Ein erstes eigenes Unix-Programm (Hello World)

helloworld.c (~/Documents/Lehre/Werkzeuge/live_examples) – VIM

```c
// gcc -std=c99 helloworld.c -o helloworld

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

int main( int argc, char ** argv )
{
    printf("Hello World!\n");

    for ( int i = 0; i < argc; i ++ )
    {
        printf("%d: %s\n", i, argv[i] );
    }

    return 0;
```


Prozesse aus Sicht der Shell

- 3 Zustände eines Prozesses (aus Sicht der Shell)
  - Foreground: Default
    - Ausgabe (stdout) des Prozesses erscheint im Terminal-Fenster
    - Eingabe (stdin) des Prozesses kommt vom Keyboard
  - Background:
    - Ausgabe erscheint im Fenster
    - Eingabe nicht erlaubt
  - Gestoppt:
    - Prozeß schläft

Kommandos zur Prozesskontrolle

Befehl	Funktion
ps	Prozesse anzeigen
ps -edfjw	Alle Prozesse anzeigen
ps -auxw	dito für einige andere Unix-Varianten
kill pid	Prozeß mit PID **pid** abbrechen (wie Ctrl-C)
kill -9 pid	... wenn der Prozeß trotzdem nicht aufhören will ☺
command ... &	Prozeß im Hintergrund starten
jobs	Prozesse im Hintergrund anzeigen
Ctrl-C	Foreground-Prozeß abbrechen (interrupt)
Ctrl-Z	Foreground-Prozeß anhalten (stoppen)
fg	Zuletzt angehaltenen Prozeß im Foreground weiterlaufen lassen
bg	Angehaltenen Prozeß im Background weiterlaufen lassen
Ctrl-S	Ausgabe des Foreground-Prozesses anhalten (Pr. läuft weiter!)
Ctrl-Q	Ausgabe weiterlaufen lassen
top	tabellarische Ansicht aller Prozesse und deren CPU-Verbrauch

Bash vs. tcsh

- Annahme im folgenden: **bash** ! (ksh / sh)
  - Häufig ist die Tcsh für die interaktive Commandline-Nutzung komfortabler
- Wie stellt man fest, welche Shell man gerade hat?
  - Einfach '**set**' eingeben:

Resultat unter tcsh

```
% set
...
tcsh        6.14.00
...
version  tcsh 6.14.00 ...
```

Resultat unter bash

```
$ set
...
BASH=/bin/bash
...
BASH_VERSINFO=([0]="3" ...
BASH_VERSION='3.2.17(1)-...
```

- Wer (t)csh hat, kann eine Bash starten mit dem Kommando **bash**

Shells: Was passiert mit einem Kommando?

- Shell durchläuft folgenden Zyklus mit jedem eingegebenen Befehl:
  1. Kommando wird aus der Eingabezeile oder einem Script gelesen
  2. Aliases werden expandiert
  3. Variable Substitutions werden vorgenommen
  4. Wildcards (File-Patterns) werden expandiert
  5. Das Kommando wird ausgeführt:
     - Entweder von der Shell selbst (built-in commands)
     - Oder in einem neuen Prozeß

1. Aliases

- Ersatz für häufige / längliche Kommandos
- Beispiel:

```
% alias mo less
% alias uni "ssh –l zach –X hera.uni-bonn.de
% mo program.cpp
% uni
```

- **alias** zeigt alle Aliases an

Alias 'uni' wird ersetzt ...
Alias 'mo' wird ersetzt ...
Definiert Alias 'uni'
Definiert Alias 'mo'

2. Variablen

- Variable = Name + Wert, Wert = Zeichenkette
- 2 Arten:
  - Normale Shell-Variablen
  - Environment-Variablen,
    sind auch in Kind-Prozessen (child process) bekannt
- Setzen:

```
$ export TMP="/tmp"
$ tmpdir="/home/stud/zach/tmp"
```

  - Achtung: **kein** Space beim =-Zeichen !
- Verwenden des Wertes einer Shell-Variablen:

```
% cp file ${tmpdir}
% echo ${TMP}
```


- Anhängen eines Strings an eine bestehende Shell-Variable:

```
$ PATH=${PATH}:${HOME}/bin
$ PATH=${HOME}/bin:${PATH}
```

- Ausdrucken des Wertes einer bestimmten Variablen:

```
$ echo $var
$ echo $VAR
```

- Ausdrucken aller Environment-Variablen:

```
$ printenv
```


Komplexere Variablen-Ersetzung

- Suffix bzw. Präfix löschen:

```
${VAR%suffix}
${VAR#prefix}
```

Löscht Schluß der Variable **VAR**, falls er gleich "**suffix**" ist

Löscht Anfang der Variable **VAR**, falls er gleich "**prefix**" ist

- Beispiel:

```
% file="program.cpp"
% mv ${file} ${file%.cpp}.c
% mv ${file} ${file%.p}.c
% mv ${file} ${file#prog}Prog
```

wird ersetzt zu

```
% mv program.cpp program.c
% mv program.cpp program.cpp.c
% mv program.cpp Program.c
```

4. File Patterns

- File-Pattern = File-Name mit Wildcards: ***** **?** **[]**

Wildcard	Bedeutung
?	Genau ein beliebiges Zeichen
*	Beliebig viele beliebige Zeichen (auch 0)
{0,1,2}	Genau ein Zeichen aus der Menge {0,1,2}
[0-9]	Genau ein Zeichen aus der Menge {0,...,9}
[a-zA-Z0-9]	Genau ein Zeichen aus der Menge {a,...,z,A,...,Z,0,...,9}
[^0-9]	Genau ein Zeichen *nicht* aus der Menge {0,...,9}

- Beispiele:

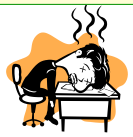
```
% ls *.cpp *.h
% ls [0-9][0-9]*.ppt
% ls ???R??
% ls *[^a-zA-Z0-9_.,-]*
```

- Achtung: das Kommando sieht nur eine Liste von File-Namen!

Kleine Warnung zu **rm**

Task: Shoot Yourself in The Foot
The proliferation of modern programming languages (all of which seem to have stolen countless features from one another) sometimes makes it difficult to remember what language you're currently using. This handy reference is offered as a public service to help programmers who find themselves in such a dilemma.

```
% ls
foot.c foot.h foot.o toe.c toe.o
% rm * .o
rm: .o no such file or directory
% ls
%
```



Quotation

- Achtung: das Kommando sieht die Wildcards nie!
- Shell expandiert Wildcards
- Quotation verhindert, daß Wildcards (allg. Meta-Zeichen) von der Shell expandiert werden
- Arten:
  - \ verhindert die Expansion des folgenden Zeichens

```
% echo \*.ppt \$PATH
```

  - "**...**" verhindert die Expansion der Wildcards, ersetzt aber Variablen

```
% echo "*.ppt $PATH"
```

  - '**...**' verhindert jegliche Expansion (Wildcards, Variablen, ...)

```
% echo '*.ppt $PATH'
```


5. Wie die Shell ein Kommando ausführt

1. Built-in: Shell führt Kommando selbst aus
   - Beispiel: **echo**
2. Sonst: externes Programm
   - Beispiel: **ls dir**
3. **command** in **PATH** (Environment-Variable) suchen
   - Falls nicht gefunden, Fehlermeldung, fertig
4. Kind-Prozeß erzeugen
   - Erinnerung: erbt Environment des Vater-Prozesses (= Shell)
5. Argumente (Zeichenketten) dem Prozeß bereitstellen
   - Argumente werden durch Space (i.a.) getrennt
6. Warten bis Kind-Prozeß beendet

Suchpfade

- Suchpfad = Environment-Variable mit Liste von durch **:** (Doppelpunkt) getrennten Verzeichnissen
- Beispiel:

```
% echo $PATH
.:/usr/bin:/bin:/usr/local/bin:/home/zach/bin
```

- Kommandos werden in **PATH** gesucht
  - Test: File mit Namen des Kommandos im ersten Verzeichnis und dieses ist *executable*? → ausführen
  - Sonst: nächstes Verzeichnis in PATH untersuchen ...
- Analog für Man-Pages und andere

Wichtige Suchpfade

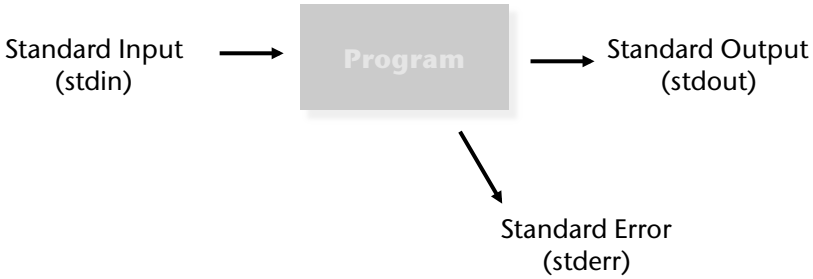
Variable	Bedeutung
PATH	Suchpfad für Kommandos
MANPATH	Suchpfad für Man-Pages
LD_LIBRARY_PATH	Suchpfad für dynamisch gelinkte Libraries

File-"Viewer"

- **more, less**
  - Interaktiv, zeigen *jeden* File-Typ im Terminal-Fenster an
  - Unter Linux/Mac ist **more** = **less**
  - Suchen mit '/', weitersuchen mit 'n', rückwärts 'N', u.v.m.
  - Environment-Variable: **setenv PAGER 'less -I'**
- Nicht-interaktiv:
  - **cat** ("concatenate" = aneinanderhängen)
  - **head** , **tail** = einige der ersten / letzten Zeilen anzeigen

Standard-I/O

- Shell etabliert 3 "Kanäle" zu / vom Prozeß:



- Default-mäßig mit Terminal (= Fenster & Keyboard) verbunden

Redirection

- Verbindet Standard-I/O mit Files
- Prozeß (Programm) bemerkt davon nichts!

Redirection	Bedeutung
> file	stdout wird nach file geschrieben
>> file	stdout wird an file angehängt
>& file	**stdout** und **stderr** umlenken
< file	Programm liest aus file, nicht von Keyboard
...	Shell bietet noch viele weitere Möglichkeiten

- Beispiel:

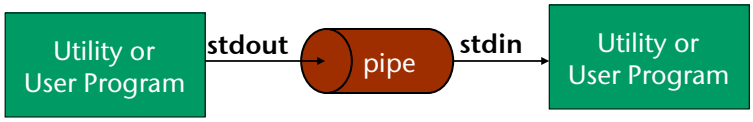
```
% ls –1 > dir-listing
% sort dir-listing
```

```
% ps -edfyl > procs.log
```

```
% bc -l < berechnung.txt
```


Pipelines

- Effiziente Möglichkeit, komplexere Kommandos aus einfacheren zusammenzubauen!
- Selber Mechanismus wie bei Redirection, um Prozesse miteinander zu verbinden:



- Syntax:

```
% command | command | command | ...
```

Beispiele

- Listings weiterverarbeiten:

```
% ls  -1 | sort | more
% ls -1 *.cpp | wc –l
% ls -1 *.cpp | sort > sorted-dir
```

```
% ls -1 | tr -s "_"
```

Tip:
Bauen Sie eine Pipeline immer eine Stufe nach der anderen auf!!

Die Unix-Philosophie

- "Small is beautiful"
- Make each tool do *one* thing only
- Make it do it well
- Read from stdin, write to stdout (if sensible)
- Use ASCII-Files

Filter

- Sind hauptsächlich dazu gedacht, um damit Pipelines aufzubauen
- Generelle Verwendung: Filter lesen von stdin, schreiben auf stdout
- Arbeiten fast immer zeilenweise

Utility	Funktion
cut	Felder oder Zeichenspalten ausschneiden
fmt	Text umformatieren
sort	Zeilenweise sortieren (auch nach Teil-Key)
uniq	Duplikate entfernen
wc	Zeichen, Wörter, und Zeilen zählen
tr	Zeichen ersetzen
grep	Zeichenketten in der Eingabe suchen (s.u.)
rev	Reihenfolge der Zeichen umkehren (zeilenweise)
bc	"Taschenrechner" mit unendlich hoher Genauigkeit

Beispiele

- Alle Filegrößen in einen File ausgeben:

```
% ls -l | tr -s " " | cut -d' ' -f5 > sizes.txt
```

- Nur die letzten paar Zeilen eines langen Directory-Listings:

```
% ls -l | tail -10
```

- Alle top-level Verzeichnisse in einem Zip-Archiv auflisten (jedes Vorkommen nur 1 Mal)

```
% unzip -l Archive.zip | tr -s " " | cut -d' ' -f5 | cut -d'/' -f1| sort | uniq
```