INHALT

Teil 1: Das Unix Betriebssystem...1

1. Historisches 1
2. Einführung und Terminologie 4
3. Ein/Ausgabesystem 6
4. Dateiverwaltung 7
5. Prozeßverwaltung 8
6. Multitasking 15
7. Shell 15

7.1. Die Shell als Kommandointerpreter 16

7.2. Die Shell als Programmiersprache 15

7.3. Shell-Skripte: 16

8. Bedienung 17

8.1. Unix-Kommandos 19

9 Die C-Shell als Programmiersprache 31

9.1. Zur Erinnerung 31

9.2. C-Shell Syntax 33

9.3. Shell-Kommandos 34

9.4 Variablen 39

9.5 Ausdrücke 41

9.6 Ein/Ausgabe 42

9.7 Erzeugen und Ausführen eines Shell-Scripts 43

9.8 Beispiele 43

Teil 2: Einführung in die Programmiersprache C 1

1. Entwicklungsgeschichte 1
2. Preprozessor 1
3. Notationsregeln 2
4. Datentypen 3
5. Deklarationen 4
6. Operationen 4
7. Kontrollstrukturen, Speicherzugriff 6

7.1. Anweisungen und Blöcke 6

7.2. Zeiger 9

8. Ein-und Ausgabe 14

8.1. Ausgabe 14

8.2. Eingabe 15

9. Programmstruktur und Funktionen 16
10. Dateioperationen 19
11. Speicherverwaltung 20
12. Übersetzen,Binden und Ausführen 20

Teil 3: Aufgaben Ü1

1 Aufgaben zum Thema Unix Ü1

2 Aufgaben zum Thema C Ü2

Teil 4. Literaturverzeichnis L1

Anhang A : vi Kommandoübersicht A1

Anhang B: EMACS Kommandos B1

Anhang C: Linux Ankündigung C1

Teil 1: Das Unix Betriebssystem

1. Historisches

Unix entstand in den Jahren 1969-73 als "privates" Betriebssystem einiger weniger Forscher und Entwickler auf einer "vergessenen" PDP 7 bei den Bell-Laboratories. Ziel des Systems war es, *eine komfortable Programmierumgebung* zu schaffen.
Die Version 1 enthielt bereits alle wichtigen Eigenschaften der heutigen Unix-Systeme, mit Ausnahme der Pipes, die ab der Version 2 implementiert waren. Das ursprünglich in Assembler geschriebene System wurde 1973 größtenteils in der ebenfalls neu entwickelten Programmiersprache C umgeschrieben.
In 1974 erschien die Version 6, die erste Version, die gegen einen geringen Kostenanteil auch an Dritte (hauptsächlich an Universitäten und Forschungszentren) abgegeben wurde.
Bei der Version 7, die 4 Jahre später erschien, war der Code nochmals überarbeitet worden, im Hinblick auf die Portabilität des Systems. Diese Version gilt als Ausgangsversion für die beiden Hauptrichtungen der heute verfügbaren Unix-Systeme: die AT&T Systeme und die **B**erkeley **S**ystems **D**istributions (BSD), aus denen sich **OSF/1** entwickelt hat.
Die Standardisierungsbestrebungen der letzten Jahren, insbesondere Posix P1003.1 (bzw. ISO 9945), haben dazu geführt, daß die Anwendungsportabilität auf Quellcode-Ebene gegeben ist.
Die noch bestehenden Unterschiede finden sich hauptsächlich in den Systemkernen; eine einheitliche Benutzerschnittstelle ist durch **OSF/Motif** (auf der Basis von X-Windows) bereits gegeben.

1.1. AT&T Unix

Der Name Unix ist ein geschütztes Warenzeichen der Fa. AT&T. Die wichtigste, auf die Version 7 des Systems I folgende Unix-Version, war das System III. Die derzeit aktuelle Version ist Unix© System V, Version 4.3. Mit dieser Variante wurde auch die Funktionalität des SUN-Solaris-Systems integriert.

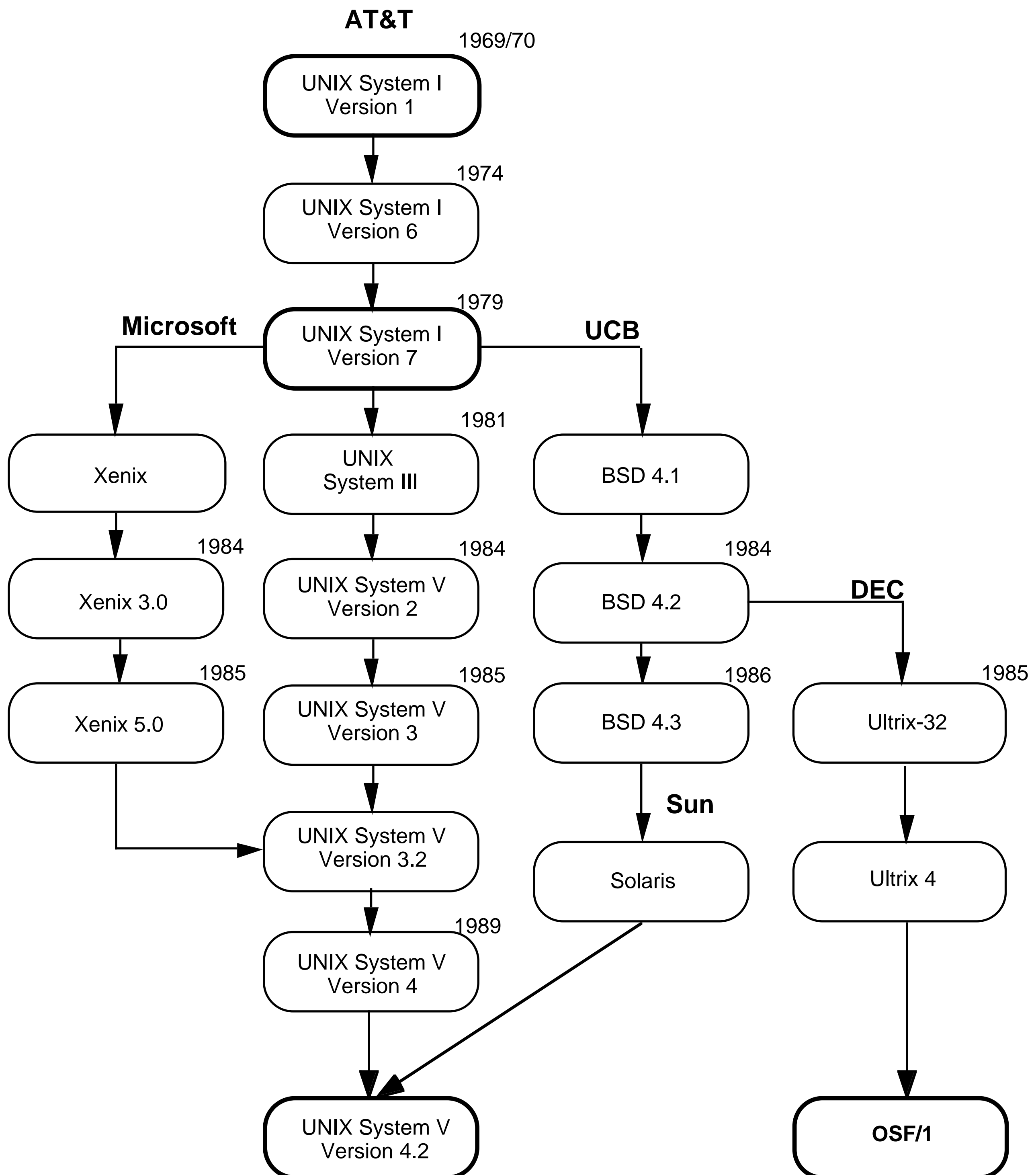


Bild 1.1 Die verschiedenen Unix -Systeme

1.2. Berkeley Systems Distribution (BSD)

Die erste Unix-Portierung auf eine VAX 11/780 (noch bei den Bell Laboratories von AT&T) wurde von den Forschern der Universität Berkeley in Kalifornien (UCB) weiterentwickelt. Diese haben einige wichtige Erweiterungen in das System eingebracht: das virtuelle Speicherkonzept, schnellere Datei-E/A, die Unterstützung der Terminal-Möglichkeiten (termcap), Netzwerkunterstützung auf der Basis des TCP/IP Protokolls etc.
Auch ein neuer Kommandointerpreter wurde in der Berkeley Version eingeführt: die C-Shell (csh). Viele dieser Verbesserungen flossen mittlerweile auch in die neuesten Versionen des AT&T Unix ein. Die DEC- und Sun-Systeme basieren auf der BSD-Version.

1.3 Minix

Im Jahr 1987 hat Andrew Tanenbaum, Professor an der Freien Universität von Amsterdam, ein Lehrbetriebssystem für PC veröffentlicht, das ohne jeden AT&T Code die Funktionalität von Unix Version 7 hat und als Quelltext für wenig Geld zu kaufen ist.
Ein Lebensnerv von Minix ist das USENET, wo in der Gruppe comp.os.minix alle Neuigkeiten, Fragen und Antworten zu Minix ausgetauscht werden. Hier werden auch Veränderungen am Betriebssystem (dem **Kernel**) veröffentlicht und gelegentlich ganze Programme verschickt.

1.4. Linux

Linux ist die PC-Variante des Unix-Systems, die durch die Initiavive des finnischen Informatikstudenten Linus Torvalds (siehe Anhang C) und der Mitarbeit von zahllosen Programmieren (kommuniziert wurde mittels Internet) entstanden ist. Die Programmquellen des Kerns stehen unter der GNU Public Licence.

Linux ist kompatibel zum POSIX.1003.1 Standard und umfaßt große Teile der Funktionalität von Unix System V und BSD.

Linux kann kostenlos aus dem Interner runtergeladen werden. Beliebt sind die sogenannte Distributionen: käufliche und relativ leicht (durch Installationsprogramme und Skripte) zu installierende Gesamtpackete auf CD-ROM oder DVD. Zu den bekannteren Distributionen zählen: SuSE , Red-Hat, Debian.

2. Einführung und Terminologie

Unix ist ein relativ kleines Betriebssystem. Seine Mächtigkeit macht sein umfangreicher Werkzeugkasten (Tools) und seine leichte Portierbarkeit aus. Eine weitere Stärke von Unix ist die Eigenschaft Ein- und Ausgabe von Programmen umzuleiten. Damit kann man umfangreiche Programme durch das Zusammenspiel von kleinen Programmen realisieren (Wiederverwendbarkeit von Software).

Unix ist ein Betriebssystem mit einem hierarchischen Dateiverwaltungssystem, einem homogenen Datentransfer und Interprozeßkommunikationsmöglichkeiten.

Die Kommunikation zwischen dem Benutzer und dem Betriebssystem erfolgt über die Shell, die als Kommandointerpreter und Programmiersprache benutzt werden kann.

Wesentliche (Linux) Eigenschaften

Multitasking: Linux unterstützt präemptives Multitasking: alle Prozesse Laufen völlig unabhängig voneinander.

Multiuser: Linux erlaubt mehreren Benutzern gleichzeitig mit dem Systen zu arbeiten.

Demand Load Executables: Es werden nur die Teile eines Programms in den Speicher geladen, die auch wirklich zur Ausführung benötigt werden.

Paging: Wird mehr Speicher benötigt als Pysikalisch vorhanden ist, muss ausgelagert werden (gerade nicht benötigte Speicherbereiche von bestimmten Prozessen werden auf Festplatte geschrieben) . Linux verwenden 4 KByte große Speicherseiten (sogenannte Pages) als Verwaltungsgrösse. Wird auf eine dieser Speicherseiten wieder zugegriffen, muß sie wieder eingelesen werden. Dieses Verfahren wird Paging genannt.

Shared Libraries: Bibliotheken sind eine Sammlung von Routinen, die ein Programm zur Abarbeitung benötigt (z.B. mathematische Funktionen, oder Ein- oder Ausgabefunktionen). Diese Routinen werden u.U. von mehrere Prozessen gleichzeitig benötigt. Da ist es naheliegend, den

Programmcode für diese Bibliotheksfunktionen nur einmal in den Speicher zu laden, und nicht für jeden Prozeß extra. Genau das ist mit Shared Libraries möglich (cfr. MS-Windows und DLLs = Dynamic Link Libraries).

Speicherschutz: Linux benutzt die Speicherschutzmechanismen der Prozessoren, um den Zugriff eines Prozesses in den Speicherbereich des Systemkerns oder den anderer Prozesse zu verhindern.

Terminologie in alphabetischer Reihenfolge:

Dispatcher: siehe Prozeßumschaltung

Event : Ein Event ist ein Ereignis. Prozesse warten auf das Eintreten von Ereignissen. Siehe Prozeßumschaltung.

Kernel : Der Kernel ist der Betriebssystemkern des UNIX Systems.

Link : Links erlauben den Zugang zu derselben Datei aus verschiedenen Verzeichnissen. Unterschiedliche Benutzer können über links auf dieselbe Datei zugreifen.

Pipe : Eine Pipe dient dem Austausch von Daten zwischen den Prozessen. Sie ist eine offene Datei, die zwei Prozesse miteinander verbindet.

Pipeline : Eine Pipeline bilden Prozesse, die linear hintereinander durch eine Pipe verbunden sind.

Prozeß : Ein Prozeß ist die Ausführung des Prozeßimages. Es wird zwischen Benutzerprozessen (Benutzerprogrammen) und Systemprozessen (Kernel Programmen) unterschieden.

Prozeßerzeugung : siehe Systemaufruf fork.

Prozeßimage : Das Prozeßimage spiegelt die bestimmte, aktuelle Umgebung wider, in der sich ein Prozeß befindet. Wärend der Ausführung eines Prozesses befindet sich dessen Image im Hauptspeicher.

Das Prozeßimage besteht aus:

- Speicherabbild des Prozesses
- Prozessor-Register-Werte
- offene Dateien
- aktuelles Verzeichnis

Prozeßterminierung : siehe Systemaufrufe exit, signal

Prozeßüberlagerung : siehe Systemaufruf exec

Prozeßumschaltung : Ein aktiver Prozeß wird deaktiviert, wenn er auf ein Event wartet. Der Prozeß, dessen Event eingetreten ist, wird aktiviert.

Scheduling : Das Scheduling erfolgt über Prioritäten. Systemprozesse haben eine höhere Priorität als Benutzerprozesse, die nach dem round robin (zirkularen) Verfahren bedient werden.

Shell : Die Shell ist der Kommandointerpreter des UNIX Systems. Sie kann auch als Programmiersprache verwendet werden um Kommandoprozeduren zu schreiben.

Shell Syntax : Kommandoname -Option [Liste von Parametern]

Synchronisation von Prozessen : Die Prozeßsynchronisation erfolgt über Ereignisse, die durch Signale angezeigt werden.

3. Ein/Ausgabesystem

Das Unix Ein/Ausgabesystem kennt zwei Klassen von E/A:

- Block-Orientiert
- Zeichen-Orientiert

und fünf Systemaufrufe für den Zugriff auf externe Daten:

- `open`
- `read`
- `write`

- `seek`
- `close`

Jeder E/A-Zugriff wird durch einen Öffnungsaufruf (`open`) eingeleitet. Gelesen und geschrieben wird ab der aktuellen Position: nach dem Öffnen zum Lesen einer Datei ist dies z.B. der Dateianfang, nach einem `seek`-Aufruf ist es die angegebene Position in der Datei. Ab da wird dann sequentiell gelesen bzw. geschrieben.

4. Dateiverwaltung

Dateien im Unix-System bestehen aus einer beliebigen Anzahl Bytes und sind normalerweise nicht strukturiert (es sei denn, der Benutzer definiert eine Struktur darauf).

Das Dateisystem hat folgende Merkmale:

- Schutz vor HW-Fehler und nicht berechtigtem Zugriff
- Geräte sind (hinter Spezialdateien) verborgen
- einheitliche Schnittstelle für jede E/A

Es werden 3 Typen von Dateien unterschieden:

gewöhnliche Dateien bestehen aus einer Folge von Zeichen. Beispiel: Textdateien, Quellcode, ausführbare Programme.

Verzeichnisse (Dateikataloge, Ordner) sind gewöhnliche Dateien bestehend aus den Datenpaaren Dateiname (Katalogname) und Dateizeiger (Katalogzeiger).

Spezialdateien (Gerätedateien) entsprechen den E/A-Geräten. Das Unix-System erlaubt dem Programmierer, auf periphere Geräte wie auf Dateien zuzugreifen, indem jedem Gerät eine Spezialdatei zugeordnet wird. Es ist dieselbe Schnittstelle wie bei gewöhnlichen Dateien verfügbar. Die Daten werden nicht im Dateisystem gespeichert, sonder direkt von dem Gerät übernommen bzw. bereitgestellt. Für jedes Gerät besteht ein Eintrag im Verzeichnis '`/dev`'.

Das Dateisystem hat eine hierarchische Struktur mit dem *Wurzelverzeichnis* '/' (*root*). Jedes Verzeichnis (Ordner, Katalog) enthält Namen der Dateien und/oder weitere Verzeichnisse (Unterordner, Unterkataloge). Der Heimatkatalog (*home directory*) jedes

Benutzers wird vom Systemverwalter erstellt. Der Benutzer kann sich weitere Kataloge und Unterkataloge einrichten.

Dateischutz

Jeder Benutzer erhält vom Administrator eine eindeutige *Benutzeridentifikation*, bestehend aus einer Benutzer- und einer Gruppennummer. Das System verwaltet diese (und andere) Benutzerinformationen in einer nur dem Administrator zugänglichen Password-Datei. Beim Einrichten einer Datei erhält diese die Benutzeridentifikation als Eigentümervermerk.

Zusätzlich gibt es für jede Datei noch neun Schutzbits für die Zugriffsberechtigung. Es wird unterschieden nach Eigentümer-, Gruppen- und "Alle-Anderen" -Zugriff, und den Zugriffsarten Lesen, Schreiben, Ausführen.

	u Eigentümer	**g** Gruppe	**o** alle Anderen
r Leseberechtigt	ja/nein	ja/nein	ja/nein
w Schreibberechtigt	ja/nein	ja/nein	ja/nein
x Ausführungsberechtigt	ja/nein	ja/nein	ja/nein

Tabelle 1: Schutzbits der Zugriffsberechtigung.

Der sogenannte *Superbenutzer* ist von sämtlichen Schutzmechanismen ausgenommen!

5. Prozeßverwaltung

Ein laufendes Unix-System besteht aus einer baumförmigen Struktur von Prozessen. Die Wurzel der Prozeßstruktur wird beim Start des Systems erzeugt (Wurzelprozeß). Auf der zweiten Stufe wird nach der Anmeldung (`login`) für jeden Benutzer ein Prozeß gestartet (Benutzer-Wurzelprozeß). Dies kann der Befehlsinterpreter (`shell`) oder ein anderes Programm sein. Auf diesen (Benutzer-Wurzel-) Prozeß baut der Benutzer seine eigene Prozeßhierarchie auf.

Im Rahmen eines Prozesses können mehrere Programme hintereinander ablaufen (Verkettung, `exec`) und ein Programm kann mehrere gleichzeitig ablaufende Prozesse erzeugen (`fork`) und kontrollieren.

Ein Prozeß ist die Ausführung seines Prozeßimages.

Jeder Prozeß läuft in einer bestimmten Umgebung ab, die das Prozeßimage widerspiegelt. Zum Prozeßimage gehört:

- das Speicherabbild
- die Prozessor-Register Werte
- die offenen Dateien
- das aktuelle Verzeichnis

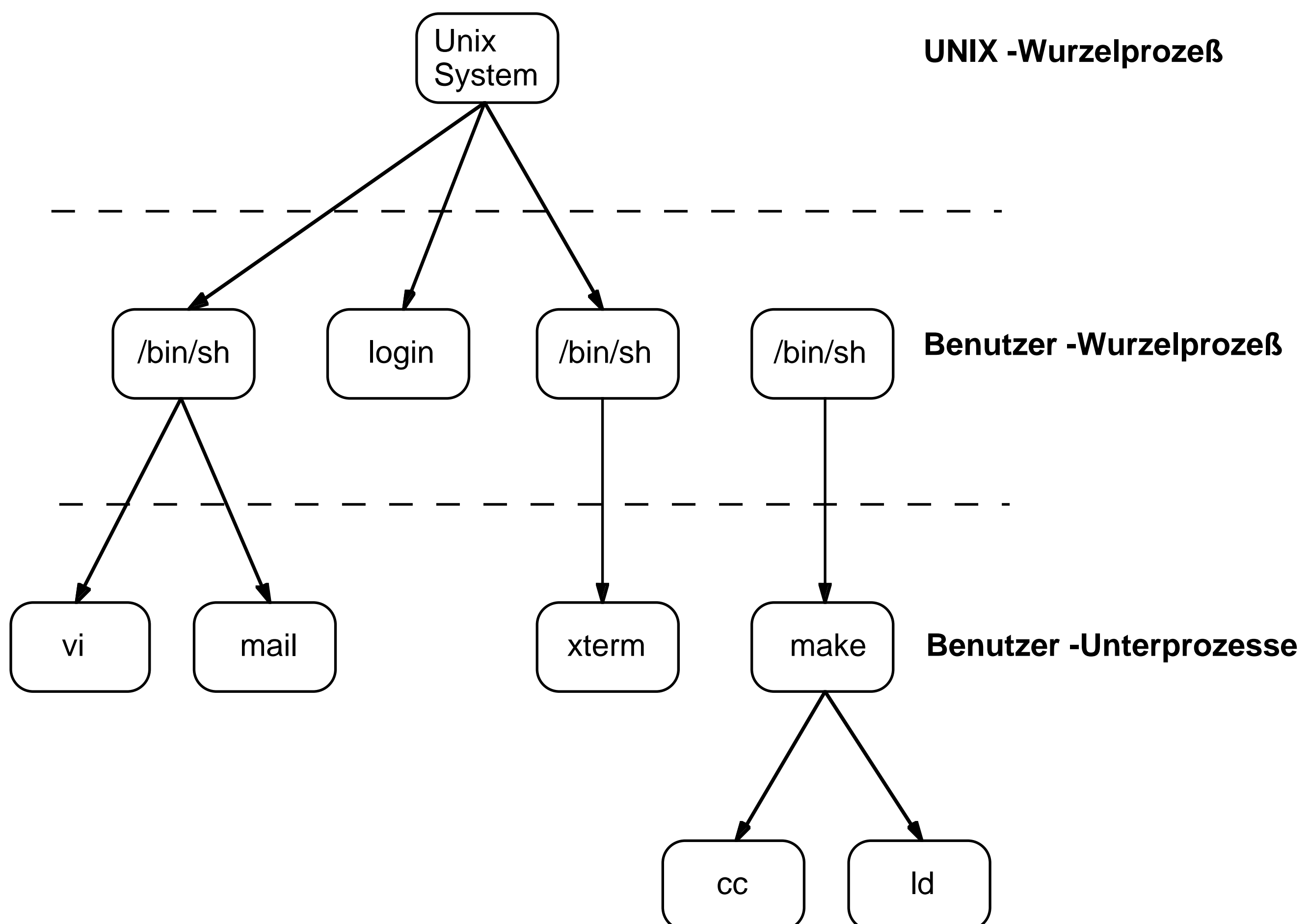


Bild 5.1: Prozeßhierarchie

Während der Ausführung eines Prozesses befindet sich dessen Prozeßimage im Hauptspeicher. Ein Prozeßimage wird ausgelagert, wenn ein Prozeß mit einer höheren Priorität diesen Speicherbereich benötigt. Das Speicherabbild eines Unix-Prozesses besteht aus drei Bereiche:

- Codebereich
- Datenbereich
- Stapelbereich

Besteht der Codebereich aus *reentrant-code*, dann kann er von mehreren Prozessen gleichzeitig benutzt werden, d.h. es gibt nur eine einzige Kopie eines gemeinsam benutzten Codesegments im Hauptspeicher. Daten- und Stapelbereich variieren während des Prozeßlaufes und wachsen aufeinander zu. Das Datensegment enthält die veränderlichen Daten eines Programms. Im Stapelsegment werden die Daten und Parameter bei Funktionsaufrufen und die Registerinhalte bei Interrupts abgelegt.

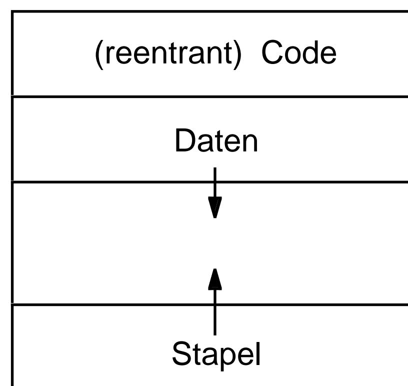


Bild 5.2: Speicherabbild eines Unix-Prozesses

Das nur lesbare Codesegment wird mit der Texttabelle (`text table`) verwaltet, in der sowohl die Adresse des Segments auf der Platte als auch die vom Hauptspeicher vermerkt ist.

Jedem Prozeß wird eine Prozeßtabelle zugeordnet in der dessen Namen, Verweise auf seine Segmente und Scheduling Information enthalten ist. Die Prozeßtabelle wird nicht ausgelagert; sie existiert solange wie der Prozeß und kann nur vom Kernel adressiert werden.

5.1. Prozeßerzeugung

Ein neuer Prozeß wird mit der Systemfunktion `fork` erzeugt. Dabei wird von dem Prozeß von dem dieser Aufruf kommt (Vaterprozeß) eine vollständige Kopie des Prozeßimages erzeugt (Kindprozeß). Der Kindprozeß erbt somit den Datenbereich, den Codebereich und die Prozeßumgebung des Vaterprozesses.

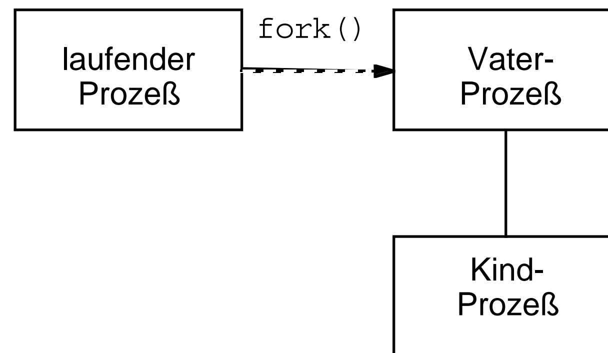


Bild 5.3 Systemfunktion `fork()`

Die fork Funktion liefert dem Vaterprozeß die Prozeßnummer des Kindprozesses, und der Kindprozeß erhält den Wert Null. Damit kann man die beiden Prozesse unterscheiden.

```
ProzeßNr = fork();
if ( ProzeßNr)
{ Bearbeite Vaterprozeß }
else
{ Bearbeite Kindprozeß };
```

Dateien, die vor dem Aufruf von fork geöffnet waren, werden von beiden Prozessen benutzt und haben einen gemeinsamen Lese/Schreibzeiger. Dies ist das Verfahren, wie Standardeingabe- und Standardausgabedateien übergeben und pipelines aufgebaut werden.

5.2. Prozeßsynchronisation

Die Prozeßsynchronisation wird durch Ereignisse (`events`) gesteuert. Prozesse können auf Ereignisse warten (`wait`) und Ereignisse erzeugen (`signal`). Durch `wait` wird die Ausführung des Aufrufers verzögert, bis er ein Signal empfangen hat oder eines der Kind-Prozesse endet.

5.3. Prozeßtabellen

Prozeßtabellen werden mit `events` assoziiert. Events werden als die Adresse der dazugehörigen Prozeßtabelle repräsentiert. Ein Vaterprozeß, der auf die Terminierung seines Kindprozesses wartet, wartet auf ein Ereignis, das durch die Adresse seiner eigenen Prozeßtabelle dargestellt wird. Ein Prozeß signalisiert seine Terminierung durch die Adresse seines Vaterprozesses. Aus der Sicht des Kernels warten alle Prozesse, bis auf

den gerade aktiven, auf ein event. Wenn dieser aktive Prozeß auf ein `event call` trifft (d.h. er muß auf ein event warten), wird der Prozeß gestartet, dessen event durch den bisher aktiven Prozeß signalisiert wurde. Dieser Vorgang heißt Prozeßumschaltung (`dispatch`).

Ein Signal wird durch verschiedene Ereignisse erzeugt. Vom Benutzer am Terminal (`quit, interrupt`), durch einen Programmfehler oder durch einen Aufruf aus einem anderen Programm (`kill`). Der Erhalt eines Signals führt normalerweise zum Abbruch des Programmes. Der Systemaufruf signal ermöglicht dem Programm das Signal zu ignorieren oder den Ablauf zu Unterbrechen und mit der Ausführung an einer anderen Stelle im Programm fortzufahren.

5.4. Prozeßverkettung: exec

Die Prozeßverkettung erfolgt mit der exec Systemfunktion, die eine Datei ausführt. Diese Funktion überlagert den aufrufenden Prozeß mit der angegebenen Datei und springt dann zum Startpunkt des Speicherimages der Datei. Es gibt keine Wiederkehr von einem erfolgreich abgesetztem exec Aufruf. Das aufrufende Image geht verloren.

5.5. Interprozeßkommunikation

Zwischen zwei Prozessen können über einen Interprozeß-Kanal Daten ausgetauscht werden. Eine pipe ist richtungsorientiert und hat zwei Enden. Ein Ende ist das lesende, das andere das schreibende. In einem Prozeß, der mittels einer pipe Daten an einen zweiten Prozeß übermitteln will, werden die Daten nicht auf dessen Standardausgabedatei geschrieben, sondern in das schreibende Ende der pipe. Der andere Prozeß, an dem die Daten gerichtet sind, bekommt seine Daten nicht aus seiner Standardeingabedatei, sondern aus dem lesenden Ende der pipe.

5.6. Prozeß-Scheduling

Das Prozeß-Scheduling erfolgt anhand der Prozeß-Prioritäten. Die Anfangspriorität eines Prozesses wird vom Kernel vergeben, der auch die Verwaltung der Ereignisse und Wartezustände übernimmt. Festplattenereignisse haben eine höhere Priorität als Benutzer- oder Terminalereignisse. Bei aktivierten Prozessen erfolgt eine Anpassung der Priorität in Abhängigkeit von der bereits verbrauchten CPU-Zeit. Prozesse, die wenig CPU-Zeit verbraucht haben, bekommen eine höhere Priorität. Systemprozesse haben eine höhere Priorität als Benutzerprozesse. Dem aktuellen Prozeß wird die CPU für eine Zeitscheibe zugewiesen. Ist der Prozeß innerhalb dieser Zeit nicht abgearbeitet,

so wird er angehalten und durch einen Prozeß mit höherer Priorität ersetzt. Prozesse mit gleicher Priorität werden nach dem zirkularen Verfahren bedient.

5.7. Auslagerung

Die Ein- und Auslagerung der Prozesse obliegt einem eigenständigen Prozeß, Swapper genannt. Ausgelagerte Prozesse werden auf der Basis 'first-out, first-in' bearbeitet, d.h. der Prozeß, der sich am längsten im Zustand Ausgelagert befindet, wird als nächster eingelagert. Eingelagerte Prozesse, die temporär ausgelagert werden sollen, werden nach deren Verweilzeit im Hauptspeicher und nach der Art der Ereignisse auf die sie warten, bestimmt.

5.8. Prozessterminierung

Prozesse können sich selbst beenden (`exit`), oder sie werden fremd beendet (`signal`). Der normale Weg einen Prozeß zu beenden ist exit. Dabei werden alle Dateien des Prozesses geschlossen, und der Vater-Prozeß wird benachrichtigt, falls dieser (mit wait) auf die Terminierung wartet.

5.9. Argumentliste

Wird ein Unix-Programm ausgeführt, dann wird dem Prozeß stets eine Argumentliste übergeben. Die Argumentliste ist eine Matrix (Array) von Zeigern auf Zeichenketten. Die Anzahl der übergebenen Argumente ist begrenzt.

Typischerweise erfolgt der Aufruf eines Programms durch eine Kommandozeilen-eingabe an einem Terminal. Wird zum Beispiel die Zeile

```
echo Guten Tag
```

an der C-Shell eingegeben, so wird das Programm `echo` ausgeführt. Diesem werden 3 Strings als Argumente übergeben: `"echo"`, `"Guten"` und `"Tag"`. Was der dann ablaufende Prozeß mit diesen 3 Argumente macht, ist seine Sache, er kann sie auch ignorieren.

Ein C-Programm erhält seine Parameterliste von der C-Initialisierungsfunktion und diese übergibt sie an die `main`-Funktion.

Die main-Funktion wird wie folgt deklariert:

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

Das erste Argument ist eine Integergröße, die die Anzahl der übergebenen Argumente festlegt. Da einem Programm mindestens ein Argument übergeben wird, nämlich sein eigener Name, ist deren Wert stehts größer oder gleich 1. Das zweite Argument ist eine Matrix (Array) von Zeigern auf Zeichenketten.

Jeder Zeiger der Matrix argv zeigt auf das jeweils erste Zeichen eines Parameterstrings. Die Anzahl der Einträge der Matrix wird durch den ersten Parameter festgelegt. Die meisten Unix Systeme fügen der Matrix ein weiteres Element argv[argc] hinzu. Dieses enthält dann den NULL-Zeiger (ANSI bzw. POSIX Norm).

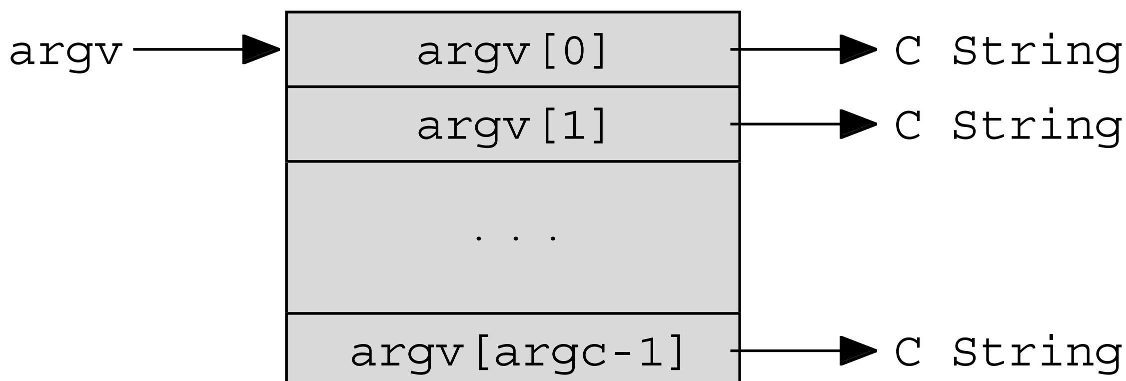


Bild 5.4: Argumente der Kommandozeile

Für das echo-Beispiel von vorhin würde das Bild dann wie folgt aussehen:

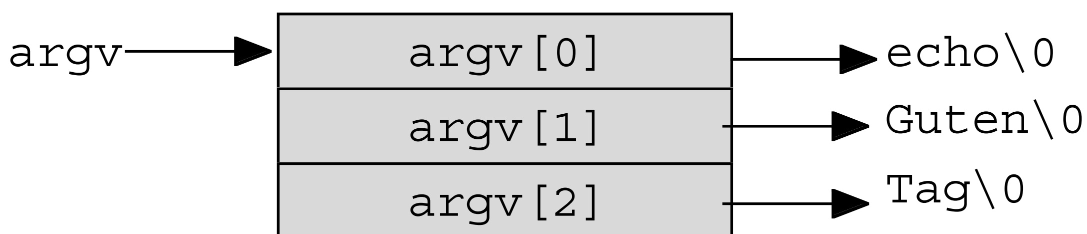


Bild 5.5: Argumente des echo Beispiels

6. Multitasking

Immer wenn ein Kommando abläuft, konkurriert es mit allen anderen Prozessen im System um die Prozessorzeit. Das Kommando `nice` informiert das System, daß eine Aktivität nicht dringend und eine niedrigere Scheduler-Priorität angemessen ist. Dies gestattet es, daß andere dringendere Arbeiten Fortschritte machen. Falls von einem Kommando oder einer Menge von Kommandos angenommen wird, daß sie einige Zeit zur Beendigung brauchen, ist es sinnvoll, die Abarbeitung in den Hintergrund zu verweisen und die Arbeit am Terminal fortzusetzen. Die shell stellt für diesen Zweck die Notation '&' bereit, womit das Kommando zur Ausführung gebracht wird, aber die shell nicht auf dessen Beendigung wartet. Damit Kommandos, die im Hintergrund ablaufen normal beendet werden und nicht etwa durch Ausloggen abgebrochen werden, gibt es das Kommando `nohup` (*no hang up*), daß den Abbruch des Jobs bei der Abtrennung des Terminals verhindert. Diese drei Möglichkeiten können miteinander kombiniert werden.

Beispiel:

```
nohup nice command &
```

das Kommando '`command`' wird mit niedrigerer Priorität (`nice`) im Hintergrund ablaufen (`&`) ohne Abbruch (`nohup`).

Wenn ein Prozeß gestartet wurde, aber nicht länger benötigt wird, kann man ihn mit dem Kommando `kill` abbrechen. Dieses Kommando benötigt die entsprechende Prozeßnummer, die durch den Befehl `ps` (= Liste aller aktiven Prozesse) erhalten werden kann. Das Kommando `at` erlaubt dem Benutzer, ein Kommando oder ein shell-Skript zu einem bestimmten Zeitpunkt zu starten.

Mit der Systemfunktion `system` kann die shell von einem Prozeß aus aufgerufen werden.

7. Shell

Die Benutzerschittstelle zu Unix bildet die *Shell*. Sie kann als Kommandointerpreter und Programmiersprache benutzt werden. Unter Ultrix™ gibt es drei Shells:

Standard-Shell (Bourne shell). Aufruf: `sh`.

C-Shell ist eine erweiterte Standard-Shell deren Syntax an die Programmiersprache C angelehnt ist. Die C-Shell wird unter Punkt 9 getrennt behandelt. Aufruf: `csh`.

TC-Shell ist an die Syntax der Kommandos des emacs-Editors anglehnt. Aufruf: `tcsh`.

Die C-Shell ist eine Erweiterung der Standard-Shell um folgende Eigenschaften:

- Dateinamenkürzel werden automatisch ergänzt
- Kommandos können Aliasnamen haben
- Gedächtnismechanismus
- Auftragskontrolle (z.B. Jobs in den Hintergrund ablaufen lassen)
- Zusätzliche eingebaute Kommandos

7.1. Die Shell als Kommandointerpreter

Nach dem Einloggen wird die Shell gestartet und ist bereit zur Entgegennahme von Kommandos. Eine Liste der Shell Kommandos befindet sich im Anhang. Die C-Shell kennt noch zusätzlich zu diesen Standard-Kommandos eine Reihe eingebauter Kommandos.

Shell Kommandos lassen sich mit einer *pipe* aneinanderfügen. Dabei wird die Ausgabe des Vorgänger-Kommandos zur Eingabe des Nachfolger-Kommandos, wobei von links nach rechts interpretiert wird. Der Operator für eine *pipe* ist '|'.

Die Shell erlaubt auch E/A-Umlenkungen von Kommandos. Mit com >file wird die Ausgabe des Kommandos in die Datei 'file' umgelenkt und mit com <file wird die Eingabe für das Kommando aus der Datei 'file' gelesen. Damit ist com1 | com2 gleichbedeutend zu com1 >file ; com2 <file.

7.5. Die Shell als Programmiersprache

Shell Programme heißen in der UNIX Terminologie Shell-Skripte.

Shell-Kommandos bilden die atomaren Anweisungen.

Shell-Variable haben als Wert eine Zeichenkette und werden durch 'set name=wert' deklariert und initialisiert. Enthält die Zeichenkette Leerzeichen, so muß sie durch Apostrophzeichen geklammert werden.

Beispiel:

```
set name1 = zeichenkette
```

oder

```
set name2 = "zeichen kette"
```

Shell-Variablen in einem Ausdruck muß '$' vorangestellt werden.

Beispiel: echo $name1 ergibt 'zeichenkette' Die Shell kennt eine ganze Reihe von vordefinierten Variablen. Shell-Variable gelten nur in der Shell, in der sie vereinbart wurden.

7.6. Shell-Skripte:

Ein Shell-Skript ist eine – Shell-Kommandos enthaltende – ausführbare Textdatei. Die erste Zeichen in einer Skript-Datei geben an, welche Shell aktiviert werden soll:

```
#!Interpreter
```

Beispiel:

```
#!/bin/csh
```

Aufruf:

```
csh file arg_1 .. arg_x
```

'csh' erzeugt eine neue C-Shell (Analog: 'sh' erzeugt eine neue Bourne-Shell)

'file' enthält das Shell-Skript

'arg_1' .. 'arg_x' sind die Positionsparameter

Shell Skripte werden ausführlich unter Abschnitt 9 (Die Shell als Programmiersprache) behandelt.

8. Bedienung

Das System stellt dem Benutzer eine Beschreibung der (ca.150) Kommandos am Terminal zur Verfügung (analog zum "help" aus anderen BS). Dazu wird das Kommando

```
man command_name (manual)
```

eingegeben. Das Nachschlagen mittels den man-Pages ist der erste Schritt des Unix-Anwenders, wenn es bezüglich Kommando-Syntax oder -Ausführung Unklarheiten gibt.

Außerdem gibt es auch noch ein interaktives Lernprogramm für Unix auf diesem System. Aufruf: `learn`.

Das aktuelle Verzeichnis heißt Arbeitskatalog (*working directory*), dessen vollständigen Pfadnamen man mit dem Kommando

```
pwd
```

(*print working directory*) erhält.

Einen neuen Katalog richtet man sich mit dem Befehl

```
mkdir dir_name
```

(*make directory*) ein. Er ist ein Unterkatalog des aktuellen Verzeichnisses. Umbenennen eines bereits existierenden Katalogs erfolgt mit

```
mv dir_name1 dir_name2
```

(*move*) und das Löschen eines (unbedingt leeren) Katalogs erfolgt mit dem Befehl

```
rmdir dir_name
```

(*remove directory*). Man wechselt von dem aktuellen Katalog in einen neuen (Arbeits-)Katalog mit

```
cd path_name
```

(*change directory*) wobei 'path_name' die Form '{/}dir_name_1/dir_name_2/...' hat.

Mit

```
cd ..
```

gelangt man jeweils eine Verzeichnistufe höher.

Der Befehl

```
ls
```

(*list*) gibt die Namen der Dateien und der Verzeichnisse des Arbeitskatalogs auf die Standardausgabe (Voreinstellung ist das Terminal) aus.

Mit

```
mv file_1 file_2
```

(*move*) benennt man die Datei mit dem Namen 'file_1' in die Datei mit dem Namen 'file_2' um und mit

```
rm file
```

(*remove*) löscht man die mit 'file' benannte Datei.

Einfache Dateihandhabung erfolgt mit dem Kommando

```
cat
```

(*catenate*). Damit kann man eine oder mehrere Dateien nacheinander auf die Standardausgabe kopieren. Ist die Standardausgabe das Terminal, so erhält man den Inhalt der Datei(en) am Bildschirm angezeigt (*cat file*).

8.1 Unix-Kommandos

Es folgt eine Kurzfassung einiger wichtiger Kommando-Beschreibungen. Es wurde in etwa die Struktur der original UNIX-Dokumentation übernommen. Diese umfaßt den Kommando-Namen, die Syntax, die Funktionsbeschreibung, Verweise, bekannte Fehler. Hier wurden die Verweise und die Fehler nicht aufgenommen, dafür wird die Funktion teilweise um Beispiele ergänzt. In der original UNIX-Dokumentation befinden sich die Kommando-Beschreibungen im Kapitel 1, wie durch der in runden Klammern gefaßten Ziffer 1 angegeben.

basename — extrahiere Dateinamen basename(1)

```
basename String [ Suffix ]
```

basename entfernt von dem String jedes Prefix das in / endet, und eine eventuell angegebene Suffix , und gibt das Ergebnis auf der stdout aus.

```
% basename main.tmp .tmp

% main
```

Entfernt die Dateierweiterung ".tmp" vom Dateinamen.

```
% find / -user root -exec basename {} \;
```

Sucht im gesamten Dateisystem alle Dateien die dem Superuser gehören, entfernt den kompletten Pfadanteil der gefundene Dateien und gibt damit den "nackten" Dateinamen auf dem Bildschirm aus (siehe auch find-Kommando).

```
% find . -name *.tmp -exec basename {} .tmp \;
```

Bitte beschreiben Sie selbst die Funktion der oben stehende Kommandozeile.

cal — Kalender Ausgabe cal(1)

```
cal [ Monat ] Jahr
```

cal erzeugt einen Kalender für das angegebene Jahr. Wird zusätzlich ein Monat angegeben, dann wird nur ein Kalender für diesen Monat erzeugt. Als Monat ist eine Zahl 1..12 anzugeben, für Jahr ist der Bereich 1..9999 zulässig!

```
% cal 12 88
```

Erzeugt den Kalender des Monats Dezember im Jahre 88, nicht 1988!

cat — Ausgabe des Datei-Inhaltes cat(1)

```
cat [ -nutTv ] Datei . . .
```

cat liest hintereinander jede Datei und gibt sie auf der Standard-Ausgabe aus. Wird keine **Datei** spezifiziert, oder falls das Argument — angegeben ist, wird von der Standard-Eingabe gelesen. Die **-u** Option erzwingt eine ungepufferte Ausgabe (OSF). Die **-n** Option zählt die Zeilen, **-T** zeigt Tabulatorzeichen, und **-v** zeigt nichtdruckbare Zeichen.

```
% cat *
```

Listet die Inhalte von sämtlichen Dateien im aktuellen Verzeichnis.

cc — C Kompiler starten cc(1)

```
cc [ Option ] Datei . . .
```

`cc` startet den C Compiler. Argumente die in .c enden, werden als C Quellprogramme aufgefaßt und übersetzt. Die erzeugten Objektdateien erhalten den gleichen Namensteil, mit .o als Extension.
Einige Optionen:

-w	unterdrücke Warnungen.
-O	Optimierung aktiv.
-S	Compiliere die angegebenen C Programme, schreibe aber den erzeugten Assembler-Code in eine gleichnamige Datei mit dem .s Suffix
-c	Nur übersetzen, nicht binden.
-o Output	Benenne das ausführbare Programm Output. Sonst wird es a.out genannt.
-Dname=def	Definiere name für den Preprozessor, wie mit #define. Ist kein def angegeben, wird name = 1 definiert.

```
cc -O -o test test.c
```

Compiliere den C Quellcode in der Datei `test.c` mit Optimierung, und erzeuge als Ausgabe die Datei `test`.

chmod — Ändere Zugriffsrechte chmod(1)

```
chmode Modus Datei . . .
```

`chmod` ändert die Zugriffsrechte jeder angegebenen Datei. Die Modus-Angabe kann absolut oder symbolisch sein. Der absolute Modus ist eine oktale Zahl der mit dem bestehenden Modus gesetzt wird. So erlaubt der Modus 600 Lese- (400) und Schreibzugriff (200) für den Besitzer.

Ein symbolischer Modus hat die Form:

[wer] op erlaubnis [op erlaubnis]

wer kann enthalten:

u für user (Eigentümer),

g für group (Gruppe) und

o für other (Andere).

op ist ein Operator:

+ zum hinzufügen einer Erlaubnis,

- fürs Entfernen, und

= fürs absolute Zuteilen (alle nicht identifizierten Bits werden gelöscht).

Erlaubnis ist jede mögliche Kombination der Zeichen r (read, lesen), w (write, schreiben), x (execute, ausführen), s (set, setze User-id oder Gruppe-id). Nur der Eigentümer einer Datei und der Superuser können den Modus der Datei veränderen.

```
chmod o+r *.h
```

Setzt die Lese-Erlaubnis aller Benutzer für alle Dateien mit der .h Extension.

cmp — Vergleiche zwei Dateien cmp(1)

```
cmp [ -l ] [ -s ] Datei1 Datei2
```

`cmp` vergleicht die beiden Dateien. Werden Unterschiede gefunden, dann werden die Byte- und die Zeilennummer ausgegeben, bei der ein Unterschied festgestellt wurde. Die **-l** Option zeigt zusätzlich die unterschiedlichen Bytes im oktalen Format, die **-s** Option gibt keine Information aus, sie erzeugt nur eine Anzeige.

```
cmp tst1.c tst2.c | wc
```

Die beiden Dateien `tst1.c` und `tst2.c` werden verglichen, und die Anzahl der Unterschiede gezählt.

cp — Kopiere cp(1)

```
cp Datei1 Datei2
```

`cp Datei1 . . . Verzeichnis`

`Datei1` wird nach `Datei2` kopiert, oder, in der zweiten Syntax, werden ein oder mehrere Dateien in das angegebene `Verzeichnis` kopiert. Eine Datei kann nicht in sich selbst kopiert werden.

```
cp Quelle.c Ziel.c
```

csh — Die C-Shell csh(1)

Siehe 9.

date — Datumsausgabe date(1)

`date`

Tagesdatum und Uhrzeit werden ausgegeben.

df — Zeige Festplattenbelegung df(1)

`df [ Dateisystem ]`

`df` gibt die Anzahl freier Blöcke des **Dateisystem** aus. Wird kein Dateisystem genannt, dann wird der freier Platz aller normalerweise installierten Dateisysteme ausgegeben.

```
df /dev/rc0a
```

diff — Zeige Dateiunterschiede an diff(1)

`diff [ Option ] Datei1 Datei2`

`diff` zeigt die zu veränderenden Zeilen der beiden Dateien an, um die Dateiinhalte zu egalisieren. Die `Option` beeinflußt das Ausgabeformat:

- e erzeuge ein sed-Script, das datei2 aus datei1 erzeugt.
- h schneller, ungenauer Durchlauf.
- b ignoriere folgende Leer- und Tabulatorzeichen.

echo — Echo der Argumente echo(1)

`echo [ -n ] [ arg ] . . .`

Zeilenweise Ausgabe der Kommandoargumente auf Standardausgabe. Die **-n** Option unterdrückt die zeilenweise Ausgabe.

```
echo Dies ist eine Zeile

Dies

ist

eine

Zeile
```

ed — Starte den Text-Editor ed ed(1)

`ed [ - ] [ -x ] [ Name ]`

`ed` ist der etwas veraltete (zeilenorientierte) Standard-Text-Editor.

find — Suche Dateien find(1)

`find Pfadnameliste Ausdruck`

`find` bearbeitet rekursiv jeden Pfadnamen der `Pfadnameliste`, auf der Suche nach Dateien, die dem angegebenen boolschen `ausdruck` genügen. Im boolschen Ausdruck sind u.a. folgende Angaben erlaubt:

-name Datei	Ergibt logisch WAHR, wenn der akuelle Dateiname gleich Datei ist.
-type t	Ergibt WAHR, wenn der aktuelle Dateityp t (steht für b, c, d, oder f mit resp. Bedeutung Block-spezial, Character-special, Verzeichnis oder normale Datei) ist.
-user name	WAHR, wenn der Datei-Eigentümer der Benutzer name ist.
-atime n	WAHR, wenn auf die Datei in den letzten n Tage zugegriffen wurde.
-mtime n	WAHR, wenn die Datei in den letzten n Tage modifiziert wurde.
-exec cmd ;	WAHR, wenn das ausgeführte cmd (Kommando) den Wert 0 zurückgibt. Das Kommando muß mit einem ; abgeschlossen werden, das, um nicht durch die shell interpretiert zu werden, normalerweise mit einem 'backslash' eingeleitet wird. Das Kommando-Argument {} wird durch den akuellen Pfadnamen ersetzt.
-print	immer WAHR; der akuelle Pfadname wird ausgegeben.

Die logischen Angaben können mit nachstehenden Operatoren kombiniert werden:

(. . .)	Klammerung einer Gruppe von Primitiven und Operatoren.
!p	! ist der unärer NICHT-Operator.
p p	Konkatenation.
p -o p	Alternation (-o ist der ODER-Operator).

fgrep — Durchsuche Datei nach einem Muster fgrep(1)

`fgrep [ Option . . . ] Muster [ Datei ] . . .`

Durchsucht eine Datei nach einem angegebenen `Muster`. Jede Zeile, die das Muster enthält wird ausgegeben. Muster sind auf Strings beschränkt.

Optionen beeinflussen die Ausgabe oder die Suche.

-c	nur die Anzahl der Treffer wird Ausgegeben.
-h	unterdrücke den Dateinamen bei der Ausgabe.
-i	ignoriere Groß-/Kleinschreibung.
-v	Alle Zeilen, außer die mit Treffer werden ausgegeben.
-x	Nur identische Zeilen werden ausgegeben .

Varianten dieses Kommandos (**grep** und **egrep**) erlauben die Angabe von Ausdrücken an der Stelle des einfachen Textmusters, und sind damit wesentlich leistungsfähiger aber dafür auch langsamer in der Ausführung.

kill — Beende eine Prozeß kill(8)

`kill [ -sig ] prozessid`

Sendet das Signal `sig` zum angegebenen Prozeß. Kein `sig` Argument impliziert das SIGTERM Signal (terminiere).

login — Anmelden beim System login(1)

`login [ Benutzername ]`

ls — Liste Inhalt eines Verzeichnisses ls(1)

`ls [ option ] name . . .`

Für jedes genannte Verzeichnis werden die darin gespeicherten Dateien aufgelistet; für jede genannte Datei wird der Namen mit der zusätzlich angeforderten Information

(**option**) gelistet. Wird **name** nicht aufgeführt, wird das aktuelle Verzeichnis aufgelistet. Einige options:

- -l Ausgabe im langen Format.
- -t Sortiere nach der zuletzt modifizierten Zeit.
- -a Alle Einträge auflisten.
- -r Umgekehrte Sortier-Ordnung.
- -g Zeige Gruppe statt Eigentümer bei der -l Option.

```
ls -al
```

Liste (im ausführlichen Format) alle Einträge im aktuellen Verzeichnis.

mail — Elektronische Post bearbeiten mail(1)

```
mail [ -r ]
```

Mail ohne Argument listet die eingetroffenen Nachrichten hintereinander, die letzte Nachricht zuerst; die `-r` Option sorgt für die umgekehrte Ordnung.

make — Verwaltung von Projekten make(1)

```
make [ -f Makefile ] [ option ] [ namen ]
```

Das `make`-Programm führt die in einem sogenannten Makefile enthaltenen Kommandos aus, um eine Aktualisierung des Projektes zu erreichen (das Datum der letzten Aktualisierung der einzelnen Dateien wird als Kriterium herangezogen). Dabei wird unter Projekt eine Sammlung von Quell- und Objektdateien verstanden, die voneinander abhängig sein können. Eben diese Abhängigkeit wird in einem Makefile definiert. Im unten stehenden Beispiel ist das Programm pgm abhängig von den beiden Objectfiles a.o and b.o, die selbst von ihren Quellfiles (a.c und b.c) und von einem gemeinsamen File incl.h abhängig sind:

```
pgm: a.o b.o
      cc a.o b.o -o pgm    # Kompilierkommando
     incl.h a.c
       cc -c a.c
```

```
incl.h b.c
   cc -c b.c
```

Make wird das Zielprogramm pgm in diesem Beispiel nur dann neu erzeugen, wenn zumindest eines der Teile, von denen es abhaengig ist, neueren Datums ist als pgm selbst, bzw. wenn pgm noch nicht existiert (als Kommando wäre `make pgm` einzugeben).

Kommentare werden durch # eingeleitet und beendet durch Zeilenende.

Wird das Makefile nicht explizit genannt (Parameter -f), dann sucht Make nach Makefiles mit dem Namen `makefile` und `Makefile`, in dieser Reihenfolge.

mkdir — Verzeichnis einrichten mkdir(1)

`mkdir name . . .`

Die Standardeinträge **.** und **..** werden automatisch miterzeugt.

mcopy — Kopieren von DOS-Dateien mcopy(1)

`mcopy -t Quelldatei Zieldatei`

`mcopy -t Quelldatei [Quelldateien...] Zielverzeichnis`

`mcopy -t DOS-Quelldatei`

Kopiert DOS-Dateien zu und von einem Unix-System. Die Angabe eines Laufwerksbuchstaben (z.B. a:) bestimmt die Übertragungsrichtung. Eine fehlende Laufwerkskennung impliziert eine Unix-Dateiangabe. Wird ein Laufwerk ohne Dateiangabe als Quelle genannt, werden alle Dateien von diesem Laufwerk kopiert.

Wird nur eine einzelne DOS-Quelldatei genannt (dritte Form), ohne Angabe eines Ziels (Beispiel "`mcopy a:main.c` "), wird diese im aktuellen Unixverzeichnis ("**.**") abgespeichert.

Die **-t** Option kennzeichnet eine Textdateiübertragung. Das Zeilenende wird dem Zielsystem entsprechend umgewandelt.

mv — Datei umbenennen oder umkopieren mv(1)

`mv datei1 datei2`

`mv datei . . . Verzeichnis`

In der ersten Version wird `datei1` in `datei2` umbenannt. Die zweite Version transferiert die genannten Dateien in das `Verzeichnis`.

nice — Starte ein Programm mit niederer Priorität nice(1)

`nice [ -wert ] Kommando [ Argument . . . ]`

Ist `wert` angegeben, wird der Prioritätswert entsprechend erhöht (=niederer Priorität).

od — Oktaler Dump od(1)

`od [ Option ] [ Datei ]`

Ausgabe von `Datei` in einen oder mehrere (`Option`) Formate. Wird `Datei` nicht angegeben, wird der Standard-Input genommen.

- -b Bytes oktal interpretieren.
- -c Bytes als ASCII interpretieren.
- -d 16 Bit Worte dezimal interpretieren.
- -o 16 Bit Worte oktal interpretieren (Voreinstellung).
- -x 16 Bit Worte hexadezimal interpretieren.
- -D 32 Bit Worte dezimal interpretieren.
- -O 32 Bit Worte oktal interpretieren.
- -X 32 Bit Worte hexadezimal interpretieren.

ps — Prozeß-Status anzeigen ps(1)

`ps [ option ]`

Standardmäßig werden nur eigene Prozesse gelistet. Es sind, je nach System, verschiedene Syntax-Angaben möglich bzw. erlaubt (siehe man-Page des jeweiligen Systems).

- -a Alle Prozesse die Terminals betreffen auflisten.
- -l Langes (ausführliches) Listen.
- -x Auch solche Prozesse, die keinem Terminal zugeordnet sind.

pwd — Zeige aktuelles Verzeichnis pwd(1)

```
pwd
```

rm,rmdir — Datei oder Verzeichnis löschen rm(1)

```
rm [ option ] datei . . .
```

```
rmdir verzeichnis
```

`rm` löscht die genannten Dateien; ist als `option -r` angegeben, werden auch Verzeichnisse (rekursiv) gelöscht. Mit der `-i` Option wird explizit nachgefragt, ob eine bestimmte Datei gelöscht werden soll.

sh — Bourne shell Starten sh(1)

Siehe 7.

talk — Mit anderen Benutzer kommunizieren talk(1)

```
talk user@host terminal
```

Der gerufene Benutzer wird mit der Nachricht

```
Message from TalkDaemon@his_machine...
   talk: connection requested by your_name@your_machine.
   talk: respond with: talk your_name@your_machine
```

auf den Kommunikationswunsch aufmerksam gemacht, und sollte mit

```
   talk  your_name@your_machine
```

antworten um die Kommunikation zu starten. Nach dem Talk-Kommando wird das Terminalfenster in zwei Bereichen unterteilt: der obere ist für die Darstellung Ihrer eingegebenen Textzeilen reserviert, der untere für die Antworten des "angesprochenen" Benutzers user. So kann auf einfache weise eine geordnete Kommunikation stattfinden. Beendet wird diese Kommunikation durch Eingabe des Interrupt-Zeichens.

time — Zeitmessung der Kommando-Ausführung time(1)

```
time kommando
```

Das `kommando` wird ausgeführt, und nach seiner Terminierung wird die verstrichene Zeit, aufgeschlüsselt nach System, Nutzer, und E/A, ausgegeben.

touch — Zeit der letzten Modifikation setzen touch(1)

```
touch [ -acm ]  datei . . .
```

touch versucht für jede genannte `datei` die aktuelle Zeit zu setzen. Die **-c** Option verhindert das Einrichten einer nicht vorhandenen, aber genannten Datei. Mit **-a** wird die Zeit des letzten Zugriffs (**a**ccess), mit **-m** die Zeit der letzten Änderung (**m**odification) neu gesetzt.

vi — Bildschirm-orientierter Editor vi(1)

```
vi  [ -r ] name . . .
```

Die `-r` Option erlaubt eine Datei zurückzugewinnen, die während eines Systemcrashes editiert wurde.

Die vi-Kommando-Übersicht finden Sie im Anhang.

wc — Worte Zählen wc(1)

```
wc [ -lwc ] datei . . .
```

Zählt die Zeilen, Worte, und Zeichen genannter Dateien oder des Standard-Input, wenn die Datei-Angabe fehlt. Die Option erlaubt die Auswahl, ob Zeilen (**l**), Worte (**w**), und/oder Zeichen (**c**) gezählt werden (**-lwc** ist default).

who — Wer ist beim System angemeldet (eingelogt)? who(1)

```
who [ am i]

    who am i
```

Unter welchen Namen bin ich angemeldet?

write — Einen anderen Benutzer Zeilen senden write(1)

```
write user [terminal]
```

Write sendet den an Ihrem Terminal eingegebenen Text zeilenweise an den mit user genannten Benutzer. EOF beendet das Kommando. Der Empfänger kann seinerseits mit write antworten (siehe auch talk).

9. Die C-Shell als Programmiersprache

Die Shell hat ihre eigene eingebaute Programmiersprache. Die Sprache wird interpretiert, d.h., die Shell analysiert die Kommandos und führt sie von Zeile zu Zeile aus. Zum Leistungsumfang gehören:

• Speichern von Daten in Variablen

• Testen von Bedingungen; wie in **if** Anweisungen

• Wiederholtes Ausführen von Kommandos, wie in **for** oder **while** Schleifen

• Einem Programm Argumente übergeben

9.1. Zur Erinnerung

Die Shell ist ein interaktives Programm; interpretativ bearbeitet es die vom Benutzer eingegebene Kommandos oder Kommandoprozeduren (sogenannte Shell-Scripts = Dateien die Shell-Kommandos enthalten).

9.1.1. Die Shell ist ein Kommandointerpreter

Nachdem der Benutzer sich beim System angemeldet hat (**login**), wird eine bestimmte Shell (es gibt unter Unix mehrere Shells: z.B. die Bourne-, C-, und Korn-Shell – wir werden uns hauptsächlich mit der C-Shell (**csh**) befassen), aktiviert (der Shell-Prozess läuft).

Die interpretative Arbeit der Shell läuft wie folgt:

• Die Shell gibt ein Bereitschaftszeichen (**prompt**) aus, und wartet auf ein (einzutippendes) Kommando.

• Sie Tippen eine Kommandozeile ein, und beenden die Zeile mit der return-Taste

• Die Shell analysiert die Kommandozeile und sucht das angeforderte Programm (sie interpretiert das erste Wort der Zeile als ein Kommando) bzw. führt das Kommando aus wenn es sich um ein eingebautes d.h. Shell-eigenes Kommando handelt (z.B. **cd** oder **pwd** sind eingebaute Kommandos). Im letzten Fall zeigt die Shell durch ausgabe ihres Promptes an, daß das Kommando abgearbeitet wurde.

• Die Shell fordert das System zur Abarbeitung des Programms auf (das dann wie ein normaler Unix-Prozeß abläuft), oder gibt eine Fehlermeldung aus.

• Ist der entsprechende Prozeß abgearbeitet, übernimmt die Shell wieder und gibt erneut ihr Bereitschaftszeichen aus.

9.1.2. Der Prozeß

Ein in den Speicher geladenes und ablaufendes Programm, ist ein Prozeß. Jeder Unix-Prozeß wird eindeutig über seinen Prozeß-Identifier (PID) identifiziert. Beim Start eines Prozesses werden immer drei sogenannte Standard-Dateien geöffnet:

Standardeingabe (stdin)

Von der Standardeingabe erwartet der Prozeß seine Eingaben; die Vor–einstellung ist die Tastatur.

Standardausgabe (stdout)

Ausgaben gehen auf die Standardausgabe; Voreinstellung ist der Bildschirm.

Standardfehlermeldung (stderr)

Fehlermeldungen gehen an diesen Ausgabekanal; Voreinstellung ist ebenfalls der Bildschirm.

9.1.3. Input/output Umleitung

Die Shell unterstützt das Umdirigieren der Eingabe- , Ausgabe-, und Fehlerkanäle in eine Datei. Handelt es sich hierbei um die Ausgabe, dann kann entweder in eine neue Datei geschrieben oder an einer bestehenden angehängt werden. Die Syntax ist:

Kommando > Ausgabedatei # der Inhalt der Datei wird überschrieben
Kommando >> Ausgabedatei # die Ausgabe wird an die Datei angehängt
Kommando < Eingabedatei
Kommando < Eingabe > Ausgabedatei
Kommando > Ausgabedatei

9.1.4. Pipes

Durch die Eigenschaft der Pipes erlaubt die Shell es auch, die Standardausgabe eines Prozesses zur Standardeingabe eines nachfolgenden Prozesses zu machen: so können auch mehrere Prozesse miteinander verkettet werden. Die Syntax:

Kommando1 **|** *Kommando2* **|** *...* **|** *KommandoN*

Eine solche Zeile wird auch Pipeline genannt.

Beispiel:

```
ypcat passwd | fgrep bprak | sort -t: +2n            # listet
alle Zeilen der passwd-Datei die den Text "bprak" beinhalten,
sortiert nach der Benutzer-ID (3. Feld des passwd-Eintrags)
```

9.1.5. Shell-Umgebung

Der Shell-Prozess wird automatisch durch die login-Prozedur aktiviert. Dieser Prozess hat eine characteristische (Arbeits-)Umgebung (**environment**), die durch die Werte, die den sogenannten Umgebungsvariablen zugewiesen werden bedingt ist.

Login-Scripts

Ein login-Script ist eine Kommandodatei die hilft die gewünschte Programmier-umgebung aufzubauen. Es gibt zwei Typen:

- ein Systemscript für alle Benutzer einer bestimmten Shell
- lokale (Benutzer-)Prozeduren in dem Heimverzeichnis des Benutzers

Die lokale Prozeduren der csh heißen :

.cshrc und **.login** ; beide werden beim Anmelden (login) ausgeführt. Zusätzlich werden die in **rc** (Bedeutung: **r**un **c**ommand) endenden Scripts jedesmal aufgerufen, wenn die aktuelle Shell eine neue Sub-Shell aktiviert.

9.2. C-Shell Syntax

csh [-cefnstvxVX] *[Kommandodatei]* *[Argumentenliste]*

Die Optionen haben folgende Wirkung:

-c Lese Kommandos im nachfolgend genannten Argument. Die restlichen Argumente werden in *argv* abgelegt.

-e Die Shell wird beendet, sollte das angestoßene Kommando fehlerhaft abbrechen oder einen Status von null verschieden zurückgeben.

-f Unterdrückt die Ausführung des .cshrc Scripts im Heimverzeichnis.

-n Kommandos werden nur analysiert (parsing), nicht ausgeführt. Dies ist Sinnvoll für die Syntaxüberprüfung von Shell-Prozeduren. Alle Substitutionen werden ausgeführt (history, command, alias, etc.)

-s Kommandoeingabe von der Standardeingabe.

-t Eine einzelne Zeile wird gelesen und ausgeführt.

-v Setzt die Shellvariable *verbose* : Mitschreibemodus ist aktiv.

-V Die *verbose* Variable wird vor der Ausführung von .cshrc gesetzt.

-X Die *echo* Variable wird vor der Ausführung von .cshrc gesetzt.

Nach Interpretation dieser Optionen wird, vorausgesetzt es folgen noch Argumente, und die **-c, -s** oder **-t** Option wurden nicht angegeben, das nächste Argument als Name einer einzulesenden und auszuführenden Kommandoprozedur angenommen.

9.3. C-Shell-Kommandos

Ein einfaches Kommando besteht aus eine Anzahl von Worten, wobei das erste Wort das Kommando nennt, das ausgeführt werden soll.

Es gibt die Möglichkeit Pipelines durch **||** bzw. **&&** zu verbinden; dann wird, entsprechend der Syntax der C-Sprache, die zweite Pipeline nur ausgeführt, wenn die erste mit Anzeigen abgebrochen wird, bzw. (im zweiten Fall) wenn die erste anzeigenlos durchlaufen wird.

9.3.1. Eingebaute Kommandos

Eingebaute Kommandos werden von der Shell selbst ausgeführt. Ein eingebautes Kommando als Element einer pipeline, wird in einer Sub-Shell ausgeführt, es sei denn, es handelt sich um das letzte Element der Pipeline.

Es folgt eine Übersicht von einigen wichtigen eingebauten Kommandos:

alias
alias *Name*
alias *Name Wortliste*
Die erste Form zeigt alle alias-Strings. Die zweite Form zeigt den **alias** für *Name*. Die letzte Form definiert die *Wortliste* als den **alias** für *Name*.

break
Stopt die Ausführung nach dem **end**-Kommando der nächst-umschließenden **foreach-** oder **while-**Schleife. Kommandos der aktuellen Zeile werden alle ausgeführt.

case *Label***:**
Ein *Label* in einer **switch** Anweisung.

cd
cd *Verzeichnisname*
chdir
chdir *Verzeichnisname*
Wechsele das aktuelle Shellverzeichnis nach *Verzeichnisname.* Wird kein Argument genannt, wird in das Heimverzeichnis gewechselt.

continue
Fortsetzen mit der Ausführung der nächst-umschließenden **foreach-** oder **while-**Schleife. Die Kommandos der aktuellen Zeile werden noch ausgeführt.

default:
Marke für den default-Fall einer **switch** Anweisung. Restliche Kommandos in der Zeile werden noch ausgeführt.

echo *Wortliste*
echo -n *Wortliste*
Wortliste wird auf Standardausgabe geschrieben, getrennt durch Leerzeichen, und abgeschlossen mit ein Zeilenwechsel, es sei die -n Option wurde angegeben.

else
end
endif
endsw
Siehe die **foreach, switch** und **while** Anweisungen.

eval *Argumente...*
Die Argumente werden als Shell-Eingabe übernommen, und die daraus resultierenden Kommandos ausgeführt.

exec *Kommando*
Kommando ersetzt die aktuelle Shell.

exit
exit (*Ausdruck*)
Die Shell terminiert mit dem Wert der *status* Variable (erste Form) oder mit dem Wert des *Ausdrucks*

foreach *Name* (*Wortliste*)
...
end
Die Variable *Name* wird suksessive ersetzt durch jedes Wort in *Wortliste*, dabei wird jeweils der Anweisungsblock bis zum zugehörigen **end** ausgeführt (**foreach** und **end** müssen einzeln, und in getrennten Zeilen stehen).

goto *Wort*
Wort wird als Zielzeile zum Forsetzen gesucht.

history
history *n*
history -r *n*
Ausgabe der (*n*) gespeicherten Kommandos.

if (*Ausdruck*) *Kommando*
Implementation der einfachen if-Verzweigung. *Kommando* darf nur ein einfaches Kommando sein (keine pipeline oder Kommando-Liste).

if (*Ausdruck1*) **then**
...
else if (*Ausdruck2*) **then**
...
else
...
endif
Implementation der geschachtelten if…then…else Verzweigung. Der Anweisungs*block* zwischen **then** und **else** bzw. **endif** wird ausgeführt.

onintr [-] [*Label* **]**
Zum Abfangen von Interrupts. Ohne Argumentangabe wird die Voreinstellung der Shell, was das Verhalten bei Unterbrechungen betrifft, wieder aktiviert. Wird - angegeben, werden alle Interrupts ignoriert. *Label* läßt die Shell eine **goto Label** Anweisung ausführen, wenn ein Interrupt erkannt wird.

popd [+n]
Das oberste Element des Verzeichnis-Stacks wird dem Stack entnommen, und wird zum aktuellen Verzeichnis. Das Argument entfernt das $\mathbf{n}^{te}$ Element aus dem Stack. Das oberste Element hat n=0.

pushd [*Name*] [**+n**]

Ohne Argumentenangabe werden die beiden oberen Stack-Elemente vertauscht. Ist *Name* angegeben, wird dieses zum aktuellen Verzeichnis, und das eben noch aktuelle Verzeichnis wird auf den Stack gelegt. Die numerische Angabe läßt das **n**te Element zuoberst auf den Stack wandern, und wechselt das aktuelle Verzeichnis nach diesen Wert.

set
set *Name*
set *Name* = *Wort*
set *Name* [*Index*] = *Wort*
set *Name* = (*Wortliste*)

Die erste Form zeigt den Wert aller Shell Variablen. Die zweite Form setzt *Name* auf Leerstring. Die dritte Form setzt *Name* auf das einfache *Wort.* Die vierte Form setzt das *Index*te Element von *Name* auf *Wort* ; dieses Element muß zu diesem Zeitpunkt bereits vorhanden sein. Die letzte Form schließlich, setzt *Name* auf die Liste der Worte in *Wortliste* .

setenv *Name Wert*

Setzt die Umgebungsvariable *Name* auf den Textstring *Wert.*

shift [*Variable*]

Die Elemente von *argv* werden um eins nach links geschoben, dabei wird das Element *argv*[1] verworfen. Ist *Variable* angegeben, wird dieselbe Aktion auf die angegebene *Variable* ausgeführt..

source *Name*

Die Shell liest Kommandos von *Name.* Schachtelung ist erlaubt; zu starke Verschachtelung führt jedoch u.U. zum Abbruch mangels verfügbaren Filedeskriptoren.

switch (*String*)
 case *str1*:
   ...
  breaksw
   ...
  default:
   ...
  breaksw
endsw

Jede **case** Marke (*str1*) wird mit dem *String* verglichen.

time [*Kommando*]
Ohne Argument wird die von der aktuellen Shell und ihren Unterprozessen verbrauchte Zeit ausgegeben. Mit Argument werden die Prozesszeiten des *Kommandos* ausgegeben.

unalias *Muster*
Alle alias-Besetzungen die dem *Muster* entsprechen, werden verworfen. Daher entfernt **unalias** * alle alias-Definitionen. Es erfolgt keine Fehlermeldung, wenn *Muster* nicht zutrifft.

unset *Muster*
Alle Variablen, deren Namen dem *Muster* entsprechen, werden verworfen. Daher entfernt **unset** * alle Variablen; diese Variante hat also sehr unangenehme Seiteneffekte. Es erfolgt keine Fehlermeldung, wenn *Muster* nicht zutrifft.

unsetenv *Muster*
Alle Variablen der Shell-Umgebung, deren Namen dem *Muster* entsprechen, werden verworfen. Siehe auch **setenv**.

while (*Ausdruck*)
...
end
Die Kommandos zwischen der **while** Zeile und der dazugehörigen **end** Zeile werden solange wiederholt, wie *Ausdruck* einen von Null verschiedenen Wert ergibt. **Break** und **continue**. können die Schleife vorzeitig beenden

9.3.2. Ausführung nicht eingebauter Kommandos

Soll ein nicht eingebautes Kommando ausgeführt werden, dann versucht die Shell dieses über den exec()-Systemaufruf zu aktivieren.

Geklammerte Kommandos werden in einer Subshell ausgeführt. Also wird

(cd; pwd)

das Heimverzeichnis ausgeben, das aktuelle Verzeichnis aber bleibt aktuell.

cd; pwd

tut dasselbe, wechselt jedoch in das Heimverzeichnis.

Ist die angegebene Datei (zur Erinnerung: Kommandoname = Dateiname unter Unix) eine ausführbare (execute-permission), aber keine binär-Datei, wird angenommen, daß es sich um eine Kommandoprozedur (script) handelt.

Dabei geben die beiden ersten Zeichen dieser Datei an, welcher Interpreter (in der Regel eine Shell) hierzu aktiviert werden soll:

"#!*Interpreter*".

Wird kein "#!*Interpreter*" angegeben, ist aber das erste Zeichen ein **#,** wird die C-Shell (**/bib/csh**) als zuständig aktiviert.

Wird "#!*Interpreter*" nicht angegeben, und ist das erste Zeichen kein #, wird die Bourne Shell (**/bin/sh**) aktiviert, um das Script zu interpretieren.

9.4. Variablen

Die csh kontrolliert einen Satz Variablen, deren Wert aus Null oder mehr Zeichenketten (Worte) besteht. Ein Variablenname besteht aus bis zu 20 Buchstaben oder Ziffern, das erste Zeichen ist ein Buchstabe. Einige dieser Variablen sind logische Variablen, d.h., die Shell interessiert sich nicht für den Wert per se, sondern nur dafür, ob ein Wert existiert oder nicht.

Der @ erlaubt numerische Kalkulation. Ein Leerstring wird als Null interpretiert.

Nachdem auf eine Eingabe die alias-Subtitution ausgeführt wurde, und bevor jedes Kommando ausgeführt wird, wird die Variable (an das $-Zeichen erkannt) durch ihren Wert ersetzt. Eine Ersetzung erfolgt nicht, wenn vor dem $ ein \ steht, außer zwischen ", wo immer substituiert wird. Variablen zwischen einfachen Hochkommas werden nie expandiert. Zeichenketten in einfachen Hochkommas ´ werden erst später interpretiert, sodaß in diesem Fall u.U. gar keine Variablen-Substitution erfolgt. Die nachstehenden Formen können benutzt werden, um Variablen in Shell-Eingaben zu kennzeichnen:

$*Variable_Namen*
${ *Variable_Namen***}**

Diese Sequenz wird in der Interpretation ersetzt durch die einzelnen Worte, getrennt durch Komma, des Variablenwertes der Variable *Variable_Namen.* Die geschweiften Klammern trennen die Werte der Variable von eventuell nachfolgenden Zeichen oder Worten.
Ist *Variable_Namen* keine csh-Variable, aber zum Environment gehörend, wird dieser Wert benutzt. Nicht csh-Variablen können nicht modifiziert werden.

$ *Variable_Namen*[*Auswahl*]
${ *Variable_Namen*[*Auswahl*]**}**

Erlaubt es, nur bestimmte Worte eines Variablenwertes anzusprechen. Die *Auswahl* besteht aus einer oder zwei (durch Bindestrich getrennte) Zahlen, die einen Wortindex-Bereich angeben. Das erste Wort eines Variablenwertes hat den index 1. Wird der erste Index eines Bereiches weggelassen, wird er als 1 angenommen. Bleibt das letzte Element eines Bereiches unbenannt, gilt als Index die Anzahl der Worte des Variablenwertes .

$# *Variable_Namen*
${# *Variable_Namen***}**

Liefert die Anzahl der Worte in der Variable *Variable_Namen.*

$ 0

Wird ersetzt durch den Dateinamen der Datei, von der die Kommandos gelesen werden.

$*Index*
${*Index***}**

Ist eine indizierte Auswahl der Variable *argv* (**$***argv[Index]*).

$ *

Selektiert alle *argv* (**$***argv[*]*).

$ <

Liest eine Zeile von der Standardeingabe. Kann benutzt werden, um ein Shellscript über Tastatureingaben zu steuern.

Vordefinierte- und Umgebungs-Variablen

argv

Diese Variable nimmt die Argumente eines csh-Kommandos auf.

cwd

Diese Variable enthält den vollständigen Pfadnamen des aktuellen Verzeichnises.

echo

Wenn gesetzt, werden Kommandos und Argumente auf die Standardausgabe mitgeschrieben .

history

Setzt die Größe des Puffers zum Speichern von vorausgegangenen Kommandos.

home
Diese Variable enthält den vollständigen Pfadnamen des Heimverzeichnises (die HOME environment-Variable).

noclobber
Diese Variable unterstützt die Ein/Ausgabe-Umleitung in sofern, daß sie verhindert, daß Dateien unbeabsichtigt gelöscht werden (Ausgabe), oder Eingaben in bereits vorhandene Dateien erfolgen.

path
Übernimmt die Einstellung der PATH environment-Variable.

prompt
Mit dieser Variable können Sie ihren eigenen Bereitschaftsstring definieren. Das Zeichen ! wird ersetzt durch die aktuelle history-Puffer-Nummer.

shell
Enthält den Dateinamen des Shell-Programms.

status
Enthält den Status-Rückgabewert des zuletzt ausgeführten Kommandos. Der Hex-Wert 0200 wird im Fehlerfall auf den Wert der Statusvariable addiert. Eingebaute Kommandos liefern die 1 als Fehleranzeige, und 0 im Normalfall.

time
Diese Variable enthält einen numerischen Wert (maximale Anzahl Sekunden), der das automatische "timing" von Kommados steuert.

verbose
Diese Variable wird durch die **-v** Kommando-Option gesetzt.

9.5. Ausdrücke

Einige der eingebauten Kommandos erwarten als Argumente Ausdrücke. Diese Ausdrücke enthalten die von der C-Sprache bekannten Operatoren (unten Gruppiert und sortiert nach fallender Gewichtigkeit):

```
*    /    %
+    -
<<   >>
<=   >=   <>
==   !=   =~   !~
```

Die Operatoren der letzten Zeile vergleichen ihre Argumente als Zeichenketten; alle andere Operatoren wirken auf Zahlen. Die Operartoren **=~** und **!~** wirken wie die

== und **!=** Operatoren, aber die rechte Seite enthält ein Muster (mit *, ? und [...] Elemente), mit dem die linke Seite verglichen wird.

Zeichenketten die mit **o** beginnen, werden als Oktalzahl interpretiert. Leere oder fehlende Argumente werden als **o** interpretiert. Das Ergebnis eines jeden Ausdrucks ist die Zeichenkettendarstellung einer Dezimalzahl.

9.6. Ein/Ausgabe

Die Standardeingabe und -ausgabe eines Kommandos können mit folgender Syntax umgeleitet werden:

< *Dateiname*
: Öffne die Datei *Dateiname* als Standardeingabe.

<< *Wort*
: Lese die Shell-Eingabe bis zu der Zeile, die *Wort* entspricht. *Wort* wird buchstäblich genommen (keine Variable-, Dateinamen-, oder Kommando-Substitutionen).

> *Dateiname*
>**!** *Dateiname*
>**&** *Dateiname*
>**&!** *Dateiname*
: Die Datei *Dateiname* wird als Standardausgabe benutzt. Die Datei wird erzeugt, wenn sie noch nicht existiert; existiert sie aber, wird sie geleert. Ist die Variable noclobber gesetzt, dann darf die Datei nicht existieren, sonst wird eine Fehlermeldung erzeugt. So kann das unbeabsichtigte Löschen von Dateien verhindert werden. Mit den !-Varianten kann dieser Test unterbunden werden
Die &-Varianten lenken zusätzlich die Standardfehlermeldungen in diese Datei.

>> *Dateiname*
>>**!** *Dateiname*
>>**&** *Dateiname*
>>**&!** *Dateiname*
: Wie >, die neue Ausgabe wird an die bestehende Datei angehängt. In diesem Fall muß, bei gesetzter noclobber-Variable, die Datei existieren, es sei denn, **!** ist angegeben. Sonst wie > .

9.7. Erzeugen und Ausführen eines Shell-Scripts

Zum Erstellen eines Shellscripts wird ein normaler Texteditor benutzt. Für kurze Prozeduren kann auch das **cat**-Kommando benutzt werden:

```
$ cat   >meinScript
echo Halihalo
^D
$
```

Damit wir die Datei als Kommando ausfrufen können, muß sie die Kennung "ausführbar" (execute-permission) zugewiesen bekommen. Dies kann mit dem **chmod** Kommando erfolgen:

```
$ chmod   +x   meinScript
$
```

9.8. Beispiele

Das erste Beispiel zeigt wie längere Texte (z.B. Erläuterungen) in eine Kommandoprozedur eingebaut werden können, ohne **echo** zu verwenden.

```
# Beispiel 1
cat << !
Dieses Beispiel zeigt die Anwendung von cat,
um längere Texte in eine Kommandoprozedur
einzubauen.
!
echo "Fertig"
```

Shellprozeduren können auch interaktiv gestalltet werden:

```
#  Beispiel 2
cat << !
Diese Beispiel zeigt die Anwendung von
Tastatureingaben in einer Kommandoprozedur.
!
echo -n "Eingabe:"
set line = $<
echo "Folgende Zeile wurde gelesen: < $line >"
echo "Fertig mit Anzeigen = $status"
```

Es folgt ein Beispiel für die **foreach** Anweisung:

```
# Beispiel 3
echo -n "mehrere Dateinamen eingeben: "
set line = $<             # lese  von der Tastatur
foreach file ($line)
  ls -l $file
end
```

Und hier ein Beispiel der **if** -Verzweigung

```
# Beispiel 4
set d = `date`
if ($d[1] == Fri) then
  echo Frohes Wochende
endif
```

Ein Beispiel wie Argumente genutzt werden:

```
# Beispiel 5
if  ($#argv != 1) then
  echo "Bedienfehler: Argument erwartet"
  exit 1
endif
...                        # normale Bearbeitung
```

Teil 2: Einführung in die Programmiersprache C

1. Entwicklungsgeschichte

Die Programmiersprache C ist eine verbesserte Nachfolgerin der Sprache B, die speziell zur Entwicklung der UNIX-Betriebssysteme enwickelt wurde. C wurde 1973 von Kernighan und Ritchie implementiert und ist als höhere Programmiersprache auch mit Sprachelementen ausgestattet, die es erlauben, maschinennah zu programmieren. Andererseits ist C auch eine allgemein einsetzbare Sprache, mit der Übersetzerprogramme, mathematische Algorithmen, Textverarbeitungssysteme und andere Dienstprogramme realisiert wurden.

2. Preprozessor

Zum C-Compiler gehört ein integrierter Preprozessor, der auf Zeilen reagiert, die mit # beginnen. Die wichtigsten Preprozessorbefehle sind `#define` und `#include`. Durch `#define` wird der dort angegebene Name im nachfolgenden Programmtext jeweils durch einen vereinbarten Text ersetzt. Auf diese Weise können Konstanten definiert werden.

Mit `#include` werden Dateien eingebunden. `#include` hat die gleiche Wirkung, als ob das eingebundene File direkt an der Aufrufstelle im Programme stünde.

Beispiele:

```
#define ANZAHL  1000   /* ANZAHL wird nun stets durch den */
                       /* Wert 1000 ersetzt               */
#include <test.c>      /* Einbinden des Files test.c      */
```

Die Klammerung des Dateinamens im `include` Befehl kann u.A. sein:

`<Datei>` gibt an dass die Datei unter dem Pfad der Standartbibliothek (anlagenabhängig, z.B. /usr/lib/include) zu suchen ist.

`"Datei"` gibt an dass die Datei aus dem aktuellen Verzeichnis geladen werden soll.

3 Notationsregeln

Diese Regeln beschreiben die grundlegende Syntax der Sprache: wie trennt oder klammert man, wie kann man Variablen nennen.

3.1 Trenner

In den meisten Fällen gelten Leerzeichen, Tabulatoren und Zeilenenden (alles sogenannte weiße Leerzeichen -whitespace-) **nicht** als Trenner: sie werden beim Kompilieren ignoriert.

Anweisungen werden normalerweise durch einen Kommapunkt voneinander getrennt. Das Komma kann als Trenner von Argumente, aber auch als Trenner von Anweisungen vorkommen (siehe weiter unten).

Beispiel eines einfachen C-Programms:

```
main (int argc)       /* mein erstes C-Programm */
{
printf ("%d Argumente wurden eingegeben!", argc-1);
}
```

Das Programm errechnet über den Ausdruck argc-1 die Anzahl der eingegebenen Argumente und gibt diese Zahl zusammen mit dem in Hochkommas gestellten Text aus.

3.2 Schlüsselworte

Tabelle 3.1 listet die 32 Schlüsselworte, die zusammen mit der formalen Syntax, die Sprache C ausmachen.

auto	const	double	float
break	continue	else	for
case	default	enum	goto
char	do	extern	if

int	short	struct	unsigned
long	signed	switch	void
register	sizeof	typedef	volatile
return	static	union	while

Tabelle 3.1

Alle Schlüsselworte sind klein geschrieben. Gross- und Kleinschreibung sind von Bedeutung in C: `continue` ist ein Schlüsselwort, CONTINUE dagegen nicht. Ein Schlüsselwort ist reserviert, d.h. es kann nicht auch als Variablen- oder Funktionsnamen benutzt werden.

3.3 Bezeichner

Alle Wörter, die nicht reserviert sind, sind Bezeichner. Für Bezeichner gelten folgende Regeln:

- alle Ziffern sowie große und kleine Buchstaben sind erlaubt, wie das Underline Zeichen "_", dass als Buchstabe gilt
- an erster Stelle muß ein Buchstabe stehen
- der Compiler erkennt die ersten N Stellen in einem Bezeichner – alle übrigen Ziffern und Buchstaben sind zwar erlaubt, werden aber ignoriert (N ist Implemetierungs-abhängig).

Beispiele für gültige Bezeichner:

```
Student   Switch   Main   hallo   g10   _myName
```

3.4 Klammerung

Im einfachen Programmbeispiel weiter oben (3.1) gibt es folgende Arten der Klammerung:

{ } Klammern eine Anzahl von Anweisungen (ein Block)

( ) enthält Parameterlisten für Funktionen. Später werden sie auch als Klammerung für Ausdrücke und Kontrollstruktur-Bedingungen vorgestellt.

/* */ "klammern" Kommentare. Das Ganze gilt für den Compiler wie ein Leerzeichen, auch wenn es sich über mehrere Zeilen ausbreitet. Kommentare können nicht geschachtelt werden, d.h. z.B.

```
/* Ungültige /* Kommentarzeile */
```

ist nicht erlaubt.

3.5 Allgemeine Struktur

Ein C-Programm besteht aus mindestens eine Funktion: der **main()** Funktion. Diese Funktion wird als erste aufgerufen (= gestartet) wenn ein C-Programm abläuft. Obwohl technisch gesehen **main** nicht zur Syntax gehört, ist es eine gute Idee, zu tun als ob. Den Namen **main** als eine Variable zu benutzten, führt mit ziemlicher Sicherheit zu Problemen bei der Kompilierung.

Im Allgemeinen hat ein C-Programm die unten aufgeführte Struktur.

```
Globale Deklarationen
main()
{
Lokale Variablen
Anweisungsfolge
}
f1()
{
Lokale Variablen
Anweisungsfolge
}
f2()
{
Lokale Variablen
Anweisungsfolge
}
.
.
```

```
fN()
{
Lokale Variablen
Anweisungsfolge
}
```

Bild 3.2 Struktur eines C-Programms

4. Datentypen

Im folgenden sind die wichtigsten Datentypen (anlagenabhängig) aufgelistet:

- int, long int, unsigned int (4 Bytes)
- short int (2 Bytes)

Variablen von diesen Typen sind ganze Zahlen im Bereich von -32.768 bis +32.767 bei 2 Byte Größe und von -2.147.483.648 bis +2.147.483.647 bei 4 Byte.

- char, unsigned char (1 Byte)

Variablen von diesem Typ speichern normalerweise den ASCII-Wert (0-127) von Zeichen ab. In C besteht jedoch die Möglichkeit, char und int Werte gegeneinander auszutauschen. Dabei wird die interne ASCII-Codierung eines Zeichens einmal als Character und das andere mal als Integer-Wert interpretiert. Char hat folgende eigene Konstanten:

´z´ ASCII-Wert des sichtbaren Zeichen z

´\c´ ASCII-Wert besonderer Zeichen; c kann u.A. folgende Werte annehmen:

n Zeilenende (newline)

t Tabulator

b Rückschritt (backspace)

´\o´ oktaler ASCII-Wert o (bis zu 3 Ziffern sind erlaubt)

Beispiele für char Konstanten: ´X´ ´+´ ´\n´ ´\7´

- float (4 Bytes)
- double float (8 Bytes)

Variablen von diesen Typen nehmen rationelle Werte von einfacher (bis auf 7 Stellen genau) bzw. doppelter (15 Stellen) Genauigkeit auf.

5. Deklarationen

Die meisten Programme müssen Daten abspeichern, um sie im späteren Ablauf verarbeiten zu können. In strukturierten Programmiersprachen werden solche Speicherplätze dem Datentyp entsprechend vor ihrer späteren Verwendung "vereinbart" bzw. deklariert.

Eine Deklaration besteht aus dem Typ gefolgt von der bzw. den zugehörigen Variablen.

Beispiele:

```
char zeichen;

long int x,y;
```

6. Operationen

Folgende Operationen sind möglich :

- Zuweisung: =
- Arithmetik: + , - , * , /
- Inkrement: ++ (i++ entspricht i = i+1;)
- Dekrement: -- (i-- entspricht i = i-1;)
- Modulo: %
- Vergleichsoperationen:

  < (ist kleiner),

  <= (ist kleiner oder gleich),

  > (ist größer),

>= (ist größer oder gleich),

== (ist gleich),

!= (ist ungleich)

- Logikoperationen:

  ! (Negation),

  && (Und-Verknüpfung),

  || (Oder-Verknüpfung)

Die Logikoperationen entsprechen dem NOT, AND und OR in Pascal.

- Bitweise Operationen (bitweise Verknüpfung von Variablen):

  & (Und-Verknüpfung),

  | (Oder-Verknüpfung),

  ! (Negation)

- der **sizeof** Operator

  Der Operator sizeof ist ein unärer Operator der die Anzahl Byte einer Variablen oder Typkennung zur Compile-Zeit liefert.

Beispiele:

```
zeichen = ´a´;

i = 1;

if (i == 0) printf("i ist null");

bitgesetzt = anz && MASKE;

printf("Short belegt %d Bytes!\n", sizeof(short));
```

7. Kontrollstrukturen

7.1. Anweisungen und Blöcke

Anweisungen werden mit einem Semekolon abgeschlossen.

Beispiel:

```
x = 0 ;

i++;
```

Geschweifte Klammern {} bilden einen Block. In jedem Block können zu Beginn neue, nur in diesem Block gültige, Variablen vereinbart werden. Außerhalb von Blöcken definierte Variablen sind global und in allen Blöcken gültig.

Beispiel: siehe unten in der else-Verzweigung.

If-else Anweisung

Einfache Verzweigung: wenn die Bedingung erfüllt ist, wird der Anweisungsteil nach der Bedingung stehend ausgeführt. Der else-Teil ist optional. Entscheidungen hängen davon ab, ob der arithmetische Ausdruck (die Bedingung) von Null verschieden ist. Der Ausdruck muß einen Integer-Wert liefern, ist er ungleich Null, so ist die Bedingung erfüllt. Es gibt **keine eigenen booleschen Datentypen**, diese werden durch die Integer-Werte 0 (unwahr) und 1 (wahr) realisiert.

Syntax:

```
if (<Ausdruck>)
<Anweisungsfolge1>
else
<Anweisungsfolge2>
```

Beispiel:

```
if (a > b)
z = a;            /* wird ausgeführt wenn a > b ist */
else {
     z = b;       /* wird ausgeführt wenn a <= b ist */
     y = a - b;   /* wird ausgeführt wenn a <= b ist */
```

```
        }
```

Vorsicht: Zuweisungen ergeben üblicherweise den Wert "wahr" (Verwechslung von = und ==).

Switch-Anweisung

Die switch-Anweisung benutzt mann bei mehrfach Verzweigungen, wenn das Ergebniss der Bedingung einen ganzzahligen Wert liefert. Der Default-Teil wird ausgeführt wenn keine der explizit genannten Ergebnisse zutreffen, und ist optional. Die einzelnen Fallunterscheidungen werden durch break abgeschlossen.

Beispiel:

```
switch  (c)
 {
  case 1   :   z = a;
           break;
  case 2   :  z = b;
           break;
  default  :   z = 0;
           break;
 }
```

Falls die Variable c den Wert 1 hat, wird z=a ausgeführt. Falls sie den Wert 2 hat, wird z=b ausgeführt, ansonsten z=0.

While-Schleife

Bei der while-Schleife wird die Bedingung <Ausdruck> vor jeder Ausführung des Schleifenrumpfs geprüft. Der Schleifenrumpf wird daher nicht unbedingt durchlaufen.

Syntax:

```
while (<Ausdruck>)
<Anweisungsfolge>
```

Beispiel:

```
while (i < n)
 {
```

```
  z = z + (z * i);
  i++;
 }
```

Do-while Schleife

Bei der do-while Schleife wird die Bedingung (<Ausdruck>) erst nach der Ausführung des Schleifenrumpfes geprüft. Der Schleifenrumpf wird also mindestens einmal durchlaufen.

Syntax:

```
do
<Anweisungsfolge>
while (<Ausdruck>);
```

Beispiel:

```
do {
    z = z + (z * i);
    i++;
   }
while (i <= n);
```

For-Schleife

Syntax:

```
for (<Initialisierung>;<Schleifentest>;<Weiterschalten>)
<Anweisungsfolge>;
```

Der Initialisierungsausdruck weist im einfachsten Fall einer Laufvariablen einen Wert zu, der Schleifentest prüft bei jedem Schleifendurchlauf die Schleifenbedingung und der Ausdruck zum Weiterschalten enthält i.A. eine Zuweisung bzw. Inkrementierung zur Fortschaltung der Laufvariablen.

Beispiel: i wird auf 0 initialisiert, pro Durchlauf um 1 erhöht und auf <n getestet:

```
for (i = 0; i < n; i++)
 z = z + (z * i);
```

Sprünge

C kennt auch Sprungbefehle, die jedoch nur äußerst restriktiv verwendet werden sollten! Eine Marke oder Sprungziel ist ein gültiger C-Bezeichner, gefolgt von einem Doppelpunkt.

Syntax:

```
goto <Marke>;
.
.
<Marke>:
```

Arrays

Mit den einfachen Datentypen von Abschnitt 2 könen auch Arrays und Pointer definiert werden.

Deklaration:

```
<Typname> <Variablenname>[<Feldgröße>]
```

Beispiel:

```
char name1[10]
```

Arrays beginnen in C prinzipiell mit dem Index 0 und laufen bis zum Index Feldgröße-1. Jetzt sind also Felder von name[0] bis name[9] vom Typ char definiert. Durch den Ausdruck name[integer] wird auf das Arrayfeld mit dem entsprechenden Index zugegriffen.

7.2 Zeiger

7.2.1 Direkte Adressierung des Datenspeichers

In den bisherigen Beispielen wurde auf den Datenspeicher nur über Variablen zugegriffen. Den mit der folgenden Vereinbahrung

```
char z = ´A´ ;
```

angelegten Speicher kann man sich wie folgt vorstellen:

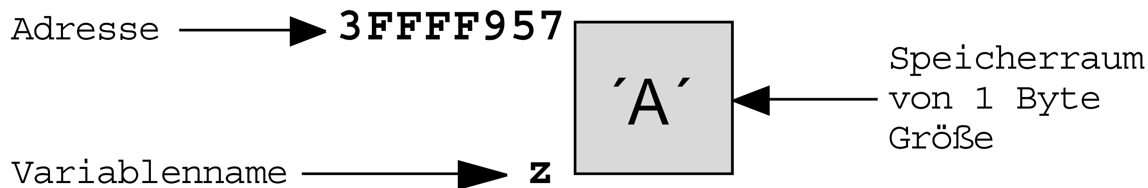


Bild 6.1 Der Zugriff auf den Speicher kann sowohl über die Adresse als auch über dem Namen erfolgen.

Diesen Speicherraum kann man in C auch direkt über seine Adresse ansprechen.

Dazu gibt es in C den Präfixoperator **&**. Vor einer normalen Variablen stehend, liefert er die Byte-Adresse, an der der entsprechende Speicherraum anfängt:

```
printf("Adresse von z ist %u\n", &z);
```

7.2.2 Zeigervariable (Pointer)

Der nächste Schritt besteht jetzt darin, sich solche Adressen verschiedener Speicherräume im Program zu merken, zur späteren Verwendung. Eine neue Variablenart, zur Abspeicherung von Adressen, wird also benötigt.

Dazu gibt es die Vereinbahrungsform:

```
char  *pz
```

Eine solche *Zeigervariable* kann also im Laufe des Programms mit verschiedenen Adressen von `char`-Speicherräumen besetzt werden, wie etwa:

```
pz = &z;
```

Zeigervariablen enthalten also Speicheradressen und sind schon deshalb keine normale Variablen. Sie haben folgende Eigenschaften:

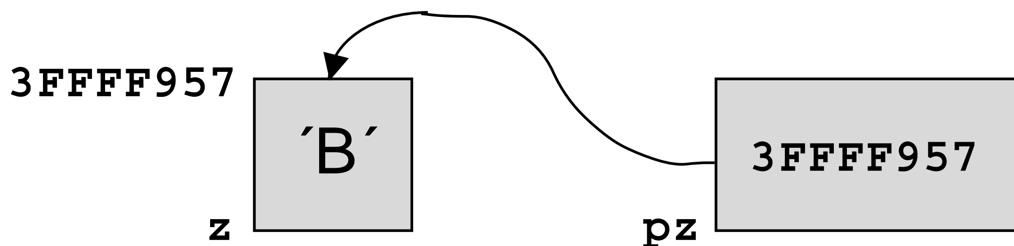
- Ihre Vereinbahrungen beziehen sich auf bekannte Datentypen, in dem sie Speicherräume von dieser Größe adressieren können.

- Von den bisherigen Operationen können nur Zuweisung, Addierung, Subtrahierung und natürlich auch Adressabfrage über & bei ihnen sinnvol benutzt werden.
- Um den adressierten *Speicherraum* anzusprechen, gibt es einen neuen Operator: *Zeigervariable (oder *Pointer).

Der neue Operator * ist besonders wichtig: er erlaubt nämlich die Manipulation von Speicherräumen über ihre Adressen:

```
*pz = ´B´;
```

Da pz die Adresse von dem `z`-Speicherraum enthält, also darauf "zeigt", hat die Zuweisung diesen Speicherraum verändert.



Der Ausdruck `*pz` *dereferenziert* den Zeiger und gilt somit, als ob man `z` benutzt hätte.

Weiteres Beispiel: Seien `symbol` und `psymbol` folgendermaßen deklariert:

```
char symbol;   /* Variable symbol vom Typ char   */
char *psymbol; /* Pointer auf ein Feld vom Typ char */
```

und es finden folgende Zuweisungen statt:

```
symbol  = 'a';          psymbol = &symbol;
```

Zuerst wird der Variablen symbol as Zeichen a, dann dem Pointer psymbol die Adresse von symbol zugewiesen. Diese sei im Beispiel 00FF hexadezimal. Das bedeutet, daß der Pointer `psymbol` nun die Adresse von symbol enthält, und damit auf `symbol` zeigt (s. Bild 14.1 unten).

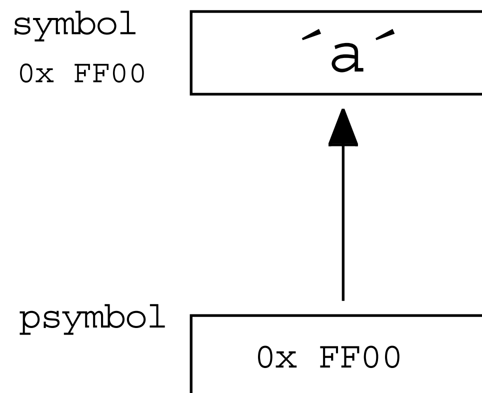


Bild 6.1 Variablenzugriff über Pointer

Um nun mit Hilfe der Variablen `psymbol` auf die Variable `symbol` (den Wert oder Inhalt = Zeichen a) zuzugreifen, wird der Stern-Operator (*) verwendet. `*psymbol` ist der Inhalt der Variablen `symbol` mit der Adresse `00FF`. In diesem Beispiel bezeichnen `symbol` und `*psymbol` beide dasselbe, nämlich den Wert `a`.

Arrays und Pointer

Arrays werden intern als Zeiger auf Datenbereiche realisiert. Der Zugriff auf ein Arrayfeld entspricht dabei dem Zugriff auf den Inhalt eines Pointers. Somit kann auf das gleiche Feld wahlweise über einen Arrayindex oder über einen Pointer zugegriffen werden. Folgende Beispiele sollen die Zusammenhänge verdeutlichen:

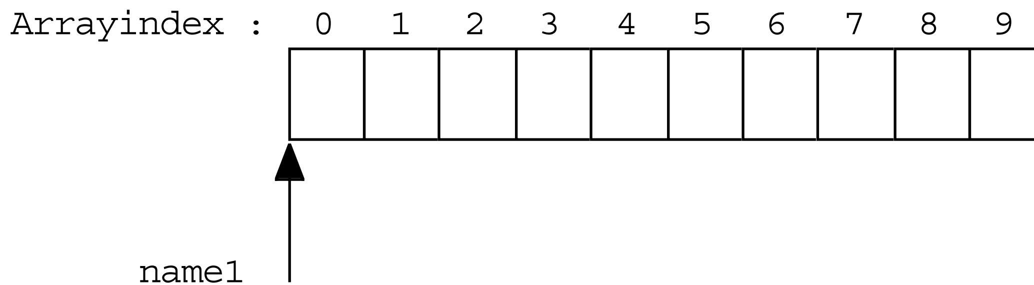
Seien name1, name2 wie folgt vereinbart:

```
char  name1[10];
char *name2;
```

Der Name eines Arrays ist ein Zeigerwert (d.h. eine Adresse), der auf das erste Element eines Arrays zeigt, es gilt also:

```
name1 == &name1[0] /* Zeiger auf das erste Element */
```

Die interne Speicherung des Arrays name1 hat also folgende Struktur:



Man erreicht ein bestimmtes Element im Array, indem man von der Adresse des ersten Elements ausgeht und entsprechend oft die Größe - gemessen in Bytes - eines Arrayelements dazu addiert. Damit erh"alt man einen Pointer auf das gewünschte Element (hier z.B. das vierte Element mit Index 3):

```
&name1[3] == name1 + 3
```

Folgende Zuweisungen sind identisch und zulässig:

```
name2 = &name1[0]   und   name2 = name1,
```

der Zeigerwert (also die Adresse) von `name2` ist jetzt gleich der Adresse von `name1`.

Structures

Einfache Datentypen können zu Strukturen zusammengefaßt werden. Zunächst muß die Structure mit ihrem Namen und ihren Komponenten deklariert werden. Dann kann ihr Name als eigener Typ zur Definition von Variablen verwendet werden. Dabei können Variablen natürlich auch als Zeiger auf eine Structure definiert sein. In diesem Fall werden Structureelemente mit dem Pfeil -> ausgewählt. Bei einer Variblen, die direkt vom Typ der Structure ist (kein Zeiger darauf), werden Komponenten mit dem Punkt . ausgewählt.

Deklaration:

```
struct <Structname>
     {
     <Typ> <Komponentenname 1>;
     .
     .
     .
```

```
        <Typ> <Komponentenname n>;
        }
```

Beispiel:

```
Typdeklaration:   struct listelement
                /* Element einer verzeigerten Liste */
                {
                long int   inhalt;
                char  code;
                struct listelement *nextelem;
                /* Zeiger auf ächstes El. */
Variablen:          struct listelement *anker;
                /* Zeiger auf Element */
                    struct listelement element;
                /* Listenelement */

Zugriff auf Komponenten:
anker->inhalt  = ... /* Zugriff über Zeiger */
element.inhalt = ... /* direkter Zugriff    */
```

8. Ein-und Ausgabe

In C gibt es eine Reihe von Libraryfunktionen, die bereits vordefiniert sind und automatisch zu Programmen gebunden werden können (`#include <stdio.h>`). Dazu gehören auch die Ausgabefunktion *printf* und die Eingabefunktion *scanf*.

8.1. Ausgabe (printf)

Die Funktion `printf` dient zur Ausgabe von Text, der direkt angegeben werden kann, und zur Ausgabe von Variablenwerten. Dazu werden sogenannte Formatelemente verwendet. Sie bestimmen den Typ. Hier einige der wichtigsten:

%d ==> Ein Integerwert wird dezimal ausgegeben.

%x ==> Ein Integerwert wird hexadezimal ausgegeben.

%c ==> Ein einzelnes Zeichen wird als Character ausgegeben.

%s ==> Eine Zeichenkette (String) wird ausgegeben.

printf wird folgendermaßen aufgerufen:

```
printf(<format>,<wert1>,<wert2>,...);
```

<format> ist eine Zeichenkette, die aus Text und/oder Formatelementen und/oder Steuerzeichen besteht, eingeschlossen in doppelte Anführungszeichen. Das wichtigste Steuerzeichen ist \n , das als Zeilentrenner wirkt (entspricht WRITELN in Pascal). <wert1> etc. sind beliebige Ausdrücke. Die Werte werden der Reihe nach unter Kontrolle der Formatelemente ausgegeben.

Beispiel für eine Ausgabe:

```
printf("Das Maximum von %d und %d ist %d\n",x,y,max);
```

Bei der Ausgabe werden die drei Formatelemente der Reihe nach durch die Werte von x, y und max ersetzt. Bei der Ausgabe von Strings mit %s muß beachtet werden, daß Strings als Pointer auf char realisiert werden. Das bedeutet, daß die Stringvariable auf das Feld zeigt, wo die einzelnen Zeichen des Strings abgespeichert sind. An die printf-Funktion muß in diesem Fall ein solcher Pointer übergeben werden.

8.2. Eingabe (scanf)

Zur Eingabe von Zeichen dient die Libraryfunktion `scanf`. Sie wird wie folgt aufgerufen:

```
scanf(<format>,<varpointer1>,<varpointer2>,...);
```

Ein `scanf`-Format ist analog aufgebaut wie ein `printf` Format. Wichtig: Die Parameter der `scanf`-Funktion (<varpointer1> etc.) müssen immer Pointer auf die einzulesenden Variablen sein, denn die Routine `scanf` schreibt die eingelesenen Werte an die Speicherstellen, die durch die Zieladressen <varpointer> gegeben sind. Dieser übergabemechanismus entspricht der Wertübergabe "by reference" in Pascal, die in C nur über Zeiger realisiert werden kann.

Beispiel:

```
scanf("%d %x",&a,&b);
```

Die Variable a wird dezimal, die Variable b hexadezimal eingelesen. Als Parameter werden Zeiger auf die Variablen übergeben. Einzelne Zeichen können auch mit der Standardeingabe getchar eingelesen werden.

9. Programmstruktur und Funktionen

Funktionen können nur global definiert werden. In einem C Programm muß es eine Funktion main geben, die als erste bei Ausführung des Programms automatisch aufgerufen wird und das Hauptprogramm darstellt.

Eine Funktion ist folgendermaßen strukturiert:

```
<Typname> <Name> (<Parameternamen>)
              <Parameterdeklarationen>
{
<Vereinbarungen>
<Anweisungen>
}
```

[<Typname>] ist der Typ des zurückzugebenden Funktionswertes.

[<Name>] ist der Name der Funktion.

[<Parameternamen>] sind eine optionale Liste von Parameternamen, die durch Komma getrennt sind. Bei Funktionen ohne Parameter müssen die Klammern jedoch auch angegeben werden. Wichtig: Nach) darf bei der Funktionsdeklaration kein ; stehen!

[<Parameterdeklarationen>] sind eine Folge von Typdeklarationen für die Parameternamen; sie müssen mit der Liste der Parameternamen übereinstimmen.

[<Vereinbarungen>] sind Definitionen von lokalen Variablen und Deklarationen von Objekten, deren Geltungsbereich auf die Funktion beschränkt sein soll.

[<Anweisungen>] stellt den Funktionsrumpf dar.

In C gibt es keinen Unterschied zwischen Funktionen und Prozeduren. Die aus Pascal bekannten Prozeduren werden als Funktionen ohne Wertrückgabe realisiert. Soll der Wert eines übergebenen Parameters durch die Funktion verändert werden (vgl. VAR - Parameter in Pascal), so muß dieser als Zeigerwert, also als Adresse übergeben werden.

Im folgenden wird ein Beispiel für eine Parameterübergabe ``by value" und für eine Übergabe ``by reference" aufgeführt:

Beispiel 1.

```
main() /* Hauptprogramm */
{
 long int x,y,max; /* Variablendeklarationen */
 scanf("%d %d",&x,&y);
 max = maximum(x,y); /* Aufruf der Funktion "maximum" */
 printf("Maximum ist %d",max);
}

/* Fkt. "maximum", Rückgabetyp long int */

long int maximum(a,b)
  long int a,b;     /* Parameter mit "Call by value"   */
{
  long int max;
  if (a > b)  max = a;
   else  max = b;
  return (max); /* Wertrückgabe */
}
```

In diesem Programm werden zwei Integers eingelesen und der Funktion maximum Übergeben, die den Wert max zurückgibt.

Beispiel 2.

```
main()                    /* Hauptprogramm */

{
 long int x,y;            /* Variablendeklarationen */

scanf("%d %d",&x,&y);

vertausche(&x,&y);      /* Aufruf der Funktion "vertausche" */
printf("%d\n %d\n",x,y);
}
```

```
  /* Funktion "vertausche" ohne Wertrückgabe   */
vertausche(a,b)
  long int *a,*b;

  /* Zeiger als Parameter (call by reference)  */

{

 long int h;
h = *a;
*a = *b;
*b = h;
}
```

Hier werden die Variablen x und y durch die Funktion vertausche tatsächlich verändert (entspr. Call by Reference). Dazu werden ihre Adressen (erzeugt durch &-Operator) an die Funktion übergeben, die dann die Inhalte der mit den Adressen referenzierten Felder verändert.

10. Fileoperationen

C stellt Standardfunktionen zum Zugriff auf Files (Dateien) zur Verfügung. Um von einem File zu lesen oder darauf zu schreiben, muß es zunächst geöffnet werden. U.A. kommt dafür der fopen-Bibliotheksaufruf in Frage (Siehe auch: Unix Systemaufrufe, 2. Versuch).

```
fp = fopen(<name>,<mode>);
```

Die Funktion `fopen` initialisiert `fp`, den Filepointer, als Zeiger auf das geöffnete File. Für alle weiteren Fileoperationen wird nur noch `fp` zur Identifikation des Files verwendet. `fp` ist als `FILE *fp` deklariert.

[name] gibt den Namen des Files an,

[mode] den Lese- oder Schreibmodus (r+ oder w+).

Zeichenweise von einem File gelesen wird mit der Funktion

`fgetc(fp)` (return eines chars).

Zeichenweise auf ein File geschrieben wird mit

`fputc(ch,fp)` (char ch).

Ein File kann mit

`fclose(fp)` geschlossen werden.

Außerdem gibt es zur File-E/A noch die Funktionen

`fscanf(fp,...)` (Lesen) und

`fprintf(fp,...)` (Schreiben).

Diese verwenden die gleichen Formatsteuerzeichen wie die Funktionen scanf und printf.

11. Speicherverwaltung

Um einen Speicherbereich zu reservieren, werden die Funktionen malloc oder calloc verwendet:

```
char *storage;

storage = malloc(<Elementanzahl in Byte>);

storage = calloc(<Elementanzahl>,<Elementgröße in Bytes>);
```

Es wird ein Zeiger erzeugt, der auf den Beginn des Speicherbereichs storage zeigt. storage ist von der Größe <Elementanzahl> Byte bzw. bei calloc multipliziert mit der <Elementgröße in Bytes>. Alle in C-Programmen definierten Zeiger müssen vor dem Zugriff auf die referenzierten Felder initialisiert werden! Dazu kann z.B. die Funktion calloc dienen.

Zur Freigabe eines reservierten Speicherbereichs gibt es die Funktion

```
cfree(<Pointer auf freizugebenden Bereich>).
```

12. Übersetzen,Binden und Ausführen

Das Kommando

```
gcc <file>    (für den Gnu-C Compiler) bzw.

cc <file>     (für den zum System gehörenden C-Compiler)
```

dient zum übersetzen von C-Programmen. Die Datei <file> muß vom Typ .c sein.

Es wird nun vom Compiler eine Datei <file>.o erzeugt, die zusammen mit anderen Dateien vom Typ .o mit dem link-Kommando zu ausführbarem Code gebunden werden kann, z.B.

```
ld <file> <file1> <file2> ...
```

Die ausführbare Datei heißt dann a.out. Zum Starten dieses Programms dient dann das Kommando:

```
a.out.
```

Teil 3: Aufgaben

1 Aufgaben zum Thema Unix

Wichtig: in den Skripten müssen Befehle und Parameter in nachvollziehbarer Weise kommentiert werden!

1.1. Erstellen Sie ein Shell-Skript mit folgender Ausgabe:

- Datum und Uhrzeit
- den Pfad des aktuellen Katalogs und dessen Inhalt
- die Liste aller angemeldeter Benutzer
- die Liste aller aktiven Prozesse
- die Meldung "Ende Shell-Skript!"

1.2. Erstellen Sie ein Shell-Skript (`cgl` -> compile, link, go) das eine C-Quelldatei übersetzt, bindet und startet. Der Dateiname ist als Parameter zu übergeben (`clg filename.c`). Aus den Quellen `filename.c` soll eine ausführbare Datei `filename` erzeugt werden. Treten Fehler beim Übersetzen auf, sollen diese in einer Fehlerdatei `filename.c.err` protokolliert werden, der Benutzer soll darüber informiert werden und es sollen keine weitere Aktionen (Binden, Starten) stattfinden.

Hinweis: Siehe dazu die Optionen des Compilers und des Binders unter man gcc und man ld.

1.3. Die nachstehende Kommandoprozedur liest eine Eingabe von stdin, und hängt die eingelesenen Zeilen vor eine im ersten Argument genannte Datei. Als Zwischenspeicher wird eine temporäre Datei genutzt, deren Namenserweiterung sich aus der PID ergibt (wegen der Einmaligkeit):

```
# Aufgabe : die vs (Vorspann) Prozedur
set  td = /tmp/vs.$$   # temp. Datei (einmaliger Dateiname)
set  ziel = $argv[1]   # Zieldatei aus Kommando übernehmen

cat - $ziel > $td      # liest zunächst von stdin, dann $ziel
mv  $td  $ziel         # ersetzen
```

Erweiteren Sie die **vs** Prozedur um folgende Funktionen:

- Eine Prüfung, ob eine Zieldatei genannt wird; Abbruch mit Fehleranzeige wenn nicht.
- Ein Test, ob die Zieldatei geschrieben werden kann (-w $ziel), mit entsprechender Fehlermeldung falls nicht.
- Ein Test, ob die Zieldatei bereits existiert (-e $ziel), mit entsprechender Fehlermeldung falls nicht.

1.4. Schreiben Sie eine Kommandoprozedur, die als Rahmen für weitere Prozeduren genutzt werden kann und folgende Funktionalität aufweist:

- zwei Parameter werden erwartet: eine Eingabe- (Quelle) und eine Ausgabedatei (Ziel)
- werden die Parameter nicht expliziet bei Kommandoeingabe (Argumente) genannt, sollen sie interaktiv abgefragt werden.
- die Quelldatei muß vorhanden und lesbar sein, die Zieldatei darf nicht vorhanden sein.

2 Aufgaben zum Thema C

Im folgenden (Aufgaben 1. - 2.) werden einige kleinere Aufgaben gestellt, die als praktische Übung der beschriebenen Sprachkonstrukte dienen sollen. Erstellen Sie die Programme zu Hause und geben Sie den Code dann zum 1. Versuchstermin ein und testen Sie ihn zusammen mit Ihrem Betreuer.

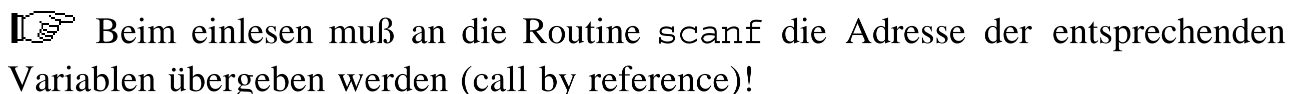
Wichtig: in den Programmen (C-Quellcode) müssen alle Anweisungen in nachvollziehbarer Weise kommentiert werden!

2.1. Schreiben Sie ein kleines C-Programm, daß die Größe (in Byte) der Datentypen `char`, `short`, `int` und `long` mit `printf` in folgendem Format (Beispiel für Datentyp `char`) ausgibt:

Der `Variablentyp char besteht aus 1 Byte(s).`

2.2a. Es soll ein einfaches Hauptprogramm geschrieben werden, das in einer Schleife 10 Zahlen einliest und sie dann wieder ausgibt. Verwenden Sie dazu die Funktionen scanf und printf mit entsprechenden Formatdefinitionen für dezimale Ein-/Ausgabe. Verwenden Sie eine `for` Schleife und definieren Sie die verwendeten Variablen im Rumpf der `main` Routine. Zum Einlesen der Zahlen soll ein Array verwendet werden. Der unten stehende Rahmen soll benutzt werden.

```
#include <stdio.h>        /* Einbinden der Standard E/A
aus der Bibliothek */
main()           /* Hauptprogramm ohne Parameter */
{
long int i;
long int feld[i];
...
}
```


☞ Beim einlesen muß an die Routine `scanf` die Adresse der entsprechenden Variablen übergeben werden (call by reference)!

2.2b. Ergänzen Sie nun das erstellte Hauptprogramm durch eine Subroutine (Funktion) zum Ausgeben der Zahlen. Als Parameter soll der Wert der auszugebenden Variablen an die Funktion übergeben werden (call by value).

2.2c. Führen Sie nun einen zweiten Parameter ein, der "by reference" übergeben wird. Sie müssen ihn als Zeiger auf den entsprechenden Datentyp (`long int`) definieren! Die Funktion soll den Inhalt dieses Parameters auf 1 setzen, wenn die übergebene Zahl (erster Parameter) >= 0 war, sonst auf 0. Im Hauptprogramm soll dann der zurückgelieferte Wert ausgedruckt werden.

2.2d. Setzen Sie nun auch die Präprozessor-Statements `#define` und `#include` ein. Definieren Sie sich als Feldgrenze des Arrays im Hauptprogramm eine Konstante und verwenden Sie diese auch bei der Ein-/Ausgabeschleifen. Schreiben

Sie das Unterprogramm in eine getrenntes File und binden Sie dieses mit `#include` in die Hauptprogrammdatei ein.

2.3. Programmieren Sie eine einfache Studenten-Datenbank mit folgende Merkmale:

a. Definieren Sie eine C-Datenstruktur (`struct`) "Student" in der Form:

Name, Vorname, Matrikelnummer

b. Erstellen Sie ein Feld von 10 Elementen mit der in a. erzeugten Struktur.

c. Erweitern Sie die Student-Struktur um einen Verweis/Zeiger (`pointer`) auf ein Nachfolge-Element.

d. Erzeugen Sie mit der in 3c. erstellten Struktur eine einfach verkettete Liste von 10 Studenten.

e. Editieren Sie 3a-d in C und überprüfen Sie die Korrektheit mit dem Compiler.

☞ Schreiben Sie dazu ein Programm das nur Datenstrukturen definiert!

2.4.

a. Schreiben Sie zu 3a.+b. ein C Programm zum Einlesen von 10 Studentendatensätzen. Verwenden Sie dazu die Funktionsaufrufe scanf und printf. Achten sie auf korrekte Parameterübergabe!

b. Übersetzen Sie das Programm, entfernen Sie Syntax-Fehler und testen Sie das Programm.

c. Schreiben Sie nun wie in 4a. zu 3c.+d. ein Programm zum Einlesen von 10 Studentendatensätzen. Achten Sie auf die Verzeigerung!

d. Übersetzen Sie das Programm, entfernen Sie Syntax-Fehler und testen Sie das Programm.

e. Wie reagieren Ihre Programme bei Eingabefehler (z.B. Steuerzeichen)?

2.5. Optionale Aufgabe (nicht Pflicht)

a. Entwerfen Sie einen Algorithmus und schreiben Sie das dazugehörige C-Prozeduren für folgende Funktionen:

- Erfassen eines neuen Studenten
- Suchen nach der Matrikelnummer
- Sortieren nach der Matrikelnummer
- Löschen eines Studenten-Datensatzes

b. Schreiben Sie ein Menü zu 5a. mit dem Zusatz "Ende", als C-Hauptprogramm.

c. Integrieren Sie alle bisherigen Entwürfe zu einem lauffähigem C-Programm.

d. Schreiben Sie das Programm aus 5c. so um, daß die Funktionen auf eine anzugebende Datei ausgeführt werden. (Eingabedatei - Menü - Ausgabedatei)

4. Literaturverzeichnis

1. Bosse, Kai : Programmieren mit C, Praktische Beispiele für PC-Anwendungen. Berlin und München, Siemens AG, 1987.

2. Comer, Douglas E. : Internetworking with TCP/IP, Vol. 1: Principles, Protocols, and Architecture. Prentice-Hall Internat. Editions, 1991.

3. Feuer, Alan R. : C-Puzzlebuch: C-Beispiele zum Wühlen und Wundern. München Wien : Hanser, 1990.

4. Gulbins, Jürgen; Obermayer, Karl : Unix System V.4, Begriffe, Konzepte, Kommandos, Schnittstellen, Springer, 1995.

5. Krüger, Gerhard : Telematik I. Institut für Telematik, Universität Karlsruhe, Skriptum zum Vorlesung. Studentenwerk.

6. Krüger, Gerhard : Telematik II. Institut für Telematik, Universität Karlsruhe, Skriptum zum Vorlesung. Studentenwerk.

7. Stevens, W. Richard : Programmieren von Unix-Netzen. München Wien : Hanser, 1992.

8. Wettstein, Horst: Architektur von Betriebssystemen. München : Hanser, 1984.

Anhang A : vi Kommando-Übersicht

Anfangen/Beenden

% **vi** *name* Editieren von *name* am Anfang der Datei
% **vi** +***n*** *name* … bei Zeile *n*
% **vi** + *name* … am Ende der Datei
% **vi -t** *tag* Beginne bei tag
% **vi** +/***pat*** *name* Suche nach pat
% **view** *name* Nur lesen
ZZ Beenden mit Speichern der Änderungen
^Z Vorläufiges Abbrechen

Vi Zustände

Kommando Modus Anfangszustand. Alle andere Zustände fallen zurück in den Kommando Modus

Einfügen Einschalten mit **a i A I o O c C s S R.**
Normaler Text wird anschließend mit **ESC** beendet.

Letzte Zeile Liest Eingaben für **: / ?** oder **!** ; zu beenden mit **ESC** oder **CR** zum Ausführen. Abbrechen mittels Interrupt.

Einfache Kommando´s

dw Wort löschen (delete word)
de . . . unter beibehalten der Punktuation
dd Zeile löschen
3dd 3 Zeilen löschen
i*abc***ESC** Einfügen des Textes *abc*
cw*neu***ESC** Wort ersetzen durch *neu*

Datei-Zugriff

:w Sichern (write) der Änderungen
:wq Sichern und beenden (quit)
:q Beenden
:q! Beenden ohne Rücksicht auf Änderungen
:e *name* Editiere die Datei *name*
:e! Neu editieren, bisherige Änderungen verwerfen
:e +*name* Editieren am Ende der Datei anfangen

:e +*n* Editieren, bei Zeile *n* anfangen
:w *name* Ausgabe auf Datei name
:w! *name* Überschreibe Datei name
:sh Die Shell starten, dann weiter editieren
:!*cmd* Kommando *cmd* ausführen, dann weiter editieren
:n Nächste Datei aus der Argumentenliste editieren
:n *args* Neue Argumentenliste zur Verfügung stellen
:f Aktuelle Datei und Zeile zeigen

Positionieren in der Datei

^F Vorblättern um einen Bildinhalt (forward)
^P Zurückblättern . . .
^D Halber Bildinhalt vor
^U Halber Bildinhalt zurück
/muster Nächste Zeile mit *muster* finden
?*muster* Letzte Zeile mit *muster* finden
n letztes / oder ? wiederholen
N letztes / oder ? zurück
/muster/+n *n*-te Zeile nach *muster*
?*muster*?-*n* *n*-te Zeile vor *muster*
% Suche zugehörige Klammer (,) , { oder }

Darstellung

^L Löschen und Auffrischen des Fensters/Bildschirms
^E Fenster um 1 Zeile zurück
^Y Fenster um 1 Zeile vor

Zeilenorientiertes Positionieren

H Erste Fensterzeile
L Letzte Fensterzeile
M Mittlere Fensterzeile
\+ Nächste Zeile, erstes darstellbare Zeichen
- Zeile zurück, erstes darstellbare Zeichen
CR Return, wie +
↓ oder **j** Nächste Zeile, selbe Spalte
↑ oder **k** Zeile zurück, selbe Spalte

Zeichenorientiertes Positionieren

↑ erstes darstellbare Zeichen
0 Zeilenanfang
$ Zeilenende
h oder → vor
i oder ← zurück
^H wie ←
Leerzeichen wie →
f*x* suche nächstes *x*
F*x* suche letztes *x*

Worte, Sätze, Abschnitte

w Nächstes Wort
b Wort zurück
e Ende des Wortes
) Ende des Satzes
} Ende des Abschnitts
(Letzter Satz
{ Letzter Abschnitt
W Wort durch Leerzeichen begrenzt
B Wort zurück
E Ende des Wortes

Einfügen / Ersetzen

a Einfügen nach Cursor
i Einfügen vor Cursor
A Anhängen am Zeilenende
o Öffne nächste Zeile
O Öffne obere Zeile
r*x* Einzelnes Zeichen durch x ersetzen
R Zeichen ersetzen

Operatoren (zweifach angeben, um Zeilen zu bearbeiten)

d Löschen (delete)
c Ändern (change)
< Links schieben
> Rechts schieben
y Zeilen im Puffer übernehmen

Verschiedenes

C Rest der Zeile ändern
D Rest der Zeile löschen
s Zeichen ersetzen
S Zeilen ersetzen
J Zeilen zusammenfügen (join)
x Zeichen löschen
X . . . vor dem Cursor
Y Zeilen im Puffer übernehmen

Ausschneiden und Einsetzen

p Zeilen aus Puffer einsetzen
P Vorne einsetzen

Rückgängig machen, wiederholen

u Mache letzte Änderung rückgängig
U Aktuelle Zeile wiederherstellen

Anhang B: EMACS Kommandos

C- entspricht gleichzeitiges Drücken der Ctrl und nachfolgende Taste

M- entspricht der Meta- bzw. Escape und nachfolgender Taste

Aufruf

emacs Aufruf	emacs [dateiname]
emacs zwischendurch verlassen	C-z
Ende der Sitzung	C-x C-c

Dateien

Datei in den emacs-Buffer einlesen	C-x C-f
Datei zurückschreiben	C-x C-s
Dateiverzeichnis editieren	C-x d

Benutzerhilfen

Allgemeines Hilfsangebot	C-x ?
Interaktiver Einführungskurs	C-h t
Hypertext-Infosystem	C-h i
Zeige alle Kommandos zu einem Schlüsselwort	C-h a
Zeige alle Funktionen zu einer Tastenkombination	C-h c
Erkläre eine bestimmte Funktion	C-h f
Modus spezifische Informationen	C-h m

Für den Notfall

Kommando abbrechen	C-g
Letzte Änderung zurücknehmen	C-x u oder C-_
Bildschirm neu schreiben	C-l

Suchkommandos

Inkrementelles Suchen vorwärts	C-s
Inkrementelles Suchen rückwärts	C-r
Suche nach regularem Ausdruck	C-M-s
Beende inkrementelles Suchen	ESC

Bewegen im Text

Bewegungseinheit	vorwärts	rückwärts
Einzelnes Zeichen	C-b	C-f
Wort	M-b	M-f
Zeile	C-p	C-n
Zeilenanfang/ende	C-a	C-e
Satz	M-a	M-e
Absatz	M-[	M-]
Seite	C-x[	C-x]
Anfang/Ende des Buffers	M-<	M->
Bildschirm hoch/runter	M-v	C-v
Bildschirm links/rechts	C-x<	C-x>

Löschen

Bereich	C-w
Letzte gelöschte Einheit einfügen	C-y

Zu löschende Einheit	vorwärts	rückwärts
Einzelnes Zeichen	DEL	C-d
Wort	M-DEL	M-d
Bis Zeilenanfang/ende	M-0 C-k	C-k
Satz	C-x DEL	M-k

Markieren

Marke setzen	C-@ oder C-SPC
Cursorposition mit Marke vertauschen	C-x C-x
Markiere Absatz	M-h
Markiere Seite	C-x C-p
Markiere ganzen Buffer	C-x h

Interaktives ersetzen

Zeichenkette	M-%
Regulärer Ausdruck	M-x *query-replace-regexp*
Gültige Eingaben im Ersetzungsmodus:	
Hier ersetzen, weitergehen	SPC
Hier ersetzen, nicht weitergehen	,
Hier nicht ersetzen, weitergehen	DEL
Ab hier alles ersetzen	!
Ersetzungen abbrechen	ESC

Vertauschen

Einzelzeichen	C-t
Wörter	M-t
Zeilen	C-x C-t

Fenster

Lösche alle anderen Fenster	C-x 1
Lösche dieses Fenster	C-x 0
Fenster vertikal aufteilen	C-x 2
Fenster horizontal aufteilen	C-x 5
Cursor ins nächste Fenster	C-x o
Fenster verlängern	C-x ^
Fenster verbreitern	C-x }

Buffer

Wähle anderen Buffer	C-x b
Liste alle Buffer	C-x C-b
Buffer löschen	C-x k

Minibuffer Kommandos

Soweit wie möglich vervollständigen	TAB
Vervollständige dieses Wort	SPC
Vervollständigen und Ausführen	RET
Zeige mögliche Vervollständigungen	?

Tags

Suche Tag	M-
Suche in alle Dateien der Tag-Tabelle	M-x Tags-Suche
Ersetze gemäß Tags-Query in allen Dateien	M-x Tags-Query
Weiter mit Tag-Operation	M-,

Shells

Führe Shell Kommando aus	M-!
Interaktive Shell im neuen Fenster	M-x shell

Makros

Anfang Makrodefinitionen	C-x(
Ende Makrodefinitionen	C-x)
Makro ausführen	C-x e