeitstempel
- Verweis auf Dateiinhalt

Jedes unter Linux gängige UNIX-Dateisystem (z.B. ext2/3/4, ReiserFS, xfs usw.) unterstützt diese Rechte. Gar nicht umgesetzt werden die Rechte hingegen auf VFAT-Dateisystemen; dort können Dateirechte lediglich beim Einhängen simuliert werden. Partitionen mit dem Windows-Dateisystem NTFS werden zwar in Linux standardmäßig ähnlich wie VFAT-Partitionen behandelt; mit den Mount-Optionen permissions und acl lässt sich aber auch auf NTFS-Partitionen eine echte Rechteverwaltung wie bei UNIX-Dateisystemen einrichten. Siehe hierzu Windows-Partitionen einbinden sowie NTFS-3G.

Rechte in symbolischer Darstellung

Im Terminal lassen sich die Rechte mit dem Befehl ls -l anzeigen. Im Folgenden sind als Beispiel die Dateirechte des Verzeichnisses /var/mail/ dargestellt

```
 ls -l /var/mail/
drwxrwsr-x 2 root mail 4096 Apr 23  2012 /var/mail/
```

Für die Darstellung der Rechte sind die markierten Teile der Ausgabe relevant:

- Der erste Buchstabe (d) kennzeichnet den Dateityp.
- Danach folgen die Zugriffsrechte (rwxrwsr-x).
- Eigentümer der Datei
- Gruppe

Wie auch in anderen Betriebssystemen kann man verschiedene Rechte für Eigentümer (Owner) und Gruppe (Group) vergeben. Neben Eigentümer und Gruppe gibt es noch eine weitere, allgemeine Gruppe. Diese Gruppe nennt sich andere (Others).

Darstellungsarten

Neben der symbolischen Darstellung (z.B. rwxrwxr-x) gibt es auch noch eine oktale Darstellung (nach dem Oktalsystem). Die Grundrechte (Lesen, Schreiben, Ausführen) und Kombinationen daraus werden hierbei durch eine einzelne Ziffer repräsentiert und dem Eigentümer, der Gruppe und allen anderen zugeordnet. Je nach Anwendung wird dabei von unterschiedlichen Grundwerten ausgegangen und entweder Rechte gegeben oder entzogen. Bei chmod wird beispielsweise von der Grundeinstellung „keine Rechte“ (000) ausgegangen und Rechte gegeben, wohingegen bei umask von „alle Rechte

vorhanden“ (777) ausgegangen und Rechte entzogen werden. Entsprechend sind die Werte je nach Anwendung anders.

Mögliche Werte für:

Recht(e)	chmod (octal)	umask (octal)	Symbolisch	Binäre Entsprechung
Lesen, schreiben und ausführen	7	0	rwx	111
Lesen und Schreiben	6	1	rw-	110
Lesen und Ausführen	5	2	r-x	101
Nur lesen	4	3	r–	100
Schreiben und Ausführen	3	4	-wx	011
Nur Schreiben	2	5	-w-	010
Nur Ausführen	1	6	–x	001
Keine Rechte	0	7	—	000

Hier ein paar Beispiele:

- rwxrwxrwx entspricht 0777 (chmod) oder 0000 (umask): Jeder darf lesen, schreiben und ausführen.
- rwxr-xr-x entspricht 0755 (chmod) oder 0022 (umask): Jeder darf lesen und ausführen, aber nur der Dateibesitzer darf diese Datei (oder das Verzeichnis) auch verändern.

Die nachfolgenden Erklärungen beziehen sich vor allem auf Dateien vom Typ File (ohne Kennbuchstaben) und „Ordner“ (Directory, Kennbuchstabe d).

Nach dem Dateityp kommen drei Zeichengruppen zu je drei Zeichen. Diese kennzeichnen die Zugriffsrechte für die Datei bzw. das Verzeichnis. Hat der Benutzer/Gruppe/andere ein Recht, so wird der Buchstabe dafür angezeigt; ansonsten wird ein - dafür angezeigt.

In obigen Beispiel erscheint nach dem Dateityp dann die Zeichenfolge rwxrwsr-x. Wenn man diese in drei Dreiergruppen aufteilt, erhält man diese Gruppen:

- rwx: Rechte des Eigentümers
- rws: Rechte der Gruppe
- r-x: Recht von allen anderen (others)

Die folgende Tabelle erklärt die Bedeutung der einzelnen Buchstaben, Diese stehen immer in der gleichen Reihenfolge:

Symbole für Zugriffsrechte		
Zeichen	**Bedeutung**	**Beschreibung**
r	Lesen (read) Erlaubt lesenden Zugriff auf die Datei. Bei einem Verzeichnis können damit die Namen der enthaltenen Dateien und Ordner abgerufen werden (nicht jedoch deren weitere Daten wie z.B. Berechtigungen, Besitzer, Änderungszeitpunkt, Dateiinhalt etc.).	
w	Schreiben (write) Erlaubt schreibenden Zugriff auf eine Datei. Für ein Verzeichnis gesetzt, können Dateien oder Unterverzeichnisse angelegt oder gelöscht werden, sowie die Eigenschaften der enthaltenen Dateien/Verzeichnisse verändert werden.	

Symbole für Zugriffsrechte		
Zeichen	**Bedeutung**	**Beschreibung**
x	Ausführen (execute) Erlaubt das Ausführen einer Datei, wie das Starten eines Programms. Bei einem Verzeichnis ermöglicht dieses Recht, in diesen Ordner zu wechseln und weitere Attribute zu den enthaltenen Dateien abzurufen (sofern man die Dateinamen kennt ist dies unabhängig vom Leserecht auf diesen Ordner). Statt x kann auch ein Sonderrecht angeführt sein.	

Sonderrechte

Die oben gezeigten Dateirechte kann man als Basisrechte bezeichnen. Für besondere Anwendungen gibt es zusätzlich noch besondere Dateirechte. Der Einsatz dieser ist nur dann ratsam, wenn man genau weiß, was man tut, da dies unter Umständen zu Sicherheitsproblemen führen kann.

Sonderrechte		
Zeichen	**Bedeutung**	**Beschreibung**
s	Set-UID-Recht (SUID-Bit)	Das Set-UID-Recht („Set User ID“ bzw. „Setze Benutzerkennung“) sorgt bei einer Datei mit Ausführungsrechten dafür, dass dieses Programm immer mit den Rechten des Dateibesitzers läuft. Bei Ordnern ist dieses Bit ohne Bedeutung.
s (S)	Set-GID-Recht (SGID-Bit)	Das Set-GID-Recht („Set Group ID“ bzw. „Setze Gruppenkennung“) sorgt bei einer Datei mit Ausführungsrechten dafür, dass dieses Programm immer mit den Rechten der Dateigruppe läuft. Bei einem Ordner sorgt es dafür, dass die Gruppe an Unterordner und Dateien vererbt wird, die in diesem Ordner neu erstellt werden.
t (T)	Sticky-Bit	Das Sticky-Bit („Klebrig“) hat auf modernen Systemen nur noch eine einzige Funktion: Wird es auf einen Ordner angewandt, so können darin erstellte Dateien oder Verzeichnisse nur vom Dateibesitzer gelöscht oder umbenannt werden. Verwendet wird dies z.B. für /tmp.

Die Symbole für die Sonderrechte erscheinen an der dritten Stelle der Zugriffsrechte, die normalerweise dem Zeichen x (für executable) vorbehalten ist, und ersetzen ggf. dieses. Die Set-UID/GID-Rechte werden anstelle des x für den Besitzer bzw. die Gruppe angezeigt, das Sticky-Bit anstelle des x für andere. Wenn das entsprechende Ausführrecht gesetzt ist, wird ein Kleinbuchstabe verwendet, ansonsten ein Großbuchstabe.

INODES

Ein Inode (englisch index node, gesprochen „eye-node“) ist die grundlegende Datenstruktur zur Verwaltung von Dateisystemen mit unixartigen Betriebssystemen. Jeder Inode wird innerhalb einer Partition eindeutig durch seine Inode-Nummer identifiziert. Jeder Namenseintrag in einem Verzeichnis verweist auf genau einen Inode. Dieser enthält die Metadaten der Datei und verweist auf die Daten der Datei beziehungsweise die Dateiliste des Verzeichnisses.

Die Anwendungssoftware unterscheidet beim Lesen oder Schreiben von Daten nicht mehr zwischen Gerätetreibern und regulären Dateien. Durch das Inode-Konzept gilt bei den Unixvarianten alles als Datei („On UNIX systems it is reasonably safe to say that everything is a file: …“). Dadurch unterscheiden sich solche Betriebssysteme in der Verwaltung ihres Datenspeichers von anderen Systemen wie Microsoft Windows mit NTFS, aber auch von VMS oder MVS.

Grundsätzliches

Speichert man eine Datei auf einem Computer ab, so muss nicht nur der Dateiinhalt (Nutzdaten) gespeichert werden, sondern auch Metadaten, wie zum Beispiel der Zeitpunkt der Dateierstellung oder der Besitzer der Datei. Gleichzeitig muss der Computer einem Dateinamen – inklusive Dateipfad – die entsprechenden Nutzdaten und Metadaten effizient zuordnen können. Die Spezifikation, wie diese Daten organisiert und auf einem Datenträger gespeichert werden, nennt man Dateisystem. Dabei gibt es abhängig von Einsatzbereich und Betriebssystem unterschiedliche Dateisysteme. Umgesetzt wird die Dateisystemspezifikation von einem Treiber, der wahlweise als ein Kernel-Modul des Betriebssystemkern (Kernel) oder seltener als gewöhnliches Programm im Userspace umgesetzt sein kann.

Dateisysteme unixoider Betriebssysteme – wie Linux und macOS – verwenden sogenannte Inodes. Diese enthalten die Metadaten sowie Verweise darauf, wo Nutzdaten gespeichert sind. An einem speziellen Ort des Dateisystems, dem Superblock, wird wiederum die Größe, Anzahl und Lage der Inodes gespeichert. Die Inodes sind durchnummeriert und an einem Stück auf dem Datenträger gespeichert. Das Wurzelverzeichnis eines Dateisystems besitzt eine feste Inodenummer. Unterordner sind „gewöhnliche“ Dateien, welche eine Liste der darin enthaltenen Dateien mit der Zuordnung der dazugehörenden Inodenummern als Nutzdaten enthalten.

Soll also beispielsweise die Datei /bin/ls geöffnet werden, so läuft dies, vereinfacht, wie folgt ab:

- Der Dateisystemtreiber liest den Superblock aus, dadurch erfährt er die Startposition der Inodes und deren Länge, somit kann nun jeder beliebige Inode gefunden und gelesen werden.
- Nun wird der Inode des Wurzelverzeichnisses geöffnet. Da dies ein Ordner ist, befindet sich darin ein Verweis auf die Speicherstelle der Liste aller darin enthaltenen Dateien mitsamt ihren Inodenummern. Darin wird das Verzeichnis bin gesucht.
- Nun kann der Inode des bin-Verzeichnisses gelesen werden und analog zum letzten Schritt der

Inode der Datei ls gefunden werden.
- Da es sich bei der Datei ls nicht um ein Verzeichnis, sondern um eine reguläre Datei handelt, enthält ihr Inode nun einen Verweis auf die Speicherstelle der gewünschten Daten.

Aufbau

Jedem einzelnen von einem Schrägstrich / (slash) begrenzten Namen ist genau ein Inode zugeordnet. Dieser speichert folgende Metainformationen zur Datei, aber nicht den eigentlichen Namen:

- Die Art der Datei (reguläre Datei, Verzeichnis, Symbolischer Link, …), siehe unten;
- die nummerische Kennung des Eigentümers (UID, user id) und der Gruppe (GID, group id);
- die Zugriffsrechte für den Eigentümer (user), die Gruppe (group) und alle anderen (others);
- Die klassische Benutzer- und Rechteverwaltung geschieht mit den Programmen chown (change owner), chgrp (change group) und chmod (change mode). Durch Access Control Lists (ACL) wird eine feinere Rechtevergabe ermöglicht.
- verschiedene Zeitpunkte der Datei: Erstellung, Zugriff (access time, atime) und letzte Änderung (modification time, mtime);
- die Zeit der letzten Status-Änderung des Inodes (status, ctime);
- die Größe der Datei;
- den Linkzähler (siehe unten);
- einen oder mehrere Verweise auf die Blöcke, in denen die eigentlichen Daten gespeichert sind.

Reguläre Dateien

Reguläre Dateien (engl. regular files) sind sowohl Anwenderdaten als auch ausführbare Programme. Letztere sind durch das executable-Recht gekennzeichnet und werden beim Aufruf durch das System in einem eigenen Prozess gestartet. Als „ausführbar“ gelten nicht nur kompilierte Programme, sondern auch Skripte, bei denen der Shebang den zu verwendenden Interpreter angibt. Bei „dünnbesetzten Dateien“, sogenannten sparse files, unterscheidet sich die logische Größe vom durch die Datenblöcke tatsächlich belegten Festplattenplatz.

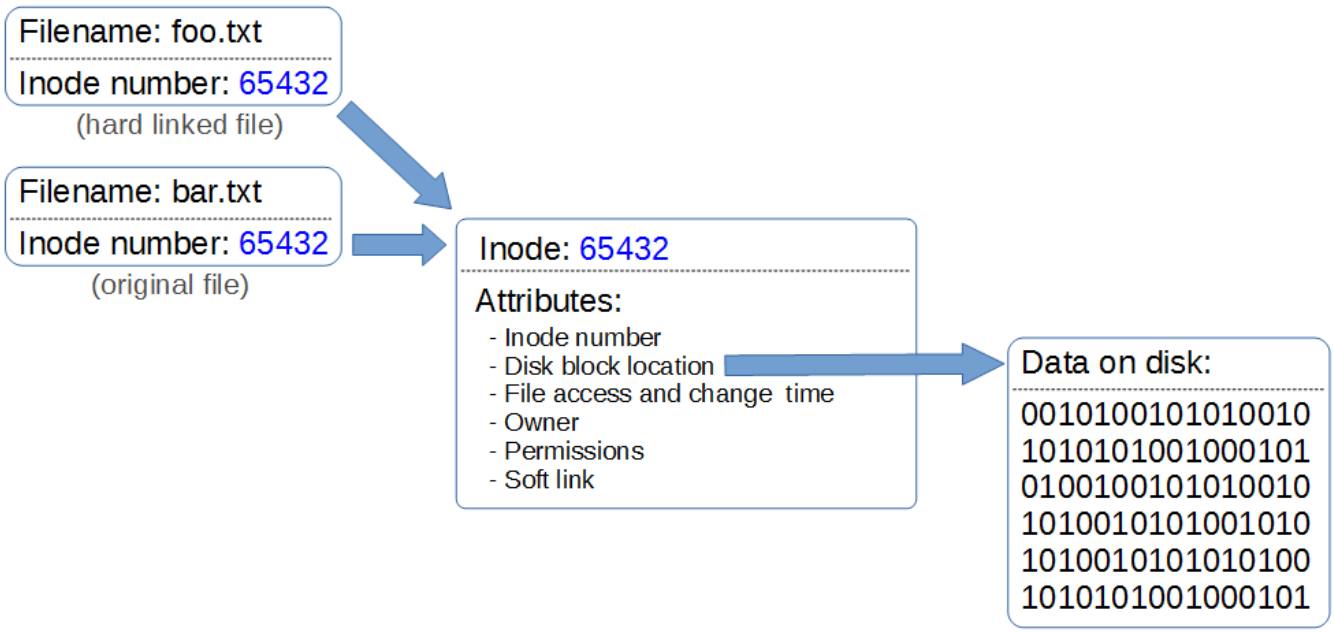
Verzeichnisse

Verzeichnisse sind Dateien, deren „Dateiinhalt“ aus einer tabellarischen Liste der darin enthaltenen Dateien besteht. Die Tabelle enthält dabei eine Spalte mit den Dateinamen und eine Spalte mit den zugehörigen Inodenummern. Bei manchen Dateisystemen umfasst die Tabelle noch weitere Informationen, so speichert ext4 darin auch den Dateityp aller enthaltenen Dateien ab, so dass dieser beim Auflisten eines Verzeichnisinhalts nicht aus den Inodes aller Dateien ausgelesen werden muss. Für jedes Verzeichnis existieren immer die Einträge . und .. als Verweise auf das aktuelle bzw. übergeordnete Verzeichnis.

Harte Links (hard links)

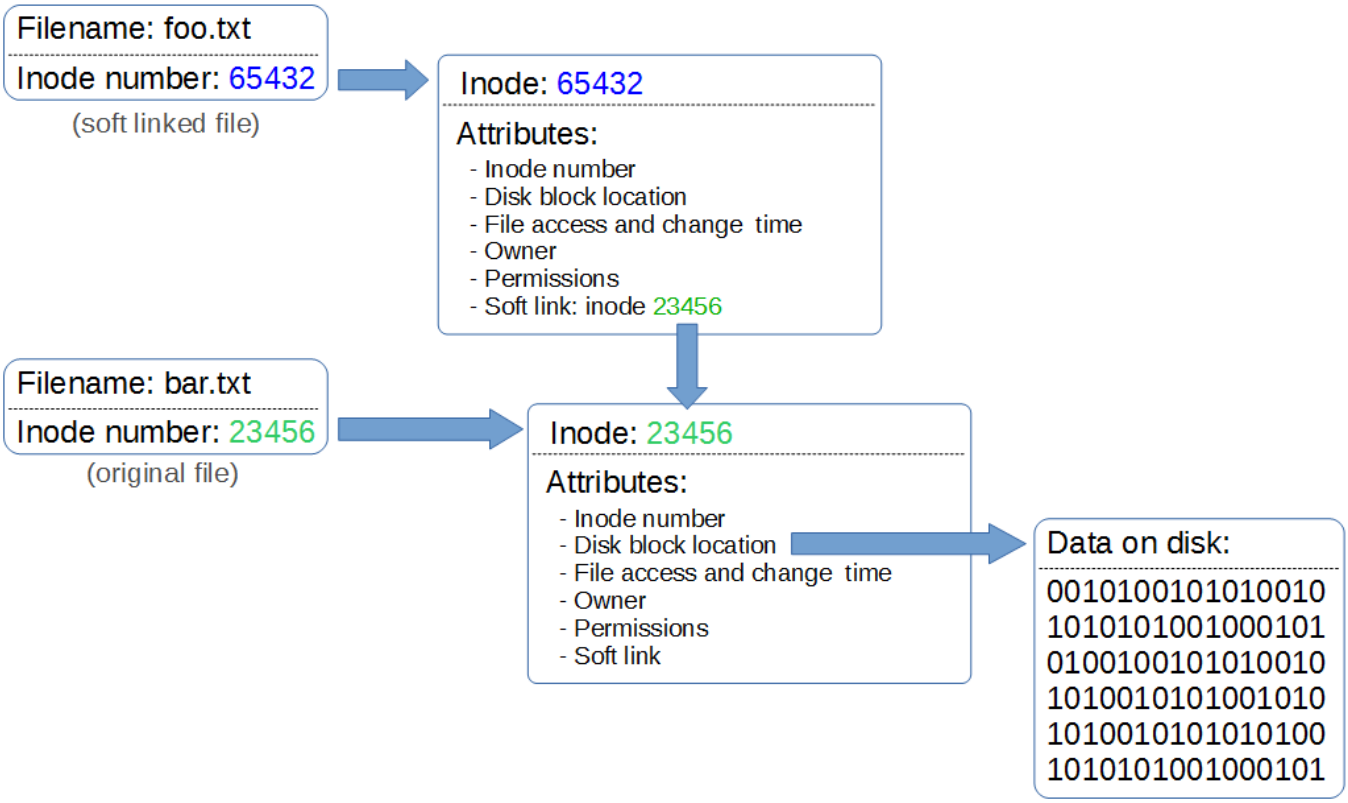
Bei Harten Links hingegen handelt es sich nicht um spezielle Dateien. Von einem Harten Link spricht man, wenn auf einen Inode mehrfach von verschiedenen Ordnern oder verschiedenen Dateinamen verwiesen wird. Alle Verweise auf den Inode sind gleichwertig, es gibt also kein Original. Im Inode gibt

der Linkzähler an, wie viele Dateiennamen auf diesen verweisen, er steht nach dem Anlegen einer Datei also bei 1 und wird erhöht, sobald weitere Hardlinks für diese Datei erstellt werden. Bei einem Verzeichnis beträgt er zwei mehr, als Unterordner darin enthalten sind, da neben dem Eintrag im Ordner darüber und dem Eintrag '.' im Ordner noch die Einträge '..' in allen Unterordnern darauf verweisen. Wird eine Datei gelöscht, so wird ihr Eintrag aus dem übergeordneten Verzeichnis entfernt und der Linkzähler um eins reduziert. Beträgt der Linkzähler dann 0, wird gegebenenfalls abgewartet, bis die Datei von keinem Programm mehr geöffnet ist, und erst anschließend der Speicherplatz freigegeben.



Symbolische Links (symbolic links, soft links, symlinks)

Bei symbolischen Links handelt es sich um spezielle Dateien, die anstelle von Daten einen Dateipfad enthalten, auf den der Link verweist. Je nach Dateisystem und Länge des Dateipfads wird der Link entweder direkt im Inode gespeichert oder in einem Datenblock, auf welchen der Inode verweist.



Praxis

Die Inodenummer einer Datei lässt sich mittels des Befehls ls -li Dateiname anzeigen

LINUX-DISTRIBUTIONEN

Wie schon besprochen besteht ein Linux-Betriebssystem aus dem Linux-Kernel und einer großen Anzahl verschiedener Anwenderprogramme. Tatsächlich gibt es ein Projekt, das eine Anleitung bietet, wie man aus den Kernelquellen und selbst selektierten Programmen ein komplettes, maßgeschneidertes Betriebssystem bauen kann. Das Projekt nennt sich „Linux From Scratch“ oder „LFS“ (www.linuxfromscratch.org) und ich kann jedem, der etwas Zeit übrig hat, nur empfehlen, dies selbst einmal zu probieren. Man erhält eine ultrakompaktes, ultraschnelles Betriebssystem und kann sagen, „das ist mein eigenes reinrassiges Linux-System“. Aber was kommt dann?

Man braucht schon einige Zeit bis alle Programme, die so benötigt werden, kompiliert und konfiguriert sind und dann müssen diese auch noch laufend aktualisiert werden. Sicherheitsupdates müssen selbst organisiert und kompiliert werden. Mit all der Administrationsarbeit kommt man zu sonst nichts mehr.

Um das zu vermeiden, gibt es Linux-Distributionen. Diese bieten nicht nur einen fertigen Satz von notwendigen Programmen, sondern auch die regelmäßige Versorgung mit Updates an. Die aus meiner Sicht wichtigsten Distributionen sollen hier aufgeführt werden:

Debian

„Debian“ (www.debian.org) ist eine nicht-kommerzielle Distribution und das „Debian-Projekt“ ist nach der „Debian-Verfassung“ geregelt, die eine demokratische Organisationsstruktur vorsieht. Darüberhinaus ist das Projekt über den „Debian Social Contract“ zu völlig freier Software verpflichtet. Mit einigen 1000 Mitarbeitern ist Debian der Gigant unter den Linux-Systemen.

Da bei Debian eine „stabile Version“ immer eine wirklich stabile Version ist, sind die Entwicklungszeiten relativ lang und böse Zungen behaupten auch, dass die stabile Version schon bei Erscheinen veraltet ist.

Allerdings bietet Debian auch immer schon die zukünftigen Versionen an und so gibt es mehrere Zweige, aus denen man sich bedienen kann:

stable - wirklich stabile Version, die auch für den kommerziellen Serverbetrieb geeignet ist!

testing - die zukünftige stable-Version. Ab einem gewissen Entwicklungsstand wird die Distribution „eingefroren“ (engl. „frozen“) - d.h. es werden keine neueren Versionen von Programmen mehr aufgenommen, sondern nur noch an der Fehlerbeseitigung bei den vorhandenen gearbeitet. Dies entspricht ungefähr dem Zustand, bei dem andere Distributoren ihre „stabilen“ Versionen veröffentlichen. Da ich schon mehrmals eine testing-Version ab dem Anfangsstadium benutzt habe, glaube ich sogar sagen zu können, dass testing nie so instabil ist, wie manche andere Distribution im „ausgereiften“ Zustand. Für den Desktopbetrieb kann ich ein eingefrorenes testing jedenfalls empfehlen.

unstable - ist der erste Anlaufpunkt für neue Versionen von Paketen und Programmen, bevor sie in testing integriert werden. Man installiert sich mit unstable das neueste vom neuen, muss aber wissen,

dass das nicht immer stabil ist.

experimental - ist kein vollständiger Zweig, denn es dient nur dazu, Programme und deren Funktionen zu testen, die sonst das ganze System gefährden würden. Es enthält immer nur die gerade getesteten, bzw. die von diesen benötigten Programmpakete.

Diese Zweige haben auch immer Codenamen und sind, einer Vorliebe der frühen Entwickler folgend, immer nach Figuren aus dem Film „Toy Story“ benannt. So heißt im Moment die stable-Version „Jessie“ und „Stretch“ ist testing. unstable ist immer „Sid“, der Junge von nebenan, der die Spielsachen zerstört, aber es lässt sich auch als Abkürzung für „still in development“ (noch in Entwicklung) deuten.

Und wer einen dieser Zweige installiert hat, kann auf eine unüberschaubare Vielzahl an Programmen zurückgreifen, auf Wunsch (und auf eigene Gefahr) auch aus den anderen Zweigen. Für Anfänger ist es wohl nur bedingt zu empfehlen, obwohl sich in den letzten Jahren sehr viel in Sachen Benutzerführung getan hat. Auch steht ein deutschsprachiges Forum (debianforum.de) zur Verfügung, wo man Hilfe bekommt und wo auch dumme Fragen gestellt werden dürfen. Für ambitionierte Linux-EinsteigerInnen, die sich auch mit den Möglichkeiten ihres Betriebssystems auseinandersetzen wollen, könnte es sogar die beste Distribution sein.

Fedora

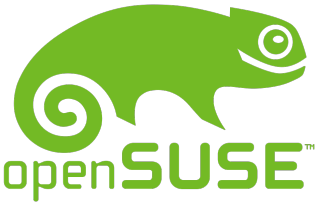
Das nicht-kommerzielle „Fedora“ (fedoraproject.org) ist der Nachfolger des traditionsreichen, kommerziellen „Red Hat Linux“, welches nicht mehr selbständig weiterentwickelt wird. Statt dessen verkauft die Firma Red Hat, das auf Fedora basierende „Red Hat Enterprise Linux“.

Fedora ist eine sehr innovative Distribution und vor allem in den USA sehr beliebt. Es werden nur völlig unter freier Lizenz stehende Inhalte akzeptiert, weshalb nach der Installation zum Beispiel keine MP3-unterstützenden Programme zu finden sind. Für Anfänger gibt es bessere Distributionen.

Gentoo Linux

Das nicht-kommerzielle „Gentoo Linux“ (www.gentoo.de) ist eine quellbasierte Linux-Metadistribution - das heißt, alle Programme, inklusive des Kernels, werden selbst kompiliert. Das klingt sehr anstrengend, ist es aber gar nicht so, da die Distribution geeignete Werkzeuge zur Verfügung stellt, mit denen dies einfachst möglich gelingt. Auch sorgt eine große, sehr aktive „Community“ bei jedem Problem für Rat und Hilfe. Dennoch ist sie für Linux-Neulinge wohl nicht empfehlenswert.

OpenSUSE



„openSUSE“ (www.opensuse.org) ist die zweitbeliebteste Distribution am Heim-PC. Die nicht-kommerzielle Variante der von Novell aufgekauften kommerziellen SUSE-Distribution (heute „SUSE Linux Enterprise“) glänzt mit einem universellen Konfigurationswerkzeug. Sie gilt als anfängerfreundlich. Die neueste Versionsnummer 42.1 bezieht sich übrigens auf die Antwort auf die Frage „nach dem Leben, dem Universum und dem ganzen Rest“ aus Douglas Adam's „Per Anhalter durch die Galaxis“. Schon 1996 hatte die Version 4.2 diesen Bezug.

Slackware

„Slackware“ (www.slackware.com) ist die älteste noch heute existierende Distribution. Sie verzichtet aus Prinzip auf grafische Einrichtungswerkzeuge und ist daher eher nur für fortgeschrittene BenutzerInnen geeignet.

Bisher nicht vorgekommen sind Distributionen, die nur Abwandlungen anderer Distributionen sind und häufig auch deren Quellen benutzen. Vor allem von Debian gibt es unzählige davon. Sie werden als „Derivate“, oder oft auch, etwas abfällig, als „Klone“ bezeichnet. Ein solcher „Debian-Klon“ hat allerdings Geschichte geschrieben:

Ubuntu

Das von der Firma des Gründers gesponserte kostenlose Betriebssystem soll nach dem Willen der Entwickler ein einfach zu installierendes und leicht zu bedienendes Betriebssystem mit aufeinander abgestimmter Software sein. Es bedient sich dazu aus den Quellen von Debian unstable und hat das Ziel, nach der enormen Popularität zu schließen, eindeutig geschafft.

Tatsächlich kann Ubuntu von der Live-CD mit wenigen Mausklicks problemlos auf die Festplatte installiert werden und dann erwartet den Benutzer ein weitgehend komplettes Betriebssystem mit vielen Multimedia-Programmen. Releases erscheinen mit schöner halbjährlicher Regelmäßigkeit und der Upgrade auf diese lässt sich ebenfalls auf Mausklick bewerkstelligen. Alle 2 Jahre gibt es eine „Versionen mit verlägerter Unterstützung“ (Long Term Support oder kurz LTS), die dann deutlich stabiler als die kürzer unterstützten ist. Für EinsteigerInnen ist Ubuntu (www.ubuntu.com) bestens geeignet.

Linux Mint

Diese Distribution war ursprünglich ein Ubuntu-Klon, aber neuerdings ist Linux Mint auch als LMDE - Linux Mint Debian Edition - erhältlich. Den Entwicklern ist es wichtig , die bestmögliche Integration von Programmen zu bieten, die bei Benutzern beliebt, aber eben nicht quelloffene freie Software sind. Die anderen Distribution, inklusive Ubuntu, bieten zwar auch die Installation von „non-free“-Paketen an, aber in einem eigenen Zweig und erst nach der Basisinstallation.

Für absolute Stabilität setzt Linux Mint immer auf LTS (Ubuntu) oder stable (Debian) Versionen auf, aber die integrierten Programme erhalten auch zwischenzeitig Versionsupgrades. Als Vorbild wird die

Benutzerfreundlichkeit und Stabilität von Apple's OS X genannt. Auch Linux Mint (www.linuxmint.com) ist für EinsteigerInnen bestens geeignet.

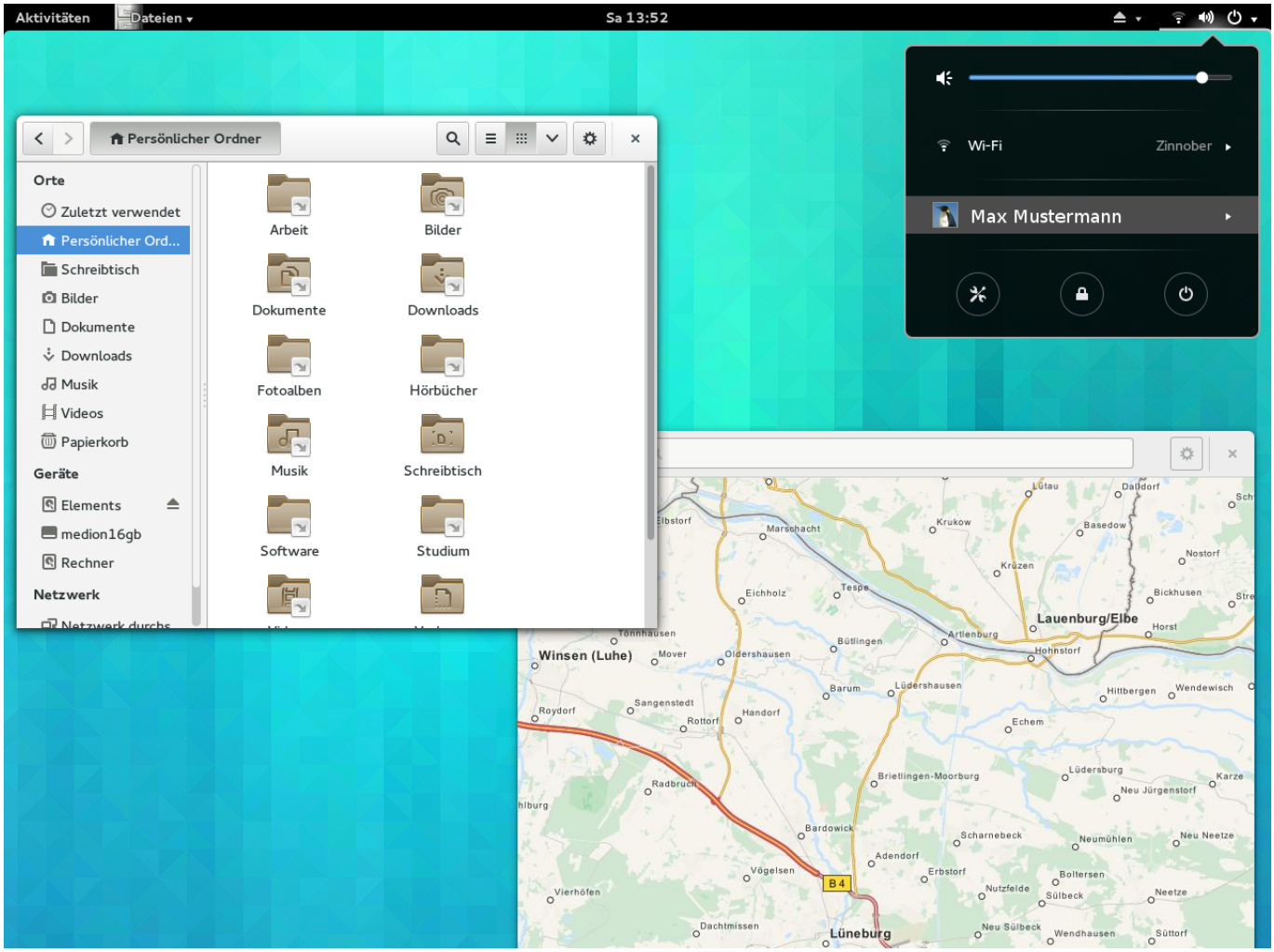
GRAFISCHE OBERFLÄCHEN (GUIs, Desktops)

Anders als bei Microsoft und Apple gibt es bei Linux-Distributionen keinen festgelegten Desktop. Alle Distributionen bieten die Möglichkeit den Desktop zu ändern und zunehmend kann schon bei der Installation der gewünschte Desktop festgelegt werden. Wenngleich für Anfänger im allgemeinen der Standard-Desktop der jeweiligen Distribution sicher eine gute Wahl ist, möchte ich abschließend auch noch kurz einige Desktops vorstellen. Gnome und KDE sind die Platzhirsche auf Linux-Bildschirmen.

Gnome

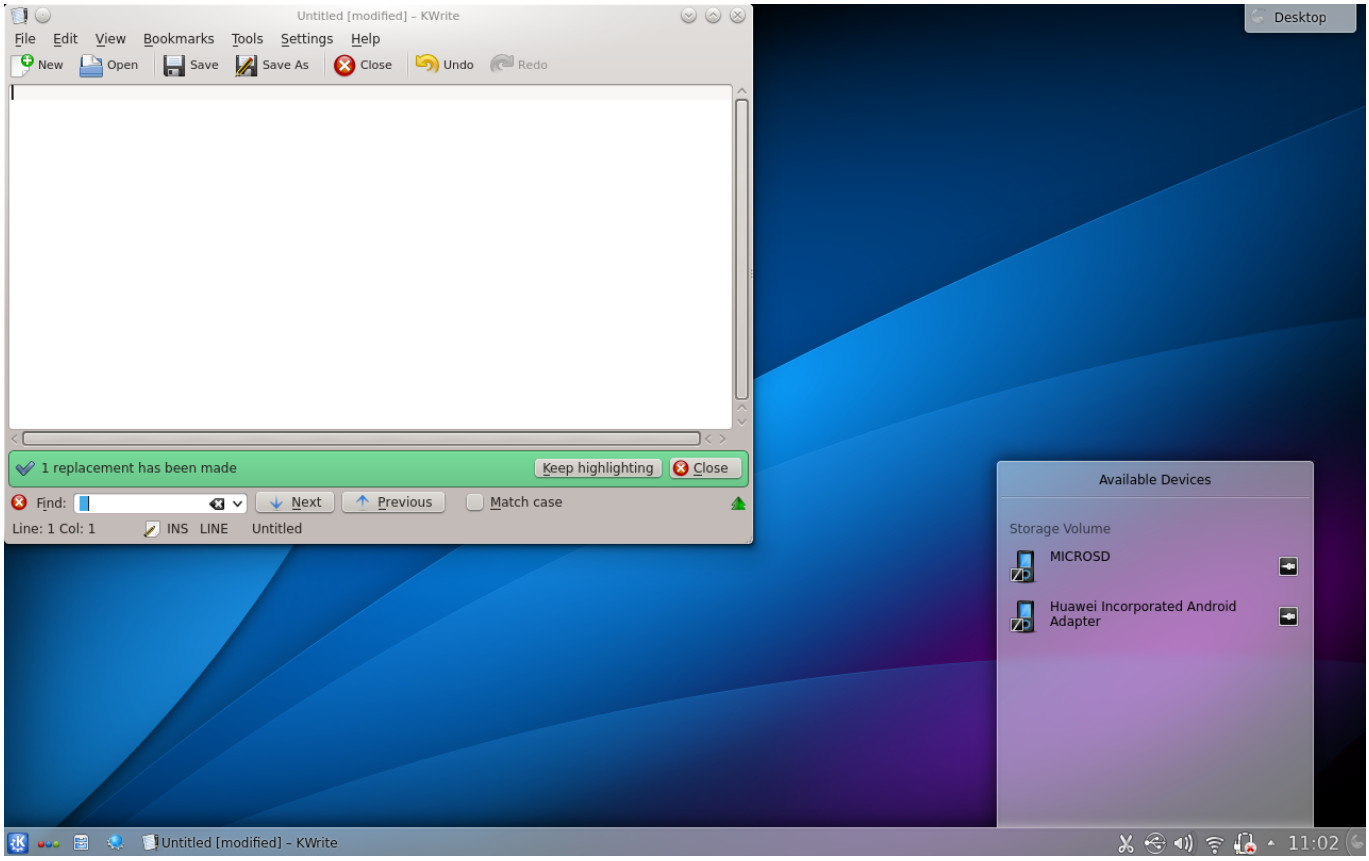
Gnome (Eigenschreibweise GNOME) [gnoʊm][3] ist eine Desktop-Umgebung für Unix- und Unix-ähnliche Systeme mit einer grafischen Benutzeroberfläche und einer Sammlung von Programmen für den täglichen Gebrauch. Gnome wird unter den freien Lizenzen GPL und LGPL veröffentlicht und ist Teil des GNU-Projekts.

Gnome ist unter anderem der Standard-Desktop von Fedora und Ubuntu. Einige Komponenten von Gnome wurden nach Windows und MacOS portiert, etwa Evolution oder GStreamer, werden jedoch teilweise, wie im Falle von Evolution, nicht mehr länger gepflegt.



KDE

KDE ist eine Community, die sich der Entwicklung freier Software verschrieben hat. Eines der bekanntesten Projekte ist die Desktop-Umgebung KDE Plasma 5 (früher K Desktop Environment, abgekürzt KDE).

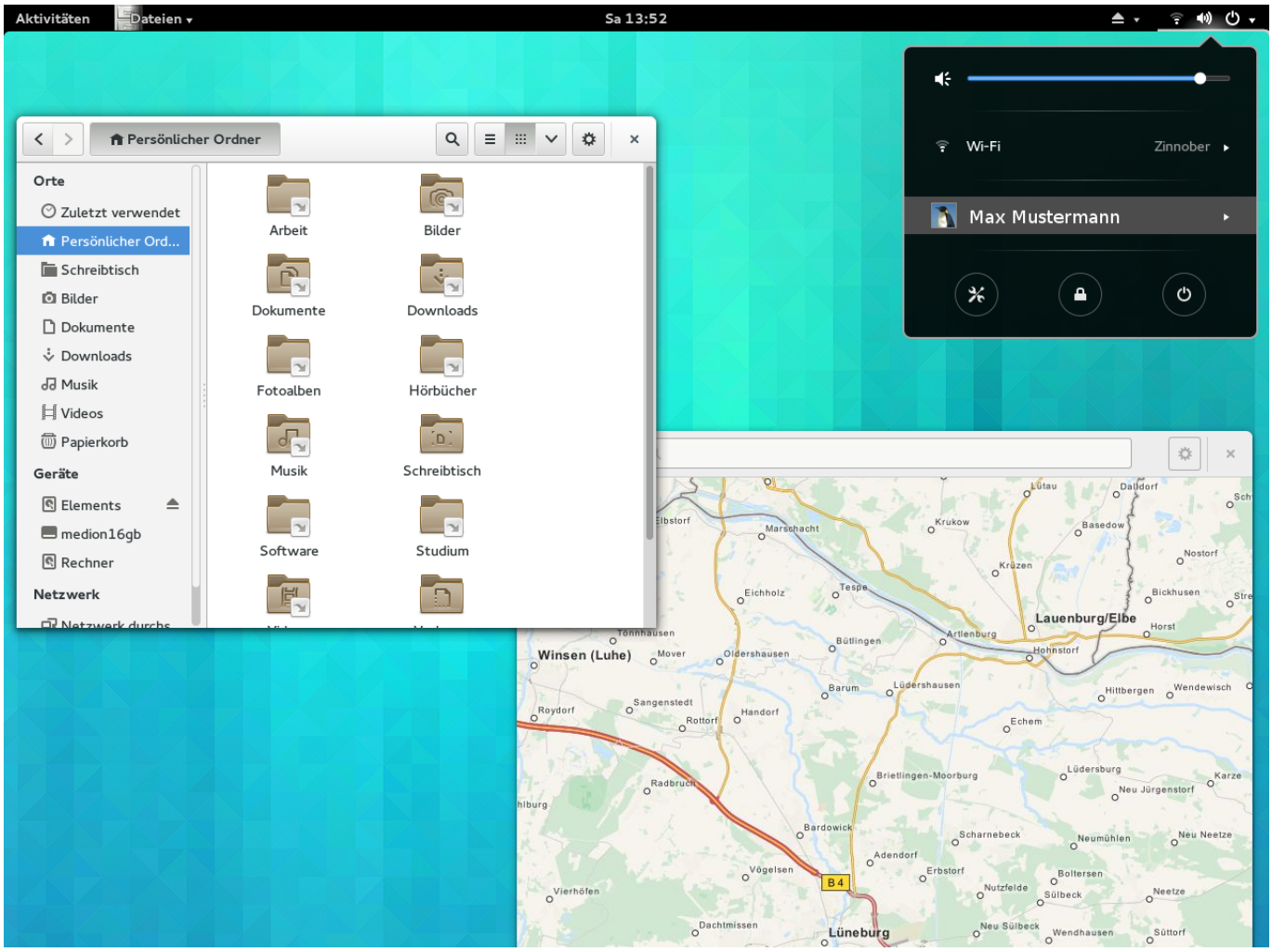


Unity

Unity ist eine durch das Unternehmen Canonical entwickelte Desktop-Umgebung für Linux-Betriebssysteme, die besonders sparsam mit Bildschirmplatz umgehen soll.

XFCE

Während erstgenannte Desktops mit Features protzen gehen die EntwicklerInnen bei XFCE einen anderen Weg. XFCE möchte einen schlanken, performanten Desktop bieten, der auf „unnötige Spielereien“ verzichtet und auch auf leistungsschwachen und zum Teil für andere Aufgaben (Multimedia) bestimmten Systemen resourcenschonend läuft. Viele heutige Distributionen bieten XFCE als vorkonfigurierte Alternative zum Standarddesktop an.



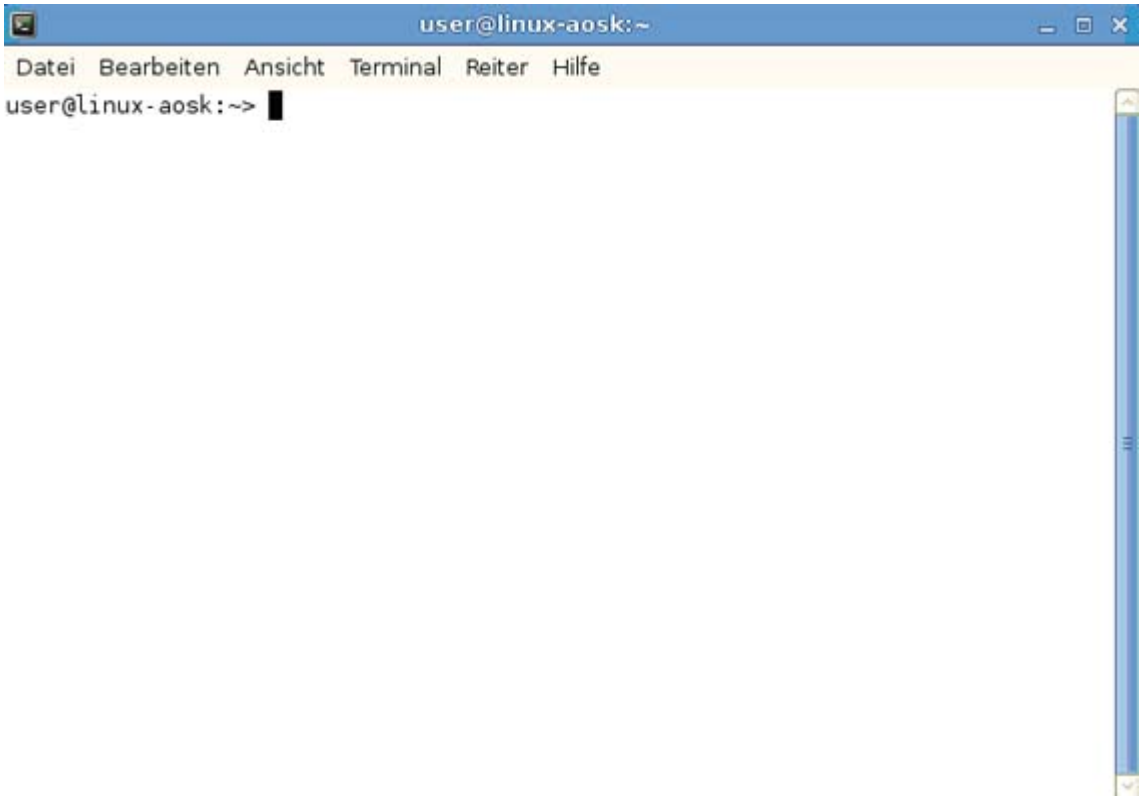
MATE und Cinnamon

Der umstrittene Upgrade von Gnome2 auf Gnome3 führte zur Geburt von MATE. Dieser Desktop ist eine direkte Fortentwicklung von Gnome2. Cinnamon dagegen ist aus den Quellen von Gnome3 entstanden, aber deutlich performanter als dieser. Was beide gemeinsam haben - sie sind die Standarddesktops von „Linux Mint“ und nur als Ubuntu-Editionen von Linux Mint gibt es auch KDE und XFCE vorkonfiguriert.

Es können auch mehrere Desktops gleichzeitig installiert werden. Bei der Anmeldung kann dann zwischen den Desktops gewechselt werden. So kann jeder seinen Lieblingsdesktop herausfinden.

Arbeiten auf der Konsole

Gnome Terminal findet man unter *Weiter Anwendungen* .



prompt *user@linux-aosk:~→* hat folgende Bedeutung:

- *user* : Benutzername
- *linux-aosk* : Rechnername
- ~ : Homeverzeichnis

Kommandos zur Bearbeitung von Dateien

Obwohl unter KDE und Gnome moderne Dateimanager zur Verfügung stehen, verwenden erfahrene Linux-Anwender oft noch immer diverse, text-orientierte Kommandos.

Kommando	**Beschreibung**	**Kommando in DOS**
Hilfe		
man *Befehl*	Hilfe zum Kommando	
Befehl - - help	Hilfe zum Kommando	
Als Root		
su	wechselt als Root (Passwort eingeben)	
sudo	einen Befehl als Root ausführen	
Verzeichnisbaum		
cd	wechselt das aktuelle Verzeichnis	
cd /	wechselt ins root-Verzeichnis	

Kommando	Beschreibung	Kommando in DOS
Hilfe		
ls	zeigt alle Dateien des aktuellen Verzeichnisse an	dir
ls -l	zeigt eine detaillierte Liste	
ls -a	zeigt versteckte Dateien an	
mkdir	erzeugt ein neues Verzeichnis	md
rmdir	löscht Verzeichnisse	rd
pwd	zeigt aktuellen Pfad an	
Joker		
*	steht für eine beliebige Anzahl von beliebigen Zeichen	
?	steht für ein beliebiges Zeichen	
Dateien		
mv quelle ziel	verschiebt Dateien bzw. ändert ihren Namen	move
cp quelle ziel	kopiert Dateien	copy
cp ordner ziel -r	kopiert gesamten ordner inkl. aller unterordner an ziel	
cat	zeigt Dateiinhalt an	type
less	öffnet Anzeigeprogramm	
more	zeigt Dateiinhalt seitenweise an	
touch Dateiname	erstellt leere Datei	
mcedit Dateiname	öffnet Datei in einem Editor zur Bearbeitung	edit
vim Dateiname	öffnet Datei mit dem Editor VIM zur Bearbeitung	
vimtutor	Tutorial zum Erlernen vom Editor VIM	
rm	löscht Dateien	del
rm unterordner -r	löscht gesamten Unterordner inkl. aller Dateien	
find -name dateinamen	sucht Dateien nach Namen	
Packen und Komprimieren von Verzeichnissen und Dateien		
tar	vereint mehrere Dateien (und Verzeichnisse) in einer Datei	
tar -t	Inhalt eines Archivs anzeigen	
tar -x	Dateien aus Archiv holen	
tar -c	neues Archiv erzugen	
tar -f	um Namen des Archiv anzugeben	
tar -xvjf	entzippen	

Weitere Befehle

- http://www.admintalk.de/konsolenbefehle.php
- http://www.shellbefehle.de/befehle/

Übung

1. Erstelle in deinem Home-Directory einen Ordner uebungen.
2. Speichere das File uebung1.tar in diesen Ordner!
3. Entpacke das Archiv mit Hilfe von `tar -xvjf uebung1.tar`. Welche Verzeicnisse und Dateien befinden sich nun in deinem Home-Directory?
4. Gib den Befehl `./hallo` ein.

5. Finde die Datei `ichbinhier`.
6. Wechsle in das Verzeichnis, in dem sich die Datei befindet.
7. Erstelle ein Verzeichnis mit dem Namen `backup` in deinem Homedirectory.
8. Kopiere das gesamte Verzeichnis `uebung1` in das Verzeichnis `backup`.
9. Lösche das Verzeichnis `uebung1`.
10. Erstelle ein Verzeichnis mit dem Namen `aufgabe` und wechsle hinein.
11. Erstelle drei leere Dateien `datei1` bis `datei3`.
12. Öffne mit einem Editor `datei1` und gib drei Zeilen Text ein. Speicher ab!
13. Lasse dir die Datei mit einem entsprechendem Kommando ausgeben!
14. Gib den Befehl `tac datei1` ein. Was passiert?
15. Wechsle in die grafische Oberfläche!
16. Orientiere dich an der Oberfläche!
17. Versuche den Bildschirmhintergrund umzustellen.
18. Öffne ein Konsolenfenster. Lösche darin den gesamten Ordner `uebungen` inklusive Unterverzeichnis.

Shell Scripts sh (Bourne Shell)

Shell-Scripts (Kommandoprozeduren) sind unter Unix das Analogon zu Batch-Dateien (Stapeldateien) von MS-DOS, sie sind jedoch wesentlich leistungsfähiger. Ihre Syntax hängt allerdings von der verwendeten „Shell“ ab. In diesem Artikel werden Bourne Shell (sh) Scripts betrachtet. Sie gelten im allgemeinen als zuverlässiger als csh-Scripts.

Ein Shell-Script ist eine Textdatei, in der Kommandos gespeichert sind. Es stellt selbst ein Kommando dar und kann wie ein Systemkommando auch mit Parametern aufgerufen werden. Shell-Scripts dienen der Arbeitserleichterung und (nach ausreichenden Tests) der Erhöhung der Zuverlässigkeit, da sie gestatten, häufig gebrauchte Sequenzen von Kommandos zusammenzufassen.

Aufruf

Bourne Shell Scripts lassen sich generell wie folgt aufrufen:

```
sh myscript
```

Im allgemeinen ist es aber praktischer, die Datei als ausführbar anzumelden (execute permission), mittels

```
chmod +x myscript
```

Das Shell-Script läßt sich dann wie ein „normales“ (binäres) Kommando aufrufen:

```
myscript
```

Vorausgesetzt ist hierbei allerdings, daß das Betriebssystem das Script mit der richtigen Shell abarbeitet. Moderne Unix-Systeme prüfen zu diesem Zweck die erste Zeile. Für die sh sollte sie folgenden Inhalt haben:

```
#!/bin/sh
```

Obwohl das Doppelkreuz normalerweise einen Kommentar einleitet, der vom System nicht weiter beachtet wird, erkennt es hier, daß die „sh“ (mit absoluter Pfadangabe: /bin/sh) eingesetzt werden soll.

Ein Beispiel

```
#!/bin/sh
# Einfaches Beispiel
echo Hallo, Welt!
echo Datum, Uhrzeit und Arbeitsverzeichnis:
date
pwd
```

```
echo Uebergabe-Parameter: $*
```

Das vorstehende einfache Script enthält im wesentlichen normale Unix-Kommandos. Abgesehen von der ersten Zeile liegt die einzige Besonderheit im Platzhalter „$*“, der für alle Kommandozeilen-Parameter steht.

Testen eines Shell-Scripts

```
sh -n myscript
```

Syntax-Test (die Kommandos werden gelesen und geprüft, aber nicht ausgeführt)

```
sh -v myscript
```

Ausgabe der Shell-Kommandos in der gelesenen Form

```
sh -x myscript
```

Ausgabe der Shell-Kommandos nach Durchführung aller Ersetzungen, also in der Form, wie sie ausgeführt werden

Kommandozeilen-Parameter

```
$0
```

Name der Kommandoprozedur, die gerade ausgeführt wird

```
$#
```

Anzahl der Parameter

```
$1
```

erster Parameter

```
$2
```

zweiter Parameter

```
$3
```

dritter ... Parameter

```
$*
```

steht für alle Kommandozeilen-Parameter ($1 $2 $3 ...)

```
$@
```

wie $* ($1 $2 $3 ...)

```
$$
```

Prozeßnummer der Shell (nützlich, um eindeutige Namen für temporäre Dateien zu vergeben)

```
$-
```

steht für die aktuellen Shell-Optionen

```
$?
```

gibt den Return-Code des zuletzt ausgeführten Kommandos an (0 bei erfolgreicher Ausführung)

```
$!
```

Prozessnummer des zuletzt ausgeführten Hintergrund-Prozesses

Beispiel

```
#!/bin/sh
# Variablen
echo Uebergabeparameter: $*
echo user ist: $USER
echo shell ist: $SHELL
echo Parameter 1 ist: $1
echo Prozedurname ist: $0
echo Prozessnummer ist: $$
echo Anzahl der Parameter ist: $#
a=17.89        # ohne Luecken am = Zeichen
echo a ist $a
```

Prozesssteuerung

Bedingte Ausführung: if

```
if [ bedingung ]
  then kommandos1
  else kommandos2
fi
```

Anmerkungen: „fi“ ist ein rückwärts geschriebenes „if“, es bedeutet „end if“ (diese Schreibweise ist eine besondere Eigenheit der Bourne Shell). Die „bedingung“ entspricht der Syntax von test, siehe auch weiter unten. Im if-Konstrukt kann der „else“-Zweig entfallen, andererseits ist eine Erweiterung

durch einen oder mehrere „else if“-Zweige möglich, die hier „elif“ heißen:

```
if [ bedingung1 ]
    then kommandos1
  elif [ bedingung2 ]
    then kommandos2
  else kommandos3
fi
```

Die Formulierung

```
if [ bedingung ]
```

ist äquivalent zu

```
if test bedingung
```

Alternativ ist es möglich, den Erfolg eines Kommandos zu prüfen:

```
if kommando
```

(beispielsweise liefert das Kommando „true“ stets „wahr“, das Kommando „false“ hingegen „unwahr“)

Wichtige Vergleichsoperationen (test)

Hinweis: Es ist unbedingt notwendig, daß alle Operatoren von Leerzeichen umgeben sind, sonst werden sie von der Shell nicht erkannt! (Das gilt auch für die Klammern.)

Zeichenketten

```
"s1" = "s2"
```

wahr, wenn die Zeichenketten gleich sind

```
"s1" != "s2"
```

wahr, wenn die Zeichenketten ungleich sind

```
-z "s1"
```

wahr, wenn die Zeichenkette leer ist (Länge gleich Null)

```
-n "s1"
```

wahr, wenn die Zeichenkette nicht leer ist (Länge größer als Null)

(Ganze) Zahlen

```
n1 -eq n2
```

wahr, wenn die Zahlen gleich sind

```
n1 -ne n2
```

wahr, wenn die Zahlen ungleich sind

```
n1 -gt n2
```

wahr, wenn die Zahl n1 größer ist als n2

```
n1 -ge n2
```

wahr, wenn die Zahl n1 größer oder gleich n2 ist

```
n1 -lt n2
```

wahr, wenn die Zahl n1 kleiner ist als n2

```
n1 -le n2
```

wahr, wenn die Zahl n1 kleiner oder gleich n2 ist

Sonstiges

```
!
```

Negation

```
-a
```

logisches „und“

```
-o
```

logisches „oder“ (nichtexklusiv; -a hat eine höhere Priorität)

```
\( ... \)
```

Runde Klammern dienen zur Gruppierung. Man beachte, daß sie durch einen vorangestellten Backslash, \, geschützt werden müssen.

```
-f filename
```

wahr, wenn die Datei existiert. (Weitere Optionen findet man in der man page zu test)

Beispiel:

```
#!/bin/sh
# Interaktive Eingabe, if-Abfrage
echo Hallo, user, alles in Ordnung?
echo Ihre Antwort, n/j:
read answer
echo Ihre Antwort war: $answer
# if [ "$answer" = "j" ]
if [ "$answer" != "n" ]
  then echo ja
  else echo nein
fi
```

Mehrfachentscheidung: case

```
case var in
  muster1) kommandos1 ;;
  muster2) kommandos2 ;;
  *) default-kommandos ;;
esac
```

Anmerkungen: „esac“ ist ein rückwärts geschriebenes „case“, es bedeutet „end case“. Die einzelnen Fälle werden durch die Angabe eines Musters festgelegt, „muster)“, und durch ein doppeltes Semikolon abgeschlossen. (Ein einfaches Semikolon dient als Trennzeichen für Kommandos, die auf derselben Zeile stehen.) Das Muster „*)“ wirkt als „default“, es deckt alle verbleibenden Fälle ab; die Verwendung des default-Zweiges ist optional.

Beispiel:

```
#!/bin/sh
# Interaktive Eingabe, Mehrfachentscheidung (case)
echo Alles in Ordnung?
echo Ihre Antwort:
read answer
echo Ihre Antwort war: $answer
case $answer in
  j*|J*|y*|Y*) echo jawohl ;;
  n*|N*) echo nein, ueberhaupt nicht! ;;
  *) echo das war wohl nichts ;;
esac
```

Schleife: for

```
for i in par1 par2 par3 ...
  do kommandos
done
```

Anmerkungen: Die for-Schleife in der Bourne Shell unterscheidet sich von der for-Schleife in üblichen Programmiersprachen dadurch, daß nicht automatisch eine Laufzahl erzeugt wird. Der Schleifenvariablen werden sukzessive die Parameter zugewiesen, die hinter „in“ stehen. (Die Angabe „for i in $*“ kann durch „for i“ abgekürzt werden.)

Beispiel:

```
#!/bin/sh
# Schleifen: for
echo Uebergabeparameter: $*
# for i
for i in $*
  do echo Hier steht: $i
done
```

Schleife: while und until

```
while [ bedingung ]
  do kommandos
done
until [ bedingung ]
  do kommandos
done
```

Anmerkung: Bei „while“ erfolgt die Prüfung der Bedingung vor der Abarbeitung der Schleife, bei „until“ erst danach. (Anstelle von „[bedingung]“ oder „test bedingung“ kann allgemein ein Kommando stehen, dessen Return-Code geprüft wird; vgl. die Ausführungen zu „if“.)

Beispiel:

```
#!/bin/sh
# Schleifen: while
# mit Erzeugung einer Laufzahl
i=1
while [ $i -le 5 ]
do
  echo $i
  i=`expr $i + 1`
done
```

Kapitel Virtualisierung als PDF exportieren

Virtualisierung

Definition

Bei der Virtualisierung handelt es sich um den Prozess der Erstellung einer softwarebasierten (virtuellen) Komponente anstatt einer hardwarebasierten (physischen). Virtualisieren lassen sich sowohl Anwendungen als auch Server, Storage und Netzwerke. Die Virtualisierung ist unabhängig von der Unternehmensgröße bei Weitem der effektivste Weg zur Reduzierung der IT-Ausgaben bei gleichzeitiger Steigerung der Effizienz und Agilität.

Gründe für Virtualisierung

Durch Virtualisierung kann die Agilität, Flexibilität und Skalierbarkeit der IT erhöht werden. Gleichzeitig werden deutliche Kosteneinsparungen ermöglicht. IT-Komponenten lassen sich einfacher verwalten und kostengünstiger betreiben, da Workloads schneller bereitgestellt, Performance und Verfügbarkeit optimiert sowie Betriebsabläufe automatisiert werden. Weitere Vorteile:

- Senkung der Investitions- und Betriebskosten
- Minimierung oder Vermeidung von Ausfallzeit
- Erhöhung der Produktivität, Effizienz, Agilität und Reaktionsfähigkeit der IT
- Beschleunigung und Vereinfachung des Anwendungs-Provisioning (Bereitstellung von Ressourcen)
- Nachhaltige Planung und einfache Wiederherstellung
- Vereinfachung des Managements von Rechenzentren
- Einfacher Austausch/Einfache Erweiterung von Software (da plattformunabhängig)

Virtualisierungsarten

Eine Einteilung in Gruppen bzw. Kategorien fällt nicht immer ganz leicht, da manche Virtualisierungstechnologien mehreren zugeordnet werden können. Hier findest du eine mögliche Einteilung:

1) Servervirtualisierung (auch Betriebssystemvirtualisierung genannt)

Die Virtualisierung von Servern bedient sich der Virtualisierung von Computern, die auch Hardware-Virtualisierung oder Betriebssystemvirtualisierung genannt wird. Hierbei wird auf physikalischer Computerhardware dem VM Host ein als sog. Hypervisor dienendes Betriebssystem installiert das vorhandene Hardwareressourcen intelligent auf virtuelle Maschinen verteilt. Dadurch ermöglicht diese Konstellation einem Serverbetriebssystem Teile der Vorhandenen Hardware, also Prozessor, Speicher und Anschlüsse, zu nutzen. Dabei wird einem Betriebssystem ein vorhandener Hardwarecomputer vorgespiegelt. Die gängigsten, kommerziellen Lösungen die von Herstellern verfügbar sind VMWare

ESX, Citrix XEN, Microsoft Hyper-V sowie IBM MPAR bei professionellen Mainframe Großrechnersystemen. Freie kostenlose Virtualisierungslösungen für Serverbetriebssysteme sind ebenfalls in großer Anzahl verfügbar. Die bekanntesten wären VMWare ESXi und VMWare Player, XEN, KVM und Oracle VirtualBox.

Man unterscheidet mehrere Arten der Servervirtualisierung:

1.1) Hardwarevirtualisierung

Die wohl am einfachsten zu verstehende Art ist die reine Hardwarevirtualisierung. Hierbei sind nicht die aktuelle Eigenschaften von Intel oder AMD-CPUs gemeint, sondern sogenannte Hardware-Partitions wie sie z. B. in IBMs System p oder Fujitsus Sparc Enterprise Serie realisiert sind. Vereinfacht ausgedrückt kauft der Kunde einen Schrank voller CPUs, Arbeitsspeicherriegel etc. Im Gegensatz zu normalen Servern muss jedoch bei der Erweiterung des Arbeitsspeichers nicht ein neues RAMModul aus dem Keller geholt und explizit in einen Server eingebaut werden. Vielmehr kann dies bei Hardware-Partitions „per Mausklick“ erfolgen, da die vorhandenen Ressourcen innerhalb des „Schranks“ frei auf die logischen Partitions verteilt werden können. Auf jeder so erstellten Partition kann dann ein entsprechendes Betriebssystem installiert werden. Die Virtualisierung erfolgt hierbei durch die Aufteilung der vorhandenen Hardware in kleinere Rechner (die erwähnten Partitions). Vollvirtualisierung

1.2) Vollvirtualisierung

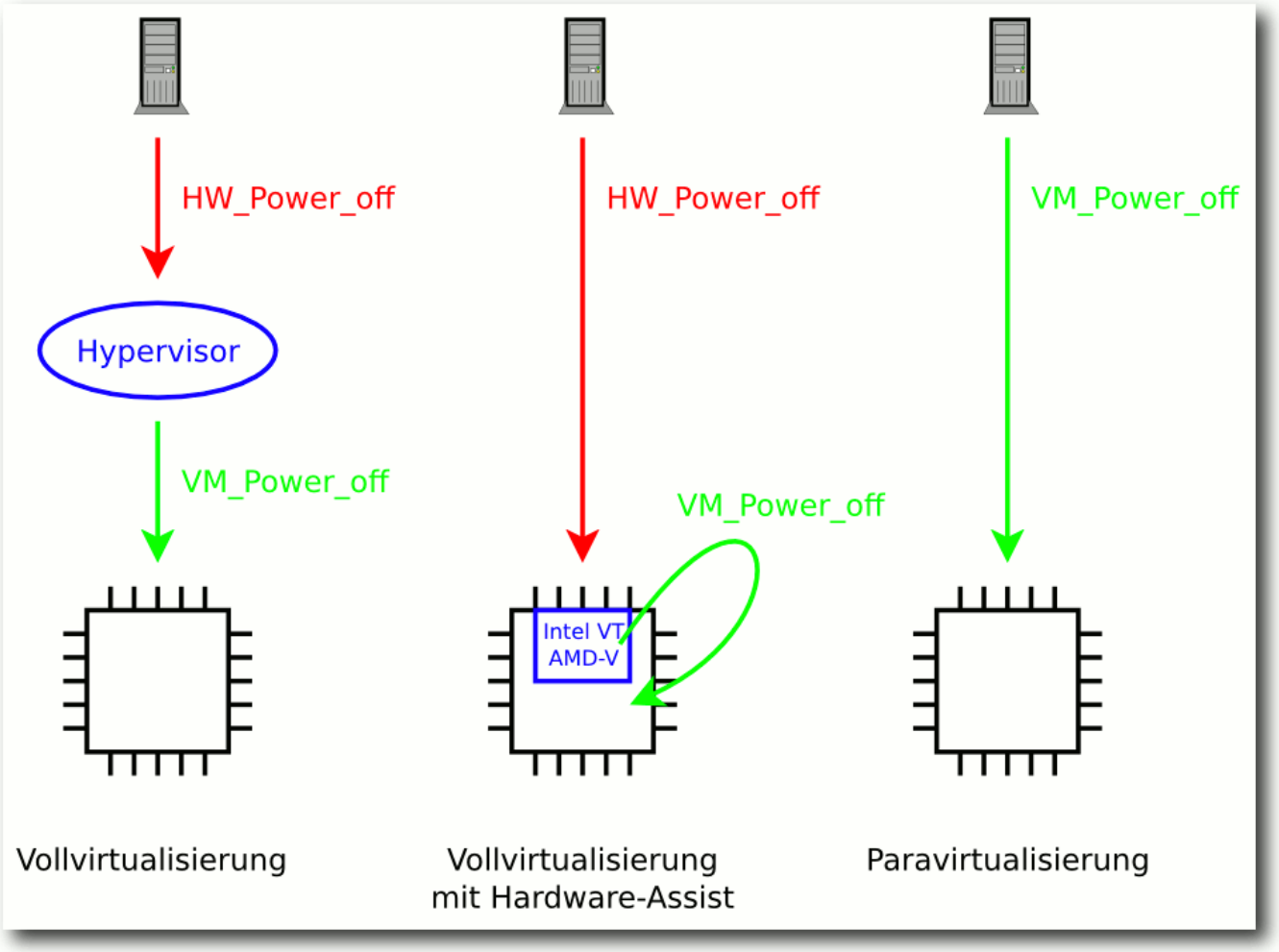
Vollvirtualisierung bezeichnet eine in Software verwirklichte Technologie. Sie kann sowohl als Hypervisor realisiert werden, das heißt die Virtualisierungsschicht läuft direkt auf der Hardware, oder als Hosted-Lösung (siehe unten) In diesen Abschnitt wird erst einmal nur die HypervisorLösung betrachtet. Die Gast-Betriebssysteme müssen bei der Vollvirtualisierung nicht speziell angepasst werden, da diese in der virtuellen Umgebung bekannte Hardware vorfinden. Während das Gastbetriebssystem denkt, auf physikalische Geräte (wie zum Beispiel Netzwerkkarten) zuzugreifen, sind diese jedoch nur virtuell vorhanden. Wenn nun ein Gastbetriebssystem mit einem anderen Rechner kommunizieren will, so sendet es beispielsweise über seine virtuelle Intel EPro1000-Netzwerkkarte Daten ins LAN. Diese Daten müssen nun von der Virtualisierungsschicht über die physikalische Netzwerkkarte weitergeleitet werden („Virtualisierungsoverhead“ für Ein- und Ausgabeoperationen). Reine Rechenoperationen werden direkt auf den physikalischen CPUs ausgeführt. Die Problematik besteht hierbei darin, dass spezielle Befehle unterbunden werden müssen. Als Beispiel sei das Ausschalten einer virtuellen Maschine genannt. Um bei einem einfachen Modell zu bleiben, wird im Folgendem angenommen, dass hierfür das virtualisierte Betriebssystem den CPU-Befehl „Power-Off-Hardware“ ausführt. Würde dieser Befehl jedoch eins-zu-eins vom Hypervisor an den physikalischen Rechner weitergeleitet werden, würde der komplette Server mit allen VMs ausgeschaltet werden – anstatt einer einzelnen VM. Deshalb muss dieser Befehl abgefangen und anders verarbeitet werden – hier wird er durch den Befehl „Power-OffVM“ ersetzt. Dies bedeutet, dass grundsätzlich sämtliche von virtuellen Maschinen ausgeführte Befehle auf solche kritischen Kommandos hin untersucht und bei Bedarf umgeschrieben werden müssen. Dieser Vorgang wirkt sich entsprechend auf die Performance aus („Virtualisierungsoverhead“ für CPU-Befehle).

1.3) Hardware-Assisted-Virtualisierungen

Genau hier setzen Hardware-Assisted Virtualisierungen an. Bei diesen Lösungen übernimmt die Hardware das Umsetzen der Befehle von „Power-Off-Hardware“ zu „Power-Off-VM“. Dies hat zur Folge, dass die Virtualisierungsschicht sich nicht mehr so intensiv um die Befehle der virtuellen Maschinen kümmern muss. Die Technologie wird bei Intel-CPUs als Vanderpool Technology „Intel VT-x“ 4 bzw. bei AMD als AMD-Virtualization „AMD-V“ 5 (früher Codename Pacifica) bezeichnet. In neueren Rechnergenerationen wird dieses Prinzip nicht mehr nur für CPUs sondern auch für Ein- und AusgabeBefehle verwendet (z. B. „Intel VT-d“).

1.4) Paravirtualisierung

Als letzte Technologie in diesem Abschnitt ist die Paravirtualisierung zu nennen. Der Unterschied zu den bisher genannten Techniken ist, dass hierbei das Gastbetriebssystem angepasst werden muss. Die Anpassung sorgt dafür, dass die virtuelle Maschine weiß, dass sie nicht auf physikalischer Hardware läuft und somit direkt den Befehl „Power-Off-VM“ ausführt. Deshalb erzeugt Paravirtualisierung weniger Overhead mit dem Nachteil der nötigen Anpassung des Gastes. Dadurch war diese Technologie zuerst nur Linux-Kernel basierten VMs vorenthalten. Microsoft hat mittlerweile auch für Windows Server 2008 solche Anpassungen entwickelt („Enlightenments“), welche jedoch ausschließlich Vereinfachtes Schema der Betriebssystemvirtualisierung. von Microsoft-Virtualisierungslösungen (sprich von Microsofts Hyper-V) verwendet werden dürfen, wenn nicht explizit ein Vertrag zusätzlich mit Microsoft ausgehandelt wurde. Es ist davon auszugehen, dass Microsoft sich dies entsprechend honorieren lässt. Um dieses Problem etwas zu mildern, wurden spezielle Treiber entwickelt, welche bei einem vollvirtualisierten Gast zumindest Teile paravirtualisiert betreiben können (z. B. der paravirtualisierte SCSI-Treiber derVMware Tools).



2) Desktop Virtualisierung

Die Virtualisierung von Desktop PCs entspricht technisch im weitest gehenden der Virtualisierungstechnik für Server. In der neuesten Generation der Desktop Virtualisierungen kann ein Virtualisierter PC auf beliebige Endgeräte der Anwender wie Laptops, Pads, Thin Clients oder PCs plattformübergreifend gestreamt werden. Für Desktopbetriebssysteme können die professionellen Virtualisierungslösungen für Server ebenfalls verwendet werden. Spezielle Softwarelösungen für Desktop Virtualisierung sind Microsoft Virtual PC; VMWare Workstation und Apple‘s Parallels Workstation. Konkret heißt das ein Windows 10 läuft nicht mehr am Arbeitsplatz-Rechner sondern innerhalb einer virtuellen Server-Infrastruktur (ESX, XenServer, Hyper-V). Die Benutzer verwenden entweder einen Terminal-Services-Client oder direkt einen ThinClient mit Hilfe dessen sie die virtuellen Clients steuern können.

Vorteile sind hierbei, dass sämtliche Server und Client-Betriebssysteme an einem zentralen Ort betrieben werden können (z. B. VMware vSphere-Umgebung im Serveraum). Somit ist eine zentrale Datenhaltung mit verbesserten Backup-, Performance- oder KostenersparnisMöglichkeiten gegeben. Für die Verwaltung der virtuellen Desktop-VMs wurden VMware View und Citrix XenDesktop entwickelt. Einen entscheidenden Nachteil teilen sich jedoch alle Virtualisierungslösungen, die für die (Client-)Benutzer-Interaktion ausgelegt sind: die Grafikkartenperformance. Bei einem physikalischen Client-PC ist die Grafikkarte direkt mit dem Monitor verbunden, während bei einer Client-VM die Daten von der virtuellen Grafikkarte im Serverraum erst über das Netzwerk zur lokalen Grafikkarte und letztlich zum Monitor übertragen werden müssen.

3) Applikations/Anwendungs-Virtualisierung

Die Virtualisierung von Applikationen erlaubt es Softwarepakete in komplexer Konfiguration zeitnah und unabhängig von Benutzerprofilen für Desktop PCs oder Notebooks bereitzustellen. Hierbei wird eine benötigte Anwendung inklusive aller konfigurierten Einstellungen in eine ausführbare .exe Datei umgewandelt. Durch die Bereitstellung der Datei im Netzwerk wird die Installation und Administration auf einzelnen Clients überflüssig was den administrativen Zeitaufwand erheblich verringert. Die entsprechende Software kann für den Außendienst oder das Homeoffice als portable Anwendungsdatei auf Computern gespeichert werden. Verfügbare Lösungen für die Software Virtualisierung sind VMWare ThinApp und Spoon Virtual Application Studio.

Man unterscheidet mehrere Arten der Anwendungsvirtualisierung:

3.1) Gehostete Vollvirtualisierung

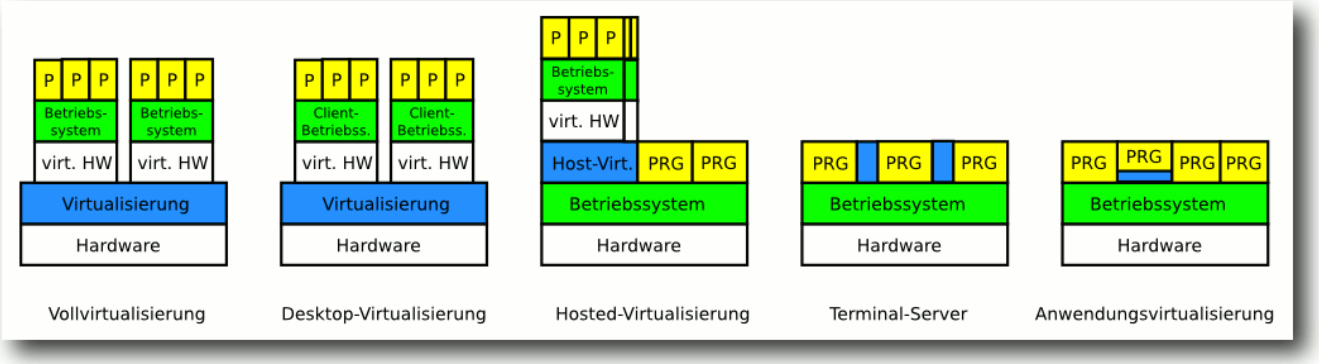
Die gehostete Vollvirtualisierung wird in diesem Artikel der Gruppe der Anwendungsvirtualisierungen zugeordnet, während sie ebenfalls der Betriebssystemvirtualisierung zugeordnet werden könnte. Da jedoch die Virtualisierungsschicht nicht direkt auf der Hardware läuft, sondern vielmehr ein Basisbetriebssystem erforderlich ist, innerhalb dessen die Virtualisierung als normale Anwendung läuft, wird sie diesen Abschnitt zugeordnet. Als bekannte Vertreter dieser Gattung wären unter anderem VirtualBox, Microsoft VirtualPC und VMware Server/Workstation zu nennen. Die Vorteile liegen auf der Hand: Es ist nicht mehr die Virtualisierungsschicht für das Ansprechen der physikalischen Hardware verantwortlich, sondern das zu Grunde liegende Betriebssystem (z. B. Linux, Windows, Mac OS X). Somit müssen keine eigenen Treiber entwickelt werden. Beispielsweise unterstützt VMware ESX-Server 4 nur eine begrenzte Anzahl an Netzwerkkarten, während die Lösung mit VMware Server sozusagen alle von Linux und Windows unterstützten Geräte verwenden kann. Auch Citrix XenServer, welches auf einem Linux-Kernel basiert und somit theoretisch sämtliche Linux-Treiber verwenden könnte, unterstützt nur einen Teil derer. Der Hauptnachteil von Hosted-Lösungen ist jedoch, dass das Virtualisierungsprogramm nur noch eines von vielen im Host-Betriebssystem ist und entsprechend auch als solches behandelt wird. Des Weiteren ist nicht garantiert, dass ein für Windows zertifizierter Treiber auch mit sämtlichen auf Windows ausführbaren Virtualisierungslösungen fehlerfrei funktioniert, während die Hersteller von Hypervisor-Produkten ihre Treiber genau für diesen Anwendungsfall testen.

3.2) Terminal-Server

Die wohl bekannteste Art der Anwendungsvirtualisierungen sind Terminal-Server. Hierbei werden nicht verschiedene Betriebssystem-Instanzen parallel auf einem Rechner ausgeführt, sondern nur parallele Sitzungen mehrere Nutzer innerhalb eines Betriebssystems. Unter Windows wären Microsofts Terminal Services und Citrix XenApp zu nennen, während Linux mit einer Vielzahl von Lösungen aufwarten kann. Dies liegt höchstwahrscheinlich daran, dass Linux schon immer als Mehrbenutzer-System entwickelt wurde. So können bereits die StandardDesktop-Manager (kdm, gdm) über XDMCP Sitzungen für Remote-Benutzer bereitstellen. Um Anwendungen in einer Mehrbenutzer-Umgebung von einander abzuschotten, wird normalerweise auf Home-Verzeichnissen und Variablen gesetzt. Dabei liegen die Programmdaten nur einmal vor (z. B. C:\Program Files\... bei Windows bzw. /usr/bin bei Linux), während für das Speichern der Dokumente eine Variable %HOMEPATH% (z. B. C:\Users\admin\) bzw. $HOME (z. B. /home/admin/) verwendet wird.

3.3) Anwendungsvirtualisierungen

Eine verwandte Art sind die (für viele Anwender missverständlich) direkt als Anwendungsvirtualisierungen betitelten Lösungen. Linux-Benutzer kennen ein ähnliches Verfahren seit Jahren. Gemeint ist eine chroot-Umgebung. Hierbei wird eine Anwendung in einer vom eigentlichen Betriebssystem abgekoppelten Umgebung („Sandbox“) ausgeführt. Die Technologien gehen hierbei heutzutage jedoch weiter und spiegeln eher das Verhalten von UnionFS wieder. Die Produkte der führenden Hersteller nennen sich VMware ThinApp, Citrix XenApp Streaming und Microsoft App-V . Alle beruhen auf dem Prinzip, dass ein Programm denkt, es laufe ungehindert im Betriebssystem, während in Wirklichkeit jedoch sämtliche Lese- und Schreiboperationen abgefangen und bei Bedarf umgebogen werden können. Einem Programm kann somit simuliert werden, dass es z. B. Daten von C:\temp nach D:\temp kopieren kann. In Wirklichkeit existieren jedoch die Ordner nicht einmal im Dateisystem sondern nur innerhalb der virtualisierten Anwendungsumgebung.



4) Storage Virtualisierung

Die Virtualisierung von Speichersystemen stellt die Zusammenfassung vorhandener Speicherressourcen wie Festplatten, NAS und SAN Systeme dar. Hierbei werden die vorhandenen Datenspeicher unterschiedlicher Technologien und Hersteller in einem gemeinsamen großen Speicherpool zusammengefügt. Einzelne, vorhandene Rechnersysteme können benötigten Speicherplatz in flexibler Größe aus dem virtuellen Speichersystem in Anspruch nehmen. Durch Hinzufügen weiterer Speicherhardware kann bei anwachsenden Datenmengen der Speicherpool im laufenden Betrieb beliebig vergrößert werden. Zusätzlich minimiert Storage-Virtualisierung den Aufwand zur Schaffung von Redundanz als Schutzmaßnahme gegen Datenverlust oder Betriebsausfälle. Man unterscheidet zwischen drei Arten der Storage-Virtualisierung. Die Hostbasierte Lösungen wie Datacore SANSymphony, Symantec Storage Foundation for Windows oder HPs LeftHand Virtual SAN laufen auf eigenen Rechnersystemen. Die zweite Möglichkeit findet direkt in Speicherhardware wie SAN Systemen statt bei denen weitere Geräte und Switches als zusätzliche vermittelnde Schicht ins SAN integriert werden. Der Storage Controller erkennt diese als Hostsystem während zugreifende Hosts diese als Storage erkennen. Die Produkte solcher kombinierter Systeme sind u.A. EMC Invista, LSI Logic StoreAge und HPs SVSP. Die dritte Möglichkeit ist Storage Virtualisierung die direkt in einem dedizierten Controller, der Anschlussmöglichkeiten für verschiedene Speichersysteme anbietet, stattfindet. Hierzu zählen IBM Systems SVC (Storage Virtualisierungs Controller) oder VMWare vSphere Storage Appliance.

5) Netzwerk Virtualisierung

Die Virtualisierung von Netzwerken bietet die Möglichkeit ein Firmennetzwerk über bauliche Abgrenzungen hinweg in mehrere Teilnetze aufzuteilen. Bei einer klassischen physikalischen Netzwerkinfrastruktur gilt ein Adressbereich für das ganze Unternehmen. Zugriffsberechtigungen und Netzwerkkontrolle kann bei dieser Konfiguration nur durch Dateiberechtigungen und komplizierte Firewall Lösungen ausgeübt werden. Eine physikalische Trennung einzelner Abteilungen ist bei dieser Konfiguration beinahe unmöglich. Bei einem virtualisierten Netzwerk können einzelne Clients sogar über Stockwerksgrenzen hinweg einer einzelnen Netzwerkgruppe zugeordnet werden. Jedes der so erstellten virtuellen Netzwerke hat einen eigenen Adressbereich obwohl sich diese im gleichen physikalischen Netzwerk befinden. Ressourcen der Entwicklungsabteilung z.B. sind somit vor ungewollten oder unautorisierten Zugriffen durch eine andere Abteilung auf unterster Ebene abgesichert. Mittlerweile implementiert fast jeder Hersteller von Netzwerkhardware die Möglichkeit zur Virtualisierung von Netzwerken in seine Produkte. Voraussetzung sind Layer2 oder Layer3 fähige Netzwerkswitche. Die besten Erfahrungen bei der Netzwerkvirtualisierung hatten wir mit Produkten der Hersteller Cisco, Extreme Networks, HP und Juniper.

VMNetwerkverbindungsarten

VMware Workstation bietet mehrere Möglichkeiten, die Netzwerkressourcen des Hosts zu nutzen. Je nach Anforderungen wird man eine dieser Möglichkeiten auswählen. Hier sind die wichtigsten erklärt:

Bridge

Hier benutzt der Gast die Netzwerkverbindung des Hosts mit einer eigenen IP in dessen lokalem Netz. Das kommt der Installation eines separaten Rechners gleich – auch von außen her gesehen.

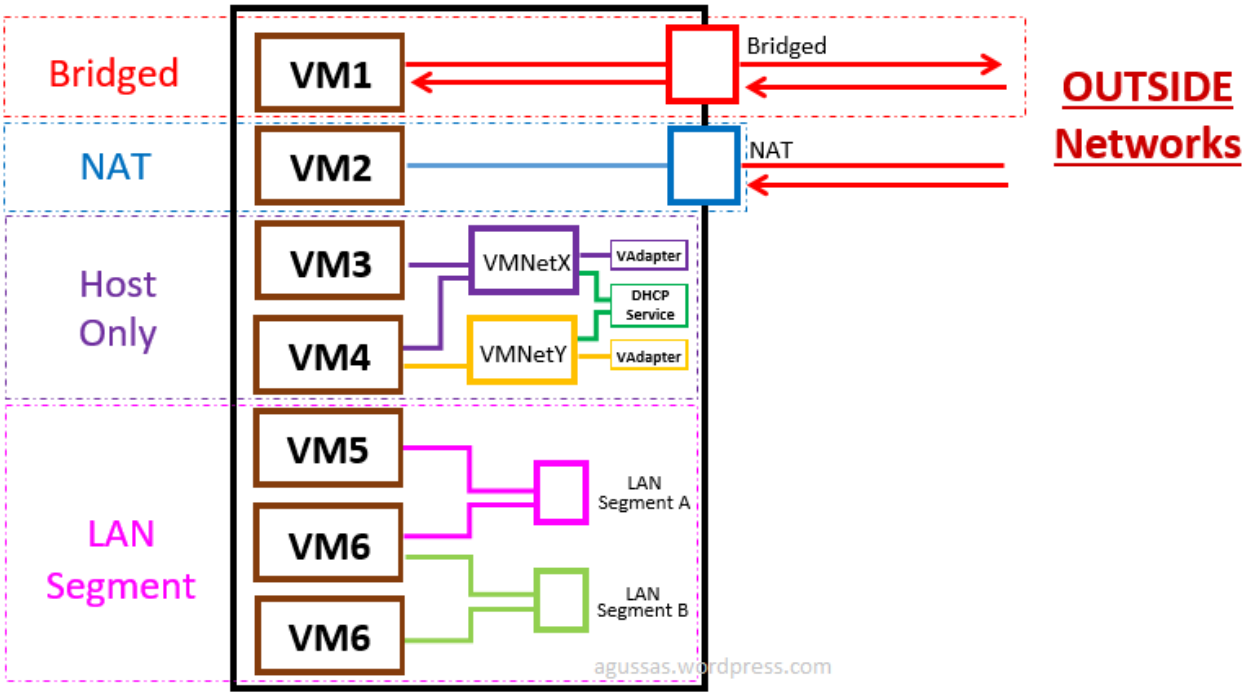
NAT

Der Gast bekommt eine IP in einem von VMware dafür eingerichteten privaten Netz, das vom physischen Netz des Hostrechners verschieden ist. Es kommt ein virtueller Netzwerkadapter zum Einsatz, den VMWare im Host einrichtet und mit einer weiteren IP des privaten Netzes konfiguriert; der Host wird auf Seite des Clients als Default-Gateway eingetragen. Via Adressübersetzung (NAT) kann der Gast nun das Host-Netz erreichen. Ressourcen des Gasts, z. B. Windows-Freigaben, sind nur vom Host aus unter der privaten IP des Gasts erreichbar. Von außen her ist nur eine IP sichtbar; dass sich dahinter mehrere Systeme befinden, kann nur durch Analyse des Datenverkehrs erkannt werden.

Host only

Auch hier richtet VMware ein privates Netz ein. Es werden jedoch keine Regeln definiert, die Datenpaketen des Gastes erlauben, dieses private Netz zu verlassen. Wenn zusätzliche Verbindungen gewünscht sind, müssen diese auf dem Host durch Routing (Forwarding) explizit hergestellt oder als Serverdienst (z. B. Proxy) realisiert werden. Diese Methode eignet sich vorzüglich, um einen dedizierten Server im lokalen Netz zu betreiben. Beispielsweise würde man für einen Terminalserver

nur den Port für RDP freischalten. Damit wäre die Maschine für ihren eigentlichen Bestimmungszweck im Netz erreichbar, während z. B. Viren, die sich über andere Ports verbreiten, beim Host enden würden. Auch für private Zwecke eignet sich diese Methode, da damit verhindert werden kann, dass der Gast unerwünschte IP-Verbindungen (z. B. für Spamversand) aufbaut. Anmerkung: Mehrere Gastsysteme können separate private Netze nutzen, die nur miteinander kommunizieren können, wenn es jeweils im Hostsystem explizit konfiguriert ist.



Kapitel C++ als PDF exportieren

C++

C+ + ist eine von der ISO genormte Programmiersprache. Sie wurde ab 1979 von Bjarne Stroustrup bei AT&T als Erweiterung der Programmiersprache C entwickelt. C+ + ermöglicht sowohl die effiziente und maschinennahe Programmierung als auch eine Programmierung auf hohem Abstraktionsniveau. Der Standard definiert auch eine Standardbibliothek, zu der verschiedene Implementierungen existieren.

Lernplattform SoloLearn

- C++ spielerisch lernen

Nachschlagewerke

- Einstieg in C++
- WikiBook zur C++ Programmierung
- Kurzanleitung für GNU C/C++ - Compiler
- C++ unter Linux

Behandelte Inhalte

- Funktionen
- Parameterübergabe
- Zeiger
- Arrays & Sortieralgorithmen
- Strukturen
- Klassen
- Dateibehandlung

Beispiele vom Unterricht

- Beispiel zur Parameterübergabe bei Funktionen
- Beispiel zu Arrays, Schleifen und Funktionen
- Beispiel zu Arrays, Funktionen, Schleifen und Zufallszahlen
- Beispiel zu Arrays, Sortieralgorithmen und Funktionen
- Schiffe versenken einfach - Beispiel zu Mehrdimensionale Arrays, Funktionen, Schleifen
- Schiffe versenken erweitert - Beispiel zu Strukturen, Mehrdimensionale Arrays, Funktionen, Schleifen, Zufallszahlen

Funktionen in C++

Ein wichtiges Sprachelement von C+ + kam bisher noch überhaupt nicht vor: die Funktion. Die Möglichkeit, Funktionen zu bilden, ist ein herausragendes Merkmal einer Programmiersprache. Ganz allgemein versteht man unter einer Funktion einen in sich geschlossenen Programmteil, der eine bestimmte Aufgabe erfüllt. Der Vorteil einer Funktion ist die Wiederverwendbarkeit.

Betrachten wir folgendes Beispiel:

```
int add( int x, int y)
{
  int z = x+y;

  return z;
}
```

An diesem Codeausschnitt erkennen Sie alle Bestandteile einer Funktion:

- Typ des Rückgabewerts: int
- Funktionsname: add
- Argumentliste: (int x, int y)
- Funktionskörper: Anweisung innerhalb des Blocks der Funktion, begrenzt durch die geschweiften Klammern { und }
- Return-Anweisung: return z; (bei Rückgabetyp void nicht nötig)

Zu all diesen Teilen ist natürlich noch Einiges zu sagen. Vorher noch ein typografischer Hinweis: Es hat sich eingebürgert, in Büchern zwei Klammern hinter Bezeichner zu setzen, die sich auf eine Funktion beziehen, zum Beispiel in Sätzen wie: Mit add() erreichen Sie die Addition zweier Ganzzahlen. Das sagt noch nichts über Art und Umfang der Argumentliste aus, sondern soll Sie lediglich daran erinnern, dass es dabei um eine Funktion und nicht um eine Variable geht. Ich will mich auch in diesem Buch daran halten.

Rückgabewert

In C+ + muss jede Funktion einen Typ für den Wert angeben, den sie zurückliefert. Manchmal ist es aber auch gar nicht nötig oder sinnvoll, dass eine Funktion überhaupt einen Rückgabewert hat. In diesem Fall geben Sie als Typ void an.

Was macht man nun mit einem solchen Wert? Der Programmteil, der die Funktion aufruft, kann diese an allen Stellen einsetzen, wo er sonst eine Variable oder Konstante angeben würde (in obiger Form allerdings nur dort, wo lediglich der Wert benötigt wird), also etwa:

```
int main()
{
  int a = 5;
  int b = 12;
  int c = add(a,b);
```

```
  cout << ``a = `` << a << ``, c = `` << c << ``, a+c = `` << add(a,c) <<
endl;

  return 0;
}
```

Dieses Programm hat dann die Ausgabe: a = 5, c = 17, a+c = 22

Übrigens: Selbst wenn eine Funktion einen Wert zurückgibt, müssen Sie ihn nicht beachten. Sie dürfen auch schreiben:

```
int a = 5;
int b = 12;
add(a,b);
```

auch wenn das hier keinen Sinn machen würde. Bei Funktionen mit Rückgabetyp void ist das hingegen die übliche Form des Aufrufs. Allgemein kommt es aber häufiger vor, dass Rückgabewerte ignoriert werden. Beispielsweise geben viele Funktionen Statusinformationen darüber zurück, wie gut (oder schlecht) sie ihre Aufgabe erfüllen konnten. Viele Anwender solcher Funktionen interessieren sich nicht für den Status und übergehen ihn. Das kann manchmal aber auch gefährlich werden, wenn etwa aufgetretene Fehler aus diesem Grund zunächst unentdeckt bleiben.

Funktionsname

Wie alle anderen Bezeichner in C++ dürfen Sie auch Funktionsnamen nur aus Buchstaben, Ziffern sowie dem Unterstrich _ bilden. Außerdem ist es nicht erlaubt, Funktionsnamen zu verwenden, die Schlüsselwörten gleichen (etwa for).

Argumentliste

Eine Funktion kann immer nur auf den Daten arbeiten, die ihr lokal vorliegen. Außer global (das heißt außerhalb aller Funktionen) definierten Variablen sind das nur die Parameter, die das Hauptprogramm an die Funktion übergibt. Von diesen Parametern (auch Argumente genannt) können Sie keinen, einen oder mehrere angeben, die Sie dann durch Kommas trennen.

Wenn Sie eine Funktion ohne Argumente schreiben wollen, lassen Sie den Bereich zwischen den beiden runden Klammern einfach leer – denn die Klammern müssen Sie stets schreiben! – oder Sie setzen ein Argument vom Typ void ein.

Ansonsten geben Sie für jeden Parameter seinen Datentyp und einen Namen an, unter dem er in der Funktion bekannt sein soll. Dieser Name kann vollkommen anders sein, als der im Hauptprogramm beim Aufruf verwendete. Auch in obigem Beispiel heißen die Summanden in der Funktion x und y, im Hauptprogramm aber a und b.

Funktionskörper

Hier stehen die Anweisungen, die bei einem Aufruf der Funktion ausgeführt werden. Man kann darüber streiten, wie lang Funktionen sein sollten. Es gibt Experten, die fordern, dass eine Funktion aus nicht mehr als 50 Zeilen bestehen dürfe, sonst werde sie unleserlich. Es gibt jedoch in der Praxis immer wieder Fälle, in denen längere Funktionen sinnvoll sind. Bei der objektorientierten Programmierung werden Sie allerdings ohnehin wesentlich mehr Funktionen (beziehungsweise Methoden) verwenden, die im Durchschnitt wesentlich kürzer sind als bei der prozeduralen Programmierung.

Innerhalb des Funktionskörpers können Sie die Funktionsparameter wie normale Variablen verwenden; zusätzlich können Sie natürlich auch noch lokale Variablen definieren. Außerdem ist es selbstverständlich erlaubt, aus einer Funktion wieder andere Funktionen aufzurufen. (Sie dürfen sogar die Funktion selbst wieder aufrufen; man spricht dann von einer rekursiven Funktion – aber das ist ein eigenes Thema.)

Return-Anweisung

Die Anweisungen im Funktionskörper werden so lange abgearbeitet, bis das Programm auf das Ende der Funktion oder eine return-Anweisung trifft. Diese erfüllt einen doppelten Zweck:

Sie legen fest, welchen Wert die Funktion an das Hauptprogramm zurückliefern soll. Das kann eine Variable sein oder ein Ausdruck, eine Konstante oder der Rückgabewert einer anderen Funktion (wobei Letzteres als schlechter Stil gilt). Hat Ihre Funktion den Rückgabetyp void, geben Sie an dieser Stelle überhaupt nichts an. Der Typ des angegebenen Werts muss jedoch in jedem Fall mit dem deklarierten Rückgabetyp übereinstimmen. Sie beenden die Funktion und kehren zum Hauptprogramm zurück. Jede return-Anweisung – sei sie nun am Ende oder irgendwo inmitten des Funktionskörpers – markiert das Ende der Abarbeitung der Funktion und den Rücksprung an die Stelle, von der aus die Funktion aufgerufen wurde. Sie können also die Funktion schon beenden, bevor alle Anweisungen ausgeführt sind, zum Beispiel wenn eine bestimmte Bedingung erfüllt ist. Ist der Rückgabetyp void, muss am Ende der Funktion keine return-Anweisung stehen (auch nicht das Schlüsselwort return), wie in folgendem Beispiel:

```
void ausgabe( int z)
{
  cout << ``Das Ergebnis ist: `` << z << endl;
}
```

Wenn Sie allerdings bei Funktionen mit irgendeinem anderen Rückgabetyp die return-Anweisung vergessen, meldet der Compiler einen Fehler.

Der Prototyp

Bevor Sie eine Funktion verwenden können, müssen Sie dem Compiler zunächst mitteilen, dass es eine Funktion dieses Namens gibt, wie viele und welche Parameter sie hat und welchen Typ sie zurückliefert. Dies geschieht mit einem so genannten Prototyp der Funktion. Der Prototyp sieht

genauso aus wie die Funktion selbst bis auf den Funktionskörper; dieser fehlt und wird durch ein einfaches Semikolon ; ersetzt. (Es ist sogar erlaubt, die Namen der Argumente wegzulassen und nur ihre Typen anzugeben. Analog zu den Klassen (siehe Seite [*]) ist also der Prototyp die Deklaration und die Funktion mit Körper die Definition.

Aufruf

Der Aufruf einer Funktion erfolgt durch Nennung des Funktionsnamens. An den Funktionsnamen schließt sich immer ein Klammerpaar an, das gegebenenfalls auch Parameter enthalten kann. Dieses Klammerpaar ist zwingend erforderlich. Die Parameter des Aufrufs müssen zu den Parametern der Funktion passen. Besitzt die Funktion einen Rückgabewert, kann der Funktionsaufruf als Ausdruck verwendet werden. Er kann also beispielsweise auf der rechten Seite einer Zuweisung stehen.

Übergabe der Argumente

C+ + kennt mehrere Varianten, wie einer Funktion die Argumente übergeben werden können: call-by-value, call-by-reference und call-by-pointer.

call-by-value (Wertübergabe)

Bei call-by-value (Wertübergabe) wird der Wert des Arguments in einen Speicherbereich kopiert, auf den die Funktion mittels Parametername zugreifen kann. Ein Werteparameter verhält sich wie eine lokale Variable, die „automatisch“ mit dem richtigen Wert initialisiert wird. Der Kopiervorgang kann bei Klassen (Thema eines späteren Kapitels) einen erheblichen Zeit- und Speicheraufwand bedeuten!

Beispiel

```
#include <iostream>
#include <conio.h>

using namespace std;

//Prototyp, Deklaration
int sumW(int x,int y);   //Parameterübergabe per Wert (Value)

int main(int argc, char** argv) {
    cout << "Hello World" << endl;

    int a=5,b=6,erg=0;
    cout << "Parameteruebergabe per Wert" << endl;
    erg=sumW(a, b);
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    getch();

    return 0;
}

//Funktionendefinition - Parameterübergabe per Wert
int sumW(int x,int y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    return x+y;
```

```
}
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
a	0x0001	5
b	0x0005	6
...	.	.
...	.	.
x	0x00a1	5 → 6
y	0x00a5	6 → 7

Die Ausgabe des Programms ist:

```
Hello World
Parameteruebergabe per Wert
x: 6
y: 7
a: 5
b: 6
```

call-by-reference (Übergabe per Referenz)

Sollen die von einer Funktion vorgenommen Änderungen auch für das Hauptprogramm sichtbar sein, müssen in C sogenannte Zeiger verwendet werden. C+ + stellt ebenfalls Zeiger zur Verfügung. C+ + gibt Ihnen aber auch die Möglichkeit, diese Zeiger mittels Referenzen zu umgehen, was im alten C nicht möglich war. Beide sind jedoch noch Thema eines späteren Kapitels.

Im Gegensatz zu call-by-value wird bei call-by-reference die Speicheradresse des Arguments übergeben, also der Wert nicht kopiert. Änderungen der (Referenz-)Variable betreffen zwangsläufig auch die übergebene Variable selbst und bleiben nach dem Funktionsaufruf erhalten. Um call-by-reference anzuzeigen, wird der Operator & verwendet, wie Sie gleich im Beispiel sehen werden. Wird keine Änderung des Inhalts gewünscht, sollten Sie den Referenzparameter als const deklarieren, um so den Speicherbereich vor Änderungen zu schützen. Fehler, die sich aus der ungewollten Änderung des Inhaltes einer übergebenen Referenz ergeben, sind in der Regel schwer zu finden.

```
#include <iostream>
#include <conio.h>

using namespace std;

//Prototyp, Deklaration
int sumR(int &x,int &y);   //Parameterübergabe per Referenz (Reference)


int main(int argc, char** argv) {
    cout << "Hello World" << endl;
```

```
    cout << "Parameteruebergabe per Referenz" << endl;
    a=5,b=6,erg=0;
    erg=sumR(a, b);
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;


    getch();

    return 0;
}

//Funktionendefinition - Parameterübergabe per Referenz
int sumR(int &x,int &y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    return x+y;
}
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
x,a	0x0001	5 → 6
y,b	0x0005	6 → 7
...	.	
...	.	
...	.	
...	.	

Die Ausgabe des Programms ist:

```
Hello World
Parameteruebergabe per Referenz
x: 6
y: 7
a: 6
b: 7
```

call-by-pointer (Übergabe per Zeiger)

Wenn Sie einen Zeiger als Parameter an eine Funktion übergeben, können Sie den Wert an der übergebenen Adresse ändern.

```
#include <iostream>
#include <conio.h>

using namespace std;

//Prototyp, Deklaration
int sumZ(int *x,int *y);   //Parameterübergabe per Zeiger (Pointer)

int main(int argc, char** argv) {
    cout << "Hello World" << endl;
    cout << "Parameteruebergabe per Zeiger" << endl;
    a=5,b=6,erg=0;
    erg=sumZ(&a, &b);
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    getch();

    return 0;
}


//Funktionendefinition - Parameterübergabe per Zeiger
int sumZ(int *x,int *y){
    *x+=1; //x=x+1; x++; ++x;
    *y+=1;

    cout << "x: " << *x <<endl;
    cout << "y: " << *y <<endl;

    return *x+*y;
}
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
a	0x0001	5 → 6
b	0x0005	6 → 7
...	.	
...	.	
*x	0x00a1	0x0001
*y	0x00a5	0x0005

Die Ausgabe des Programms ist:

```
Hello World
Parameteruebergabe per Zeiger
x: 6
y: 7
```

```
a: 6
b: 7
```

Funktionen - Beispiel

Angabe

Dieses Programm zeigt die verschiedenen Arten der Parameterübergabe bei Funktionen auf. Es werden jeweils dieselben Variablen auf 3 verschiedenen Arten (per Wert, per Referenz, per Zeiger) übergeben.

Lösung

```
/* Beispiel: Funktionen - Parameterübergabe
   Filename:Parameteruebergabe.cpp
   Author:Lahmer
   Title:Parameterübergabe
   Description: Anhand dieses Beispiels werden die verschiedenen Arten der
Parameterübergabe erklärt
   Last Change:16.01.2018
*/

//Hedader-Dateien bzw. Bibliotheken
#include <iostream>
#include <conio.h>

//Namespace (Namensraum)
using namespace std;

//Prototyp, Deklaration
int sumW(int x,int y);   //Parameterübergabe per Wert (Value)
int sumR(int &x,int &y);   //Parameterübergabe per Referenz (Reference)
int sumZ(int *x,int *y);   //Parameterübergabe per Zeiger (Pointer)


//Hauptprogramm
int main(int argc, char** argv) {
    //Ausgabe
    cout << "Hello World" << endl;

    //Lokale Variablen mit Datentyp int => ganzzahlige Zahlen
    int a=5,b=6,erg=0;

    //Parameterübergabe per Wert
    cout << "Parameteruebergabe per Wert" << endl;
    //Funktionsaufruf per Wert = Kopie der Variable
    erg=sumW(a, b);
    //Ausgabe der Variablen a bzw. b
    cout << "a: " << a <<endl;
```

```
    cout << "b: " << b <<endl;

    //Parameterübergabe per Referenz
    cout << "Parameteruebergabe per Referenz" << endl;
    a=5,b=6,erg=0;
    //Funktionsaufruf per Referenz = Verknüpfung der Variablen a bzw. b
    erg=sumR(a, b);
    //Ausgabe der Variablen a bzw. b
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    //Parameterübergabe per Zeiger
    cout << "Parameteruebergabe per Zeiger" << endl;
    a=5,b=6,erg=0;
    //Funktionsaufruf per Referenz = Übergabe der Adressen der Variablen a
bzw. b
    erg=sumZ(&a, &b);
        //Ausgabe der Variablen a bzw. b
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    //Auf Tastendruck warten
    getch();

    //Rückgabewert = 0
    return 0;
}

//Funktionendefinition - Parameterübergabe per Wert
int sumW(int x,int y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    //Rückgabewert Summe von x+y
    return x+y;
}

//Funktionendefinition - Parameterübergabe per Referenz
int sumR(int &x,int &y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    //Rückgabewert Summe von x+y
    return x+y;
}
```

```
//Funktionendefinition - Parameterübergabe per Zeiger
int sumZ(int *x,int *y){
    *x+=1; //x=x+1; x++; ++x;
    *y+=1;

    cout << "x: " << *x <<endl;
    cout << "y: " << *y <<endl;

    //Rückgabewert Summe von x+y
    return *x+*y;
}
```

Ausgabe

```
Hello World
Parameteruebergabe per Wert
x: 6
y: 7
a: 5
b: 6
Parameteruebergabe per Referenz
x: 6
y: 7
a: 6
b: 7
Parameteruebergabe per Zeiger
x: 6
y: 7
a: 6
b: 7
```

Zeiger (Pointer)

Zeiger (engl. pointers) sind Variablen, die als Wert die Speicheradresse einer anderen Variable enthalten.

Jede Variable wird in C++ an einer bestimmten Position im Hauptspeicher abgelegt. Diese Position nennt man Speicheradresse (engl. memory address). C++ bietet die Möglichkeit, die Adresse jeder Variable zu ermitteln. Solange eine Variable gültig ist, bleibt sie an ein und derselben Stelle im Speicher.

Am einfachsten vergegenwärtigt man sich dieses Konzept anhand der globalen Variablen. Diese werden außerhalb aller Funktionen und Klassen deklariert und sind überall gültig. Auf sie kann man von jeder Klasse und jeder Funktion aus zugreifen. Über globale Variablen ist bereits zur Kompilierzeit bekannt, wo sie sich innerhalb des Speichers befinden (also kennt das Programm ihre Adresse).

Zeiger sind nichts anderes als normale Variablen. Sie werden deklariert (und definiert), besitzen einen Gültigkeitsbereich, eine Adresse und einen Wert. Dieser Wert, der Inhalt der Zeigervariable, ist aber nicht wie in unseren bisherigen Beispielen eine Zahl, sondern die Adresse einer anderen Variable oder eines Speicherbereichs. Bei der Deklaration einer Zeigervariable wird der Typ der Variable festgelegt, auf den sie verweisen soll.

```
#include <iostream>

int main() {
     int   Wert;      // eine int-Variable
     int *pWert;      // eine Zeigervariable, zeigt auf einen int
     int *pZahl;      // ein weiterer "Zeiger auf int"

     Wert = 10;       // Zuweisung eines Wertes an eine int-Variable

     pWert = &Wert;   // Adressoperator '&' liefert die Adresse einer
Variable
     pZahl = pWert;   // pZahl und pWert zeigen jetzt auf dieselbe Variable
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
Wert	0x0001	10
*pWert	0x0005	0x0001
*pZahl	0x0009	0x0005
...	.	.
...	.	.

Der Adressoperator & kann auf jede Variable angewandt werden und liefert deren Adresse, die man einer (dem Variablentyp entsprechenden) Zeigervariablen zuweisen kann. Wie im Beispiel gezeigt, können Zeiger gleichen Typs einander zugewiesen werden. Zeiger verschiedenen Typs bedürfen einer Typumwandlung. Die Zeigervariablen pWert und pZahl sind an verschiedenen Stellen im Speicher abgelegt, nur die Inhalte sind gleich.

Wollen Sie auf den Wert zugreifen, der sich hinter der im Zeiger gespeicherten Adresse verbirgt, so verwenden Sie den Dereferenzierungsoperator *.

```
  *pWert += 5;
  *pZahl += 8;

   std::cout << "Wert = " << Wert << std::endl;
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
Wert	0x0001	10 –> 15 –> 23
*pWert	0x0005	0x0001
*pZahl	0x0009	0x0005
...	.	.
...	.	.

Ausgabe:

```
Wert = 23
```

Man nennt das den Zeiger dereferenzieren. Im Beispiel erhalten Sie die Ausgabe Wert = 23, denn pWert und pZahl verweisen ja beide auf die Variable Wert.

Um es noch einmal hervorzuheben: Zeiger auf Integer (int) sind selbst keine Integer. Den Versuch, einer Zeigervariablen eine Zahl zuzuweisen, beantwortet der Compiler mit einer Fehlermeldung oder mindestens einer Warnung. Hier gibt es nur eine Ausnahme: die Zahl 0 darf jedem beliebigen Zeiger zugewiesen werden. Ein solcher Nullzeiger zeigt nirgendwohin. Der Versuch, ihn zu dereferenzieren, führt zu einem Laufzeitfehler.

Zeiger- Beispiel

Angabe / Beschreibung

Dieses Programm demonstriert die Anwendung von Zeigern. Weiters werden auch wieder Variablen per Wert, Referenz und Zeiger an Funktionen übergeben.

Lösung

```
/* Beispiel: Zeiger
   Filename:main.cpp
   Author:Lahmer
   Title:Verdeutlichung von Zeiger
   Description: Anhand dieses Beispiels werden Zeiger näher erläutert
   Last Change:16.01.2018
*/


//Hedader-Dateien bzw. Bibliotheken
#include <iostream>
#include <conio.h>

//Namespace
using namespace std;


//Funktionsprototyp
int diff(int a, int b); //Parameterübergabe per Wert => Rückgabewert int, 2 int-Übergabeparameter
void erhoehezahl(int &a); //Parameterübergabe per Referenz => Rückgabewert void (nichts), Variable i wird als Referenz übergeben
void verringerezahl(int *b); //Parameterübergabe per Zeiger => Rückgabewert void (nichts), Variable b wird als Zeiger übergeben


//Hauptprogramm
int main(int argc, char** argv) {

    //Lokale Variablendeklaration
    int z1=99,z2=23;
    int erg=0;
    int *ptr;
    int *ptr2;    //ptr2,ptr soll auf z2 zeigen: ptr2, ptr --> z2;
    int *ptr3;    //ptrx soll auf z1 zeigen: ptrx --> z1
```

```
    //Ausgabe der Speicheradressen der angelegten Variablen
    cout << "Adresse der Variable z1: " << &z1 << endl;
    cout << "Adresse der Variable z2: " << &z2 << endl;
    cout << "Adresse der Variable erg: " << &erg << endl;
    cout << "Adresse der Variable ptr: " << &ptr << endl;
    cout << "Adresse der Variable ptr2: " << &ptr2 << endl;
    cout << "Adresse der Variable ptr3: " << &ptr3 << endl;
    cout << endl << endl;


    //Die Adresse von der Variable z2 wird in die Variable ptr gespeichert
    ptr=&z2;

    //Der Inhalt von ptr (also die Adresse von z2) wird in die Variable ptr2
gespeichert
    ptr2=ptr;

    //Die Adresse von z1 wird in die Variable ptr3 gespeichert
    ptr3=&z1;


    cout << "Pointer1: " << endl;
    cout << &ptr << endl;    //Ausgabe der Adresse der Variable ptr
    cout << ptr << endl;    //Ausgabe des Inhalts von ptr (also die Adresse
von z2)
    cout << *ptr << endl;    //Ausgabe des Inhalts von der in ptr
gespeicherten Adresse (also der Inhalt von z2)

    cout << "Pointer2: " << endl;
    cout << &ptr2 << endl;    //Ausgabe der Adresse der Variable ptr2 (also
die Adresse von ptr2)
    cout << ptr2 << endl;    //Ausgabe der Variable ptr2 (also die Adresse
von z2)
    cout << *ptr2 << endl << endl;    //Ausgabe des Inhalts an der
gespeicherten Adresse in ptr2 (also der Inhalt von z2)

    cout << "Pointer3: " << endl;
    cout << &ptr3 << endl;    //Ausgabe der Adresse der Variable ptr3
    cout << ptr3 << endl;    //Ausgabe der Variable ptr3 (also die Adresse
von z1)
    cout << *ptr3 << endl << endl;     //Ausgabe des Inhalts an der
gespeicherten Adresse in ptr3 (also der Inhalt von z1)


    cout << "########################" << endl;


    //Funktionsaufrufe
    //Die Variablen z1 und z2 werden per Wert übergeben (=Kopie im Speicher)
    erg=diff(z1,z2);
    cout << erg << endl;
```

```
    //Die Variable z1 wird per Referenz übergeben, d.h. eine neue zweite Variable in der Funktion erhoehezahl() zeigt auf denselben Speicherplatz von z1
    erhoehezahl(z1);
    cout << z1 << endl;

    //Die Variable z2 wird per Zeiger übergeben, d.h. die Adresse von z2 wird an die Funktion verringerezahl() übergeben, die Variable b in der Funktion enthält dann die Adresse von z2
    verringerezahl(&z2);
    cout << z2 << endl;

    *ptr = *ptr*10;  //Der Inhalt an der gespeicherten Adresse in ptr (also z2) wird mit 10 multipliziert und an den Inhalt der in ptr gespeicherten Adresse (also z2) geschrieben
    *ptr3 = *ptr2;   //Der Inhalt an der gespeicherten Adresse in ptr2 (also z2) wird über den Inhalt an der gespeicherten Adresse in ptr3 geschrieben

    cout << endl << endl;
    cout << "Inhalt der Variable z1: " << z1 << endl;
    cout << "Inhalt der Variable z2: " << z2 << endl;


    //Rückgabewert = 0
    return 0;
}

//Funktionsdefinition
int diff(int a, int b)
{
    //Rückgabe der Differenz a-b
    return a-b;
}


void erhoehezahl(int &a)
{
    //Die Variable a wird um 1 erhöht, Aufgrund der Übergabe der Referenz wird auch gleichzeitig die Variable z1 im Hauptprogramm erhöht
    a++;
}

void verringerezahl(int *b)
{
    //Der Inhalt der in der Variable b gespeicherten Adresse wird um 1 verringert (d.h. z2 wird verringert)
    --*b;
    //Der Inhalt der in der Variable b gespeicherten Adresse wird um 1 verringert (d.h. z2 wird verringert)
    (*b)--;
```

```
    //Der Inhalt der in der Variable b gespeicherten Adresse wird um 1 verringert (d.h. z2 wird verringert)
    *b-=1;
}
```

Beispielhafter Auszug des Speichers

Variable	Adresse	Inhalt
z1	0x9ffe3c	99 → 100 → 200
z2	0x9ffe38	23 → 22 → 21 → 20 → 200
erg	0x9ffe34	0 → 76
ptr	0x9ffe28	0x9ffe38
ptr2	0x9ffe20	0x9ffe38
ptr3	0x9ffe18	0x9ffe3c

Ausgabe des Programms

```
Adresse der Variable z1: 0x9ffe3c
Adresse der Variable z2: 0x9ffe38
Adresse der Variable erg: 0x9ffe34
Adresse der Variable ptr: 0x9ffe28
Adresse der Variable ptr2: 0x9ffe20
Adresse der Variable ptr3: 0x9ffe18


Pointer1:
0x9ffe28
0x9ffe38
23
Pointer2:
0x9ffe20
0x9ffe38
23

Pointer3:
0x9ffe18
0x9ffe3c
99

########################
76
100
20


Inhalt der Variable z1: 200
Inhalt der Variable z2: 200
```

```
--------------------------------
Process exited after 0.07103 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Arrays

Eindimensionale Arrays

Arrays und Funktionen

Einfache Sortieralgorithmen

Sortieralgorithmen

Mehrdimensionale Arrays

Mehrdimensionale Arrays

Eindimensionale Arrays und Funktionen

Beispiel: Ermittlung von n Quicktipps im Lotto 6 aus 45 Wir beginnen mit einer Version ohne und Funktionen und bauen diese dann entsprechend um!

```
void main()
{
  int n;  //Anzahl der Tipps
  int tipp[6];
  srand(time(0));
  cout<<"Wie viele Tipps moechten Sie erhalten?";
  cin>>n;
  for(int i=1;i<=n;i++){     //n Tipps ermitteln
    cout<<"\n"<<i<<". Tipp: ";

    for(int j=0; j<6;j++){
        tipp[j]=rand()%45+1;
        cout<<tipp[j]<<" ";
    }
  }
  getch();
}
```

Bei der Verwendung von Arrays in Funktionen sind einige Besonderheiten zu beachten:

- Eine Funktion kann keinen Array-Typ besitzen.
- Werden Arrays als Parameter verwendet, so sind dies in C++ automatisch Ein-Ausgabe-Parameter. Dh. es wird im Unterprogramm keine Kopie der Array-Variablen angelegt, sondern das Unterprogramm greift direkt auf die Array-Variable der aufrufenden Funktion zu.

Wir erweitern nun das obige Beispiel um den Einsatz von Funktionen:

```
void erzeugeQuicktipp(int a[6]);

void main()
{
  int n;  //Anzahl der Tipps
  int tipp[6];
  srand(time(0));
  cout<<"Wie viele Tipps moechten Sie erhalten?";
  cin>>n;
  for(int i=1;i<=n;i++){     //n Tipps ermitteln
    cout<<"\n"<<i<<". Tipp: ";
    erzeugeQuicktipp(tipp);
  }
  getch();
}

void erzeugeQuicktipp(int a[6]){
  for (int i=0; i<6; i++) {
```

```
      a[i]=rand()%45+1;
      cout<<a[i]<<" ";
  }
}
```

Erweitere das Beispiel nun um eine eingene Funktion für die Ausgabe. Außerdem soll verhindert werden, dass innerhalb eines Tipps Zahlen doppelt vorkommen.

```
    // Programm: lotto.cpp
    // Beschreibung: Erzeugen von Lotto-Tips
    #include <iostream>      // Zusatzbibliothek für Ein-und Ausgaben wird
eingebunden
    #include <conio.h>       // Zusatzbibliothek für Konsole wird
eingebunden
    #include <stdlib.h>      // Wird für Zufallszahl-Generator benötigt
    #include <time.h>        // Wird für Initialisierung des Zufallszahl-
Generator benötigt
    using namespace std;     // Standard-Namensraum wird eingestellt

    void erzeugeQuicktipp(int a[6]);
    void ausgabe(int a[6]);

    int main(){
      int n;  //Anzahl der Tipps
      int tipp[6];
      srand(time(0));
      cout<<"Wie viele Tipps moechten Sie erhalten?"; cin>>n;
      for(int i=1;i<=n;i++){     //n Tipps ermitteln
        cout<<"\n"<<i<<". Tipp: ";
        erzeugeQuicktipp(tipp);
        ausgabe(tipp);
      }
      getch();
      return 0;
    }

    void erzeugeQuicktipp(int a[6]){
      for (int i=0; i<6; i++) {
        a[i]=rand()%45+1;
        //Überprüfe, ob Zahl bereits vorhanden
        for(int j=0;j<i;j++){   //Durchlaufe die vorhandenen Einträge
           if(a[i]==a[j]){  //und überprüfe sie auf Gleichheit mit dem
aktuellen Eintrag
            i--;        //ist die Zahl bereits vorhanden, so setze
                    //Zähler i zurück, damit diese Zufallszahl
                        //automatisch nochmals erzeugt wird!
           }
        }
      }
    }
```

```
    void ausgabe(int a[6]){
      for (int i=0; i<6; i++) {
         cout<<a[i]<<" ";
      }
    }
```

Aufgaben

- Aufgaben

Arrays, Schleifen und Funktionen - Beispiel

Angabe / Beschreibung

Der Benutzer soll eine beliebige ganzzahlige Zahl eingeben und das Programm prüft, ob diese Zahl eine Primzahl ist. Wenn ja, wird die Primzahl in einem Feld der Größe 10 gespeichert.

Lösung

```
/* Beispiel: Arrays und Schleifen
   Filename: main.cpp
   Author: Lahmer
   Title: Primzahlenüberprüfung
   Description: Der Benutzer soll eine beliebige ganzzahlige Zahl eingeben
und das Programm prüft, ob diese Zahl eine Primzahl ist. Wenn ja, wird die
Primzahl in einem Feld der Größe 10 gespeichert.
   Last Change:16.01.2018
*/
//Header-Dateien
#include <iostream>
#include <conio.h>

//Namespace
using namespace std;

//Funktionsprototyp
bool checkPrimzahl(int z);  //Rückgabewert bool, Übergabe per Wert ->
Variable z
void ausgabe(int array[10],int anzahl);   //Rückgabewert void (nichts),
Übergabe per Zeiger (array, Arrays werden immer per Zeiger übergeben) und
Übergabe per Wert (anzahl)


//Hauptprogramm
int main(int argc, char** argv) {

    //Lokale Variablendeklaration
    //Integer Array
    int feld[10];
    bool erg;
    int zahl;
    char nochmal=' ';
    int k=0;

    //Beginn der Do-While Schleife
    do {
```

```
        cout << "Geben Sie bitte eine ganzzahlige Zahl ein: ";
        cin >> zahl; //Eingabe der Zahl

        //Aufruf der Funktion checkPrimzahl
        erg=checkPrimzahl(zahl);
        cout << endl << endl;

        //Bedinung, Prüfen ob erg == true ist, wenn ja => Zahl ist Primzahl,
wenn nein => Zahl ist keine Primzahl
        if(erg==true)
        {
            cout << "Die eingegebene Zahl ist EINE Primzahl!" << endl;
            feld[k]=zahl;     //Zahl wird in das Array namens feld an die
Stelle k gespeichert
            k++;    //k wird um 1 erhöht
        }
        else
        {
            cout << "Die eingegebene Zahl ist KEINE Primzahl" << endl;
        }
        cout << "Willst du noch eine ganzzahlige Zahl eingeben? (j/n)" <<
endl;
        //Einlesen der Entscheidung, ob der Benutzer nochmals eine Zahl
eingeben will
        cin >> nochmal;
        cout << endl << "################################################"
<< endl;

    } while (nochmal=='j');  //Bedingung der do-while Schleife => Solange
der Benutzer j eingibt, wird das Programm wiederholt

    //Alle Primzahlen gesammelt ausgeben
    ausgabe(feld, k);    //Übergabe der Adresse von feld und der Anzahl der
Primzahlen (k)

    return 0;
}

//Funktionsdefinition
bool checkPrimzahl(int z)
{
    //Lokale Variablendeklaration bool = kann nur true oder false beinhalten
    bool primZ=true;

    //Prüfe alle Teiler von 2 bis z-1
    for(int i=2; i<z; i++)
    {
        //Testen ob z/i ohne Rest teilbar
        if(z%i==0)
        {
            primZ=false;
```

```
        }
    }

    //Rückgabe des booleans ob die Zahl ein Primzahl ist (true oder false)
    return primZ;
}


void ausgabe(int array[10], int anzahl)
{
    //Ausgabe aller Primzahlen
    for(int i=0; i<anzahl; i++)
    {
        cout << array[i] << endl;
    }
}
```

Ausgabe

```
Geben Sie bitte eine ganzzahlige Zahl ein: 16


Die eingegebene Zahl ist KEINE Primzahl
Willst du noch eine ganzzahlige Zahl eingeben? (j/n)
j

##################################################
Geben Sie bitte eine ganzzahlige Zahl ein: 13


Die eingegebene Zahl ist EINE Primzahl!
Willst du noch eine ganzzahlige Zahl eingeben? (j/n)
j

##################################################
Geben Sie bitte eine ganzzahlige Zahl ein: 317


Die eingegebene Zahl ist EINE Primzahl!
Willst du noch eine ganzzahlige Zahl eingeben? (j/n)
n

##################################################
13
317

--------------------------------
Process exited after 19.88 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Arrays, Funktionen, Schleifen und Zufallszahlen - Beispiel

Angabe / Beschreibung

Der Benutzer gibt die Anzahl der zu erstellenden Zufallszahlen ein und das Programm generiert die Zufallszahlen (zw. 1991 und 2018) und speichert diese in ein Array. Danach wird der Mittelwert und die Summe der zufällig generierten Zahlen berechnet.

Lösung

```
/* Beispiel: Arrays, Funktionen, Schleifen und Zufallszahlen
   Filename: main.cpp
   Author: Lahmer
   Title: Berechnung des Mittelwerts und der Summe von Zufallszahlen
   Description: Der Benutzer gibt die Anzahl der zu erstellenden
Zufallszahlen ein und das Programm generiert die Zufallszahlen (zw. 1991 und
2018) und speichert diese in ein Array. Danach wird der Mittelwert und die
Summe der zufällig generierten Zahlen berechnet.
   Last Change:16.01.2018
*/
//Header-Dateien
#include <iostream>
#include <stdlib.h>
#include <time.h>

//Namensraum
using namespace std;

//Funktionsdeklaration
//Funktion, die das Array mit Zufallswerten befüllt
void erstelleArray(int *feld, int anzahl);    //äquivalent zu void
erstelleArray(int feld[]); -> Rückgabewert void (nichts), Parameterübergabe
per Pointer (feld) und per Wert (anzahl)
void ausgabe(int feld[], int anzahl);  //Rückgabewert void (nichts),
Parameterübergabe per Pointer (feld) und per Wert (anzahl)
int summe(int feld[], int anzahl); //Rückgabewert int (ganzzahlige Zahl),
Parameterübergabe per Pointer (feld) und per Wert (anzahl)
float mittelwert(int summe, int anzahl); //Rückgabewert float
(Gleitkommazahl), Parameterübergabe per Wert (summe, anzahl)


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

```
    //Lokale Variablendeklaration
    int anz=0;

    cout << "Geben Sie bitte die Anzahl der Werte des Arrays an: ";
    //Einlesen einer ganzzahligen Zahl
    cin >> anz;

    //Lokale Variablendeklaration
    int array[anz];        //Deklaration eines Arrays namens array mit der
Größe anz
    int sum=0;
    float mittel=0.0;    //Deklaration der Variable mittel und
Initialisierung mit 0.0

    //Funktionsaufruf
    erstelleArray(&array[0],anz);   //äquivalent zu
erstelleArray(feld,anzahl);
    ausgabe(&array[0],anz);
    sum=summe(&array[0], anz);
    mittel=mittelwert(sum, anz);

    //Ausgabe des Mittelwerts
    cout << endl << "Mittelwert: " << mittel;

    return 0;
}

//Funktionsdefinition
//Erzeugt Zufallszahlen im Bereich von 1991-2018, 10-90
void erstelleArray(int *feld, int anzahl)
{
    srand(time(0)); //time(0) liefert die Sekunden seit 1.1.1970 --> dadurch
wird der Zufallsgenerator initialisiert

    for(int i=0; i<anzahl; i++)
    {
        feld[i]=rand() % 28 +1991;        //*(feld+i)=rand(0);
    }

}

//Ausgabe des Arrays
void ausgabe(int feld[], int anzahl)
{
    for(int j=0; j<anzahl; j++)
    {
        cout << feld[j] << endl;   //äquivalent zu *(feld+j);
    }
}
//Berechnet die Summe aller Zahlen im Array und gibt diese zurück
int summe(int feld[], int anzahl) {
```

```
    int sum=0;

    //Berechne die Summe aller Zahlen des Arrays
    for (int i=0; i<anzahl; i++)
    {
        sum=sum+feld[i];
    }

    //Rückgabe des Inhalts von sum an das Hauptprogramm
    return sum;

}
//Berechnet den Mittelwert aller Zahlen und gibt diesen als Gleitkommazahl
zurück
float mittelwert(int summe, int anzahl)
{
    float mittel=0.0;
    mittel=summe/anzahl;

    //Rückgabe des Inhalts von mittel an das Hauptprogramm
    return mittel;
}
```

Ausgabe

```
Geben Sie bitte die Anzahl der Werte des Arrays an: 13
2017
2002
1996
1991
2000
2015
2005
1992
2001
1993
2006
1999
2003

Mittelwert: 2001
--------------------------------
Process exited after 3.698 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Sortieralgorithmen

Sortierproblem

Das Sortierproblem besteht darin, Datensätze eines gegebenen Datenbestands sortiert anzuordnen. Im Folgenden geht es darum, das Sortierproblem und seine Relevanz in der Informatik genauer zu betrachten.

Lösung des Sortierproblems

Zur Lösung des Sortierproblems sind eine Vielzahl an Verfahren entwickelt worden. Wir werden einige dieser Verfahren hier vorstellen und zur Verdeutlichung der Komplexitätsbetrachtungen in den folgenden Abschnitten nutzen. Um die Ideen und Komplexitätsbetrachtungen möglichst einfach zu gestalten, sollen nur Zahlen anstelle komplexer Datensätze betrachtet werden.

- Grundlagen Sortieralgorithmen
- Beschreibung verschiedener Sortieralgorithmen
- Sortieralgorithmen animiert
- http://www.sorting-algorithms.com/

Angabe

Sortiere ein vorgegebenes Feld (10,6,3,5,1) per SelectionSort in einer Funktion.

Lösung

```
/* Beispiel: Funktionen, Arrays und Sortieralgorithmen
   Filename: main.cpp
   Author: Lahmer
   Title: SelectionSort
   Description: Ein vorgegebenes Feld wird per SelectionSort (Suche nach
Minimum) sortiert
   Last Change:19.01.2018
*/
//Header-Dateien
#include <iostream>

//Namensraum
using namespace std;


//Funktionsdeklaration
void Sortieren(int feld[], int size);  //kein Rückgabewert, Übergabe: Feld,
Größe des Feldes
void ausgabe(int feld[], int size);        //kein Rückgabewert, Übergabe:
Feld, Größe des Feldes

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

    int feld[5] = {10, 6, 3, 5, 1};

    //Funktionsaufruf
    Sortieren(feld, 5);                 //Aufruf der Funktion Sortieren
    ausgabe(feld, 5);                  //Aufruf der Funktion ausgabe

    return 0;
}


//Funktionsdefinition
void Sortieren(int feld[], int size){
    int position=0, hzahl;        //position=Position des Minimums,
hzahl=Hilfszahl für Zahlentausch
    for(int j=0; j<size; j++)      //Gehe von 0 bis 4 durch
    {
```

```
        position=j;    //1. Position des sortierten Teiles des Arrays (1. Durchlauf 0, 2.Durchlauf = 1; 3. Durchlauf =2; ... =4)
        for(int i=j+1; i<size; i++)   //Suche Minimum - beginne immer bei j+1 ; da ein Vergleich von feld[i]<feld[position] keinen Sinn macht, da sie im ersten Schritt auf dieselbe Zahl zeigen
        {
            if(feld[i]<feld[position])    //Schaue ob Feld[i] kleiner als das aktuelle Minimum ist
            {
                position=i;                //Wenn ja, speichere die aktuelle Position i in position
            }
        }
        hzahl=feld[j];                //speichere die Zahl an Position j in die Hilfsvariable hzahl
        feld[j] = feld[position];    //speichere das Minimum an der Position position an die Position j
        feld[position] = hzahl;        //speichere hzahl an die Position position
    }
}

//Funktionsdefinition
void ausgabe(int feld[], int size)   //Ausgabe des Feldes
{

    for(int i=0; i<size;i++)
    {
        cout << feld[i] << endl;
    }

}
```

Ausgabe

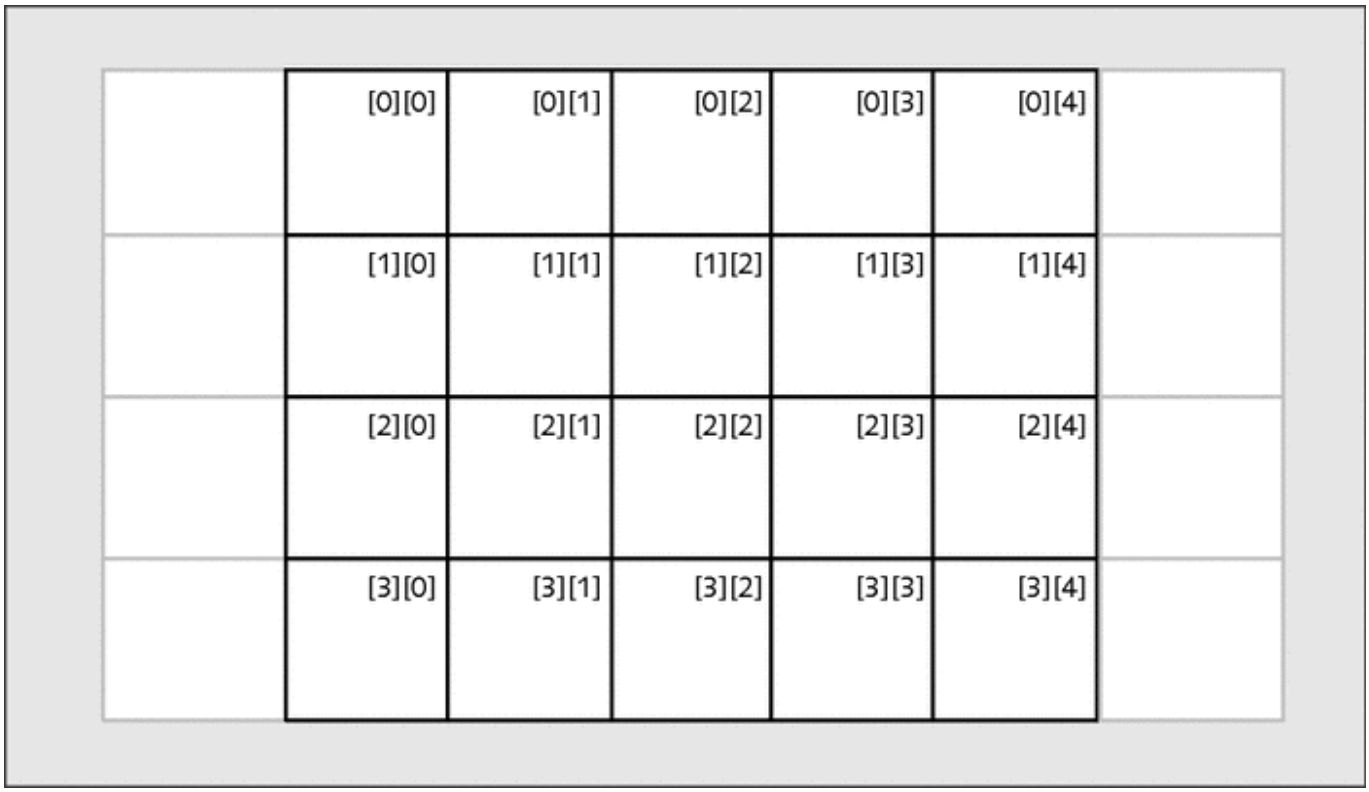
```
1
3
5
6
10

--------------------------------
Process exited after 0.08686 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Mehrdimensionale Arrays

Arrays, wie sie bisher besprochen wurden, können Sie sich als einen Strang von hintereinander aufgereihten Zahlen vorstellen. Man spricht dann von eindimensionalen Arrays oder Feldern. Es ist aber auch möglich, Arrays mit mehr als nur einer Dimension zu verwenden:

```
int Matrix[4][5];    /* Zweidimensional - 4 Zeilen x 5 Spalten */
```

Hier wurde z. B. ein zweidimensionales Array mit dem Namen Matrix definiert. Dies entspricht im Prinzip einem Array, dessen Elemente wieder Arrays sind. Sie können sich dieses Feld wie bei einer Tabellenkalkulation vorstellen.



```
/* Beispiel: Mehrdimensionale Arrays
   Filename: main.cpp
   Author: Lahmer
   Title: Schiffe versenken
   Description: Es soll eine vereinfachte Variation von Schiffe versenken
erstellt werden.
   Last Change: 19.01.2018
*/

//Header-Dateien
#include <iostream>

//Namespace
using namespace std;

//Globale Variablen
#define SPALTEN 5
#define ZEILEN 5
#define SCHIFFLAENGE 3


//Funktionsprototyp
void initializeArray(char feld[ZEILEN][SPALTEN]);  //Rückgabewert void,
Übergabe per Zeiger
void printArray(char feld[ZEILEN][SPALTEN]);   //Rückgabewert void, Übergabe
per Zeiger
int check(char feld[ZEILEN][SPALTEN], char schiff[ZEILEN][SPALTEN], int
zeile, int spalte);  //Rückgabewert int, Übergabe per Zeiger, per Zeiger,
per Wert & per Wert


//Hauptprogramm
int main(int argc, char** argv) {

    //Lokale Variablendeklaration
    //Mehrdimensionales Character-Feld für die Anzeige
    char feld[ZEILEN][SPALTEN];
    //Mehrdimensionales Character-Feld für das Schiff
    char schiff[ZEILEN][SPALTEN];
    int zeile=0, spalte=0, treffer=0;


    //Feld für das Spielfeld mit Werten initialisieren
    initializeArray(feld);

    //Feld für das Schiff  mit Werten initialisieren
    initializeArray(schiff);

    //Schiff verstecken
    schiff[1][2]='X';
    schiff[1][3]='X';
```

```
    schiff[1][4]='X';

    printArray(feld);

    //Do-While Schleife solange nicht das ganze Schiff versenkt ist
    do {

        cout << endl << "Geben Sie die Zeilennummer ein:";
        cin >> zeile;
        cout << endl << "Geben Sie die Spaltennummer ein:";
        cin >> spalte;

        //Die Treffer mitzählen, um zu wissen wann das Schiff versenkt ist
        treffer=treffer+check(feld,schiff,zeile,spalte);

        //Spielfeld ausgeben
        printArray(feld);

    }while(treffer<SCHIFFLAENGE);

    cout << endl << endl << "!!!!!!!!!!! HERZLICHEN GLUECKWUNSCH SIE HABEN
DAS SCHIFF VERSENKT !!!!!!!!!!!!!!!";

    return 0;
}

//Alle Feldelemente mit # initialisieren
void initializeArray(char feld[ZEILEN][SPALTEN]) {

    for (int i=0; i<ZEILEN;i++)
    {
        for (int j=0; j<SPALTEN;j++)
        {
            feld[i][j]='#';
        }
    }
}

//Spielfeld ausgeben
void printArray(char feld[ZEILEN][SPALTEN]) {

    cout << endl << endl << "-----------------------------------------------
--------------------";

    cout << endl << "Spielfeld: " << endl;
    cout << "   ";
    for (int i=0; i<ZEILEN;i++)
    {
        cout << i << " ";
    }
    cout << endl;
```

```
    for (int i=0; i<ZEILEN;i++)
    {
        cout << i << " ";
        for (int j=0; j<SPALTEN;j++)
        {
            cout << feld[i][j] << " ";
        }
        cout << endl;
    }
    cout  << "-----------------------------------------------------------
-----" << endl << endl;

}


//Prüfen, ob ein Schiff getroffen wurde
int check(char feld[ZEILEN][SPALTEN], char schiff[ZEILEN][SPALTEN], int
zeile, int spalte) {

    if(schiff[zeile][spalte]=='X')
    {
        cout << endl << "!!!! TREFFER !!!!!" << endl;
        feld[zeile][spalte]='X';

        return 1;
    }
    else {
        cout << endl << "!!!! KEIN TREFFER !!!!!" << endl;
        feld[zeile][spalte]='-';

        return 0;
    }

    return 0;

}
```

```
/* Beispiel: Schiffe versenken (Strukturen, Schleifen, Funktionen,
Zufallszahlen)
   Filename: main.cpp
   Author: Lahmer
   Title: Schiffe versenken
   Description: Es soll eine weitere Variation vom Spiel Schiffe versenken
erstellt werden, wo die Schiffe automatisch gesetzt werden.
   Last Change: 30.01.2018
*/


#include <iostream>
#include <time.h>       //Bibliothek für time(NULL)
#include <stdlib.h>     //Bibliothek für rand(), srand()
#include <windows.h>

using namespace std;

//GLOBALE VARIABLEN
#define ZEILEN 10
#define SPALTEN 10
#define LAENGE 3
#define RICHTUNGEN 8

void intializeArray(char array[ZEILEN][SPALTEN]);  //initialisiere Array-
Elemente mit #
void printArray(char array[ZEILEN][SPALTEN]);      //gib Array aus
int check(char Schiff[ZEILEN][SPALTEN], char Feld[ZEILEN][SPALTEN], int y,
int x); //Prüfe Array ob Schiff getroffen wurde
bool setzeSchiff(char Schiff[ZEILEN][SPALTEN], struct SchiffInfo
schiff1);//Setze Schiff zufällig

struct SchiffInfo{
    int laenge;         //Länge von 2-5
    int startX;         //Anfangsposition auf der X-Achse
    int startY;         //Anfangsposition auf der Y-Achse
    int richtung;    //0->oben, 1->rechts oben, 2->rechts, 3->rechts
unten,4->unten, 5->links unten, 6->links, 7-> links oben
};



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

    //LOKALE VARIABLEN
    char feld[ZEILEN][SPALTEN];        //Deklaration eines mehrdimensionalen
Arrays
    char schiff[ZEILEN][SPALTEN];    //Deklaration eines mehrdimensionalen
Arrays
    int zeile=0, spalte=0, zaehler=0, versuche=0;
    bool Schiffversteckt=false;
```

```
    intializeArray(feld);        //Array namens feld initialisieren
    intializeArray(schiff);        //Array namens schiff initialisieren

    struct SchiffInfo s1;
    srand(time(NULL));
    do {
        s1.laenge=rand()%5+2;        //Bestimmen der Schiffslänge
        s1.startX=rand()%SPALTEN;    //Bestimmen der Startposition auf der
X-Achse
        s1.startY=rand()%ZEILEN;    //Bestimmen der Startposition auf der Y-
Achse
        //s1.richtung=rand()%RICHTUNGEN;    //Bestimmen der Setzrichtung des
Schiffes
        s1.richtung=7;

        cout << "Laenge: "<< s1.laenge << endl;
        cout << "StartY: "<< s1.startY << endl;
        cout << "StartX: "<< s1.startX << endl;

        //Schiff verstecken
        Schiffversteckt=setzeSchiff(schiff,s1);

    }while(Schiffversteckt==false);        //Solange bis das Schiff
erfolgreich gesetzt wurde

    /*
    schiff[4][2]='X';
    schiff[3][3]='X';
    schiff[2][4]='X';
    */

    printArray(feld);        //Array feld ausgeben
    printArray(schiff);        //Array schiff ausgeben


    do{

        cout << endl << "Geben Sie bitte die Zeilennummer ein:";
        cin >> zeile;
        cout << endl << "Geben Sie bitte die Spaltennummer ein:";
        cin >> spalte;

        system("cls");        //Löscht die Konsolenausgabe

        zaehler=zaehler+check(schiff, feld, zeile, spalte);        //Prüfen
ob Schiff getroffen wurde
        versuche++;        //Zählen der Versuche

        printArray(feld);        //Array namens feld ausgeben
```

```
    }while(zaehler<s1.laenge);

    cout << endl<< "Sie haben das Schiff nach insgesamt " << versuche << "
Versuche versenkt!! GRATULATION!!" << endl;


    return 0;
}

//Initialisiert das übergebene Array mit #
void intializeArray(char array[ZEILEN][SPALTEN])
{
    for(int i=0;i<ZEILEN;i++)
    {
        for(int j=0;j<SPALTEN;j++)
        {
            array[i][j]='#';
        }
    }

}
//Gibt das übergebene Array aus
void printArray(char array[ZEILEN][SPALTEN])
{

    cout << endl << "Spielfeld: ";
    cout << endl << endl;

    cout << "   ";
    for(int i=0;i<SPALTEN;i++)
    {
        cout << i << " ";
    }
    cout << endl;
    for(int i=0;i<ZEILEN;i++)
    {
        cout << i << " ";
        for(int j=0;j<SPALTEN;j++)
        {
            if(array[i][j]=='X')
            {
                SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),4);
            }
            cout << array[i][j] << " ";
            SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),15);
        }
        cout << endl;
    }
}
//Prüft ob Schiff getroffen wurde (1) oder nicht (0)
```

```
int check(char Schiff[ZEILEN][SPALTEN], char Feld[ZEILEN][SPALTEN], int y,
int x)
{
    if(Schiff[y][x]=='X')
    {
        cout << endl << "!!!! TREFFER !!!!!" << endl;
        Feld[y][x]='X';

        return 1;
    }
    else {
        cout << endl << "!!!! KEIN TREFFER !!!!!" << endl;
        Feld[y][x]=' ';

        return 0;
    }
}

//Setze das übergebene schiff1 im Array namens Schiff, falls möglich
bool setzeSchiff(char Schiff[ZEILEN][SPALTEN], struct SchiffInfo schiff1)
{
    int zaehler=0;
    if(Schiff[schiff1.startY][schiff1.startX]!='X')
    {
        zaehler=1;
        //Prüfen der Richtung 7 = nach links oben
        if(schiff1.richtung==7)
        {
            if(schiff1.startY-(schiff1.laenge-1)>=0 && schiff1.startX-
(schiff1.laenge-1)>=0)        //Prüft ob in Y- und X-Richtung genung Platz
für das Schiff ist
            {
                //Schiff hat genug Platz
                for(int i=1;i<schiff1.laenge;i++)        //Prüft jede
Koordinate ob bereits ein Schiff vorhanden ist
                {
                    if(Schiff[schiff1.startY-i][schiff1.startX-i]!='X')
                    {
                        //cout << "Position ist frei";
                        zaehler++;
                    }
                }
            }

            //Prüfen ob Schiff genug Platz hatte
            cout << "Zaehler: " << zaehler << endl;
            if(zaehler==schiff1.laenge)
            {
                //Wenn ja, setze Schiff in Roter Farbe

                for(int i=0;i<schiff1.laenge;i++)
```

```
                {
                    Schiff[schiff1.startY-i][schiff1.startX-i]='X';

                }

                return true;

            }
            else
            {
                return false;
            }
        }

    }
    return false;

}
```

Struktur - eine zusammengesetzter Datentyp

Mit Arrays können Variablen gleichen Typs zusammengestellt werden. In der realen Welt gehören aber meist Daten unterschiedlichen Typs zusammen. So hat ein Auto einen Markennamen und eine Typbezeichnung, die als Zeichenkette unterzubringen ist. Dagegen eignet sich für Kilometerzahl und Leistung eher der Typ Integer. Für den Preis bietet sich der Typ float an. Bei bestimmten Autohändlern könnte auch double erforderlich sein. Alles zusammen beschreibt ein Auto.

Modell

Vielleicht werden Sie einwerfen, dass ein Auto noch mehr Bestandteile hat. Da gibt es Bremsscheiben, Turbolader und Scheibenwischer. Das ist in der Realität richtig. Ein Programm interessiert sich aber immer nur für bestimmte Eigenschaften, die der Programmierer mit dem Kunden zusammen festlegt. Unser Beispiel würde für einen kleinen Autohändler vielleicht schon reichen. Eine Autovermietung interessiert sich vielleicht überhaupt nicht für den Wert des Autos, aber möchte festhalten, ob es für Nichtraucher reserviert ist. Eine Werkstatt dagegen könnte sich tatsächlich für alle Teile interessieren. Ein Programm, das die Verteilung der Firmenfahrzeuge verwaltet, interessiert sich vielleicht nur für das Kennzeichen. Es entsteht also ein Modell eines Autos, das bestimmte Bestandteile enthält und andere vernachlässigt, je nachdem was das Programm benötigt. Bereits in C gab es für solche Zwecke die Struktur, die mehrere Variablen zu einer zusammenfasst. Das Schlüsselwort für die Bezeichnung solch zusammengesetzter Variablen lautet struct. Nach diesem Schlüsselwort folgt der Name des neuen Typen. In dem folgenden geschweiften Klammernblock werden die Bestandteile der neuen Struktur aufgezählt. Diese unterscheiden sich nicht von der bekannten Variablendefinition. Den Abschluss bildet ein Semikolon.

struct

Um ein Auto zu modellieren, wird ein neuer Variablentyp namens TAutoTyp geschaffen, der ein Verbund mehrerer Elemente ist.

```
struct TAutoTyp // Definiere den Typ
{
    char Marke[MaxMarke];
    char Modell[MaxModell];
    long km;
    int kW;
    float Preis;
};  // Hier vergisst man leicht das Semikolon!
```

Syntaxbeschreibung

Das Schlüsselwort struct leitet die Typdefinition ein. Es folgt der Name des neu geschaffenen Typs, hier TAutoTyp. In dem nachfolgenden geschweiften Klammerpaar werden alle Bestandteile der

Struktur nacheinander aufgeführt. Am Ende steht ein Semikolon, das man selbst als erfahrener Programmierer immer wieder einmal vergisst. Variablendefinition Damit haben wir den Datentyp TAutoTyp geschaffen. Er kann in vieler Hinsicht verwendet werden wie der Datentyp int. Sie können beispielsweise eine Variable von diesem Datentyp anlegen. Ja, Sie können sogar ein Array und einen Zeiger von diesem Datentyp definieren.

```
TAutoTyp MeinRostSammler; // Variable anlegen
TAutoTyp Fuhrpark[100];   // Array von Autos
TAutoTyp *ParkhausKarte;  // Zeiger auf ein Auto
```

Elementzugriff

Die Variable MeinRostSammler enthält nun alle Informationen, die in der Deklaration von TAutoTyp festgelegt sind. Um von der Variablen auf die Einzelteile zu kommen, wird an den Variablenname ein Punkt und daran der Name des Bestandteils gehängt.

```
// Auf die Details zugreifen
MeinRostSammler.km = 128000;
MeinRostSammler.kW = 25;
MeinRostSammler.Preis = 25000.00;
```

Zeigerzeichen

Wenn Sie über einen Zeiger auf ein Strukturelement zugreifen wollten, müssten Sie über den Stern referenzieren und dann über den Punkt auf das Element zugreifen. Da aber der Punkt vor dem Stern ausgewertet wird, müssen Sie eine Klammer um den Stern und den Zeigernamen legen.

```
TAutoTyp *ParkhausKarte = 0;      // Erst einmal keine Zuordnung
ParkhausKarte = &MeinRostSammler; // Nun zeigt sie auf ein Auto
(*ParkhausKarte).Preis = 12500;   // Preis für MeinRostSammler
```

Das mag zwar logisch sein, aber es ist weder elegant noch leicht zu merken. Zum Glück gibt es in C und C++ eine etwas hübschere Variante, über einen Zeiger auf Strukturelemente zuzugreifen. Dazu wird aus Minuszeichen und Größer-Zeichen ein Symbol zusammengesetzt, das an einen Pfeil erinnert.

```
ParkhausKarte->Preis = 12500;
L-Value
Strukturen sind L-Values. Sie können also auf der linken Seite einer
Zuweisung stehen. Andere Strukturen des gleichen Typs können ihnen
zugewiesen werden. Dabei wird die Quellvariable Bit für Bit der Zielvariable
zugewiesen.
TAutoTyp MeinNaechstesAuto, MeinTraumAuto;
MeinNaechstesAuto = MeinTraumAuto;
```

Trotzdem die beiden Strukturvariablen nach dieser Operation ganz offensichtlich gleich sind, kann man dies nicht einfach durch eine Anwendung des doppelten Gleichheitszeichen nachprüfen. Sie können bei Strukturen die Typdeklaration und die Variablendefinition zusammenfassen, indem der

Name der Variablen direkt nach der geschweiften Klammer eingetragen wird.

```
struct // hier wird kein Typ namentlich festgelegt
{
    char Marke[MaxMarke];
    char Modell[MaxModell];
    long km;
    int kW;
    float Preis;
} MeinErstesAuto, MeinTraumAuto;
```

Hier werden im Beispiel die Variablen MeinErstesAuto und MeinTraumAuto gleich mit ihrer Struktur definiert. Werden auf diese Weise gleich Variablen dieser Struktur gebildet, muss ein Name für den Typ nicht unbedingt angegeben werden. Damit ist dann natürlich keine spätere Erzeugung von Variablen dieses Typs möglich. Initialisierung Auch Strukturen lassen sich initialisieren. Dazu werden wie bei den Arrays geschweifte Klammern verwendet. Auch hier werden die Werte durch Kommata getrennt.

```
TAutoTyp JB = {"Aston Martin", "DB5", 12000, 90, 12.95};
TAutoTyp GWB = {0};
```

Objektorientierte Programmierung (OOP)

Am Anfang steht wohl die Frage was Objektorientierte Programmierung überhaupt ist. Dies ist nicht ganz einfach zu erklären und es gibt viele Definitionen die dies versuchen. Grob gesagt ist Objektorientierte Programmierung (OOP) ein Verfahren zur Strukturierung von Programmen, bei dem Programmlogik zusammen mit Zustandsinformationen (Datenstrukturen) in Einheiten, den Objekten, zusammengefasst werden. Es ist also möglich mehrere Objekte des gleichen Typs zu haben. Jedes dieser Objekte hat dann seinen individuellen Zustand. Dies bietet die Möglichkeit einer besseren Modularisierung der Programme, sowie einer höheren Wartbarkeit des Quellcodes.

Klassen

Die Klasse (class) ist die zentrale Datenstruktur in C++. Sie kapselt zusammengehörige Daten und Funktionen vom Rest des Programmes ab. Sie ist das Herz der objektorientierten Programmierung (OOP).

Klassen sind ein erweitertes Konzept von Datenstrukturen denen es erlaubt ist, außer Daten, noch Methoden zu beinhalten. Ein Objekt ist eine Instanz einer Klasse, welches sich die Eigenschaften der Klasse zunutze machen kann. Es ist möglich mehrere Objekte einer Klasse zu instanziieren, wobei jedes dieser Objekte zwar die selben Eigenschaften hat, intern aber einen ganz individuellen Zustand haben kann.

Beispiel: Zwei Instanzen der Klasse „Gegner“ werden erzeugt. Nachdem ein Gegner Schaden erlitten hat sind seine Lebenspunkte auf 50 gesunken. Die des anderen Gegners haben allerdings noch den Wert 100.

Klassen werden normalerweise mit dem Schlüsselwort **class** deklariert. Außerdem haben sie immer folgendes Format:

```
class Klassenname
{
Zugriffsspezifizierung 1:
Member 1;
...
Zugriffsspezifizierung 2:
Member 2;
...
...
};
```

Der Klassenname identifiziert die Klasse, der Objektname ist eine optionale Liste von Namen von Objekten dieser Klasse. Der Körper der Klasse wird durch geschweifte Klammern gekennzeichnet und kann verschiedene Member (Methoden, Membervariablen) beinhalten. Diesen können verschiedene Zugriffsspezifierungen zugewiesen werden.

Zugriffsrechte

public: Auf Members kann von überall zugegriffen werden von wo das Objekt sichtbar ist

private: Auf Member kann nur innerhalb anderer Member zugegriffen werden oder aus befreundeten Klassen und Funktionen

protected: Members sind verfügbar für Member der selben Klasse, befreundeten Funktionen/Klassen und Abgeleiteten Klassen

Attribute

Die Festlegung, welche Daten zu einem Typ gehören, erfolgt bei der Definition der Klasse. Die Objekte enthalten jeweils unabhängig voneinander einen Satz von Datenkomponenten (Attributen). Ihre Startwerte können durch Konstruktoren festgelegt werden. C++11 erlaubt auch die Zuweisung von Anfangswerten bei der Definition:

```
class Bruch
{
  // ...
  public:
      int z = 0, n = 1;
};
```

Elementzugriff

Im Beispiel davor wird ein Objekt vom **Typ Enemy** angelegt. Das **Objekt** enthält die **2 Variablen health (integer) und name (String)**, wie es in der Klassendefinition zu sehen ist. Um auf eine öffentliche Elementvariable zugreifen zu können, wird an den Objektnamen ein **Punkt** und dann der in der Klassendefinition verwendete Elementname gehängt. Im Beispiel wird der Name auf Andreas gesetzt.

```
    ...
    //Zugriff auf öffentliche Variable name
    strcpy(e1.name,"Andreas");
    //Zugriff auf öffentliche Variable id
    e1.id=1;
    ...
```

Klassenmethoden

Nun können Sie ein Objekt von der Klasse Enemy erzeugen. Aber was nützt Ihnen die schönste Datenstruktur in Ihrem Programm, wenn sie nicht durch Funktionen zum Leben erweckt wird? Sie werden z.B. einen Namen und Lebenspunkte eingeben und ausgeben wollen. Vielleicht wollen Sie die Lebenspunkte verringern oder erhöhen. Kurz gesagt, ein Datenverbund ist nichts wert ohne Funktionen, die auf ihn wirken. Aber die Funktionen sind auch nur im Zusammenhang mit ihrem

Datenverbund sinnvoll. Aus diesem Grund werden die Funktionen ebenso in die Klasse integriert wie die Datenelemente. Eine Funktion, die zu einer Klasse gehört, nennt man Elementfunktion oder auf englisch member function. In anderen objektorientierten Programmiersprachen spricht man auch von einer Methode oder Operation. Aufruf So, wie Sie auf Datenelemente nur über ein real existierendes Objekt, also eine Variable dieser Klasse zugreifen können, kann auch eine Funktion nur über ein Objekt aufgerufen werden. Objekt und Funktionsnamen werden dabei durch einen Punkt getrennt. Die Funktion arbeitet mit den Daten des Objekts, über das sie gerufen wurde. Aus prozeduraler Sicht könnte man es so sehen, dass eine Elementfunktion immer bereits einen Parameter mit sich trägt, nämlich das Objekt, über das sie aufgerufen wurde.

Beispiel:

```
#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

//Klasse Enemy
class Enemy {
    public:   //Zugriffsrecht public
        void setHealth(int h);
        int getHealth();
        char name[50];
                int id;
    private:  //Zugriffsrecht private
        int health;
};
//Methode setHealth zum Setzen der Lebenspunkte
void Enemy::setHealth(int h)
{
    health=h;
}

//Methode getHealth() zum Auslesen der Lebenspunkte
int Enemy::getHealth()
{
    return health;
}

//Hauptprogramm
int main()
{
    Enemy e1;   //Objekt e1 der Klasse Enemy wird erzeugt

    //Zugriff auf öffentliche Variable name
    strcpy(e1.name,"Andreas");
    //Zugriff auf öffentliche Variable id
    e1.id=1;

    //Aufruf der Methode setHealth
```

```
    e1.setHealth( 100 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";
    //Aufruf der Methode setHealth
    e1.setHealth( 50 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.n";

    return 0;
}
```

Methodendefinition innerhalb einer Klasse

Alternativ können Sie die Elementfunktion auch direkt in der Klasse definieren. Das wird leicht unübersichtlich, darum sollten Sie das nur bei sehr kurzen Funktionen tun.

```
#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

class Enemy {
    public:     //Zugriffsrecht public
                /***** INLINE - METHODENDEFINITION *****/
        //Methode setHealth zum Setzen der Lebenspunkte
        void setHealth(int h)
        {
            health=h;
        }
        //Methode getHealth() zum Auslesen der Lebenspunkte
        int getHealth()
        {
            return health;
        }

        char name[50];
                int id;


    private:     //Zugriffsrecht private
        int health;
};

//Hauptprogramm
int main()
```

```
{
    Enemy e1;   //Objekt e1 der Klasse Enemy wird erzeugt

    //Zugriff auf öffentliche Variable name
    strcpy(e1.name,"Andreas");

    //Aufruf der Methode setHealth
    e1.setHealth( 100 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";
    //Aufruf der Methode setHealth
    e1.setHealth( 50 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";

    return 0;
}
```

Aufruf

So, wie Sie auf Datenelemente nur über ein real existierendes Objekt, also eine Variable dieser Klasse zugreifen können, kann auch eine Funktion **nur über ein Objekt aufgerufen werden**.
Objekt und **Funktionsnamen** werden dabei durch einen **Punkt** getrennt. Die Funktion arbeitet mit den Daten des Objekts, über das sie gerufen wurde. Aus prozeduraler Sicht könnte man es so sehen, dass eine Elementfunktion immer bereits einen Parameter mit sich trägt, nämlich das Objekt, über das sie aufgerufen wurde.

```
    ....
    Enemy e1;   //Objekt e1 der Klasse Enemy wird erzeugt

    //Aufruf der Methode setHealth
    e1.setHealth( 100 );
    ....
```

Zusammenfassung

Die objektorientierte Programmierung hat einige neue Begriffe aufgebracht.

Objekt und Klasse

Der zentrale Begriff des Objekts bezeichnet einen Speicherbereich, der durch eine Klasse beschrieben wird. Prinzipiell kann sich der Anfänger ein Objekt als eine besondere Art einer Variablen vorstellen

und die Klasse als die Typbeschreibung. Im Buch wird ein Objekt auch hin und wieder als Variable bezeichnet, insbesondere dann, wenn sich ein Objekt an dieser Stelle wie jede andere Variable verhält.

Attribut

Die Daten-Elemente einer Klasse werden in diesem Buch meist Elementvariable genannt. In der objektorientierten Literatur findet sich dafür auch die Bezeichnung Attribut. Damit wird angedeutet, dass die Elementvariablen die Eigenschaften eines Objekts beschreiben.

Methode und Operation

Die Funktionen einer Klasse, die hier als Elementfunktionen bezeichnet werden, finden sich in der objektorientierten Literatur unter dem Namen Methode oder Operation wieder. Diese Bezeichnung bringt zum Ausdruck, dass die Funktion nur über das Objekt erreichbar und damit eine Aktion des Objekts ist.

DATEIBEHANDLUNG

Programme speichern ihre Informationen in Variablen. Leider bleiben diese immer nur bis zum nächsten Stromausfall oder Betriebssystemabsturz erhalten. Und wenn das Programm verlassen wird, ist ihr Inhalt ebenfalls Geschichte. Damit Sie auf Ihre Daten auch morgen noch kraftvoll zugreifen können, empfiehlt es sich, diese in einer Datei abzulegen. Dazu können Sie die Daten als Ausgabestrom in die Datei schreiben. Das Vorgehen entspricht dem bei der Bildschirmausgabe per cout. Diese Form wird sequenziell genannt, weil die Daten nacheinander in der Reihenfolge, wie sie geschrieben wurden, in der Datei landen. Sie können aber auch einen Datenblock an eine beliebige Stelle der Datei schreiben. Später können Sie diesen Datenblock wieder zurückholen, indem Sie den internen Dateizeiger an diese Stelle positionieren und den Datenblock wieder lesen. Diese Vorgehensweise ist typisch für Klassen, insbesondere, wenn sie in irgendeiner Form sortiert abgelegt werden sollen.

fstream

Über die Headerdatei fstream wird die Funktionalität zum Lesen und Schreiben von Dateien zur Verfügung gestellt. Soll nur geschrieben werden, dann kann auch die Headerdatei ofstream genutzt werden. Ebenso kann, wenn nur gelesen werden soll, als Header ifstream zum Einsatz kommen. Der übliche Ablauf zum Lesen oder Schreiben von Dateien ist folgender:

- Objekt zum Lesen oder Schreiben im Programm anlegen
- Objekt mit dem Dateinamen zum öffnen verknüpfen
- Text über das Objekt schreiben oder lesen
- Datei schliessen

Schreiben einer Datei

In Zeile 5 wird ein Objekt angelegt, dass ähnlich wie cout oder cin die Ausgabe in die Datei übernimmt. Der Name des Objekts kann beliebig vergeben werden, so lange dieser noch nicht verwendet wird. Die Datei beispiel.txt wird in Zeile 6 geöffnet und im aktuellen Verzeichnis angelegt, falls diese noch nicht existiert. Nun da die Datei geöffnet wurde, kann in Zeile 7 etwas in die Datei geschrieben werden. Zum Schluss muss noch die Datei geschlossen werden, damit alle gepufferten Schreibvorgänge abgeschlossen werden (Zeile 8).

```
1  #include <fstream>
2  using namespace std;
3
4  int main () {
5      ofstream fileout;
6      fileout.open("beispiel.txt");
7      fileout << "Das steht jetzt in der Datei.";
8      fileout.close();
9      return 0;
10 }
```