Programmierkurs Vorlesung 1

Kommandozeile und Compiler

Andreas Naumann

Institut für Wissenschaftliches Rechnen
Universität Heidelberg

27. März 2023

Original Authors: Lorenz Braun, Dr. Ole Klein, Dr. Steffen Müthing

Inhalte

- Praktisches Programmieren mit C++
- Umgang mit Unix-Kommandozeile (Linux, macOS)
- Vorstellung von wichtigen Themen und Werkzeugen rund ums Programmieren
  - Versionsverwaltung
  - Buildsysteme
  - Fehlersuche (Debugging)

- "Fördern durch Fordern":
  - Aufgaben möglichst knapp außerhalb der "Comfort Zone"
  - Dafür sehr kulante Bewertung der Leistungen, sofern Ansätze und Bemühen erkennbar sind
- "Learning by doing":
  - Vorlesung dient der Grundlage
  - Das Verständnis kommt durch Praxis

Modul — Lernziele und Inhalt

Lernziele:

Die Studierenden können selbstständig Programme und Lösungen von Programmieraufgaben in C++ entwerfen, realisieren und testen[, und] sind in der Lage mit gängigen Programmierwerkzeugen und Tools unter Linux umzugehen.

Inhalt:

Die Lehrveranstaltung vertieft die Programmierkenntnisse aus dem Modul Einführung in die Praktische Informatik (IPI). Im Vordergrund steht der Erwerb praktischer Fähigkeiten. Die Studierenden lernen algorithmische Lösungen systematisch in Programme umzusetzen. Es wird die Programmiersprache C++ unter dem Betriebssystem Linux verwendet. [. . .]

Modul — Voraussetzungen und Prüfungsmodalitäten

Voraussetzungen: keine (auch nicht IPI!)

Prüfungsmodalitäten:

Erfolgreiche Teilnahme an den Übungen und erfolgreiche Teilnahme an einer schriftlichen Prüfung. Im Wintersemester wird am Ende der Vorlesungszeit eine Klausur angeboten. Wird diese nicht bestanden so kann die Prüfungsleistung in einer zweiten Klausur vor Beginn der nächsten Vorlesungszeit erbracht werden. Im Sommersemester wird nur eine Klausur angeboten.

Organisation

Termine

Vorlesung	täglich, 9:15–11:30 INF 205, Hörsaal
Übungen	täglich, 13–15 und 15–17 ct INF 205, PC-Pool SW 1 und SW 2 Bei Bedarf noch 17 – 19 ct
Klausur	14. April, Zeit TBA

Informationen, Übungsblätter etc.

https://scoop.iwr.uni-heidelberg.de/teaching/2023ss/grundkurscpp/

Tutorien und Vorlesung

- Kern der Veranstaltung sind die Tutorien
- ECTS-Punkte für Tutorien und Heimarbeit
- Klausur basiert auf den behandelten Themen / Techniken

- Keine Anwesenheitspflicht (z.B. falls Stoff schon bekannt)
- Folien der Vorlesung werden online veröffentlicht
- Klausurrelevant nur in dem Sinne, dass natürlich der Stoff zu den Übungen passen wird

Übungen

- Um Programmieren zu lernen, muss man programmieren!
- Übungszettel enthalten ausschließlich Programmieraufgaben
- Neue Übungszettel nach der Vorlesung
- Darauffolgendes Tutorium:
  - Vorstellung der Lösungen
  - Fragen an Tutoren zum Blatt
- Freitags: Ausgabe eines Pflichtblatts
- Abgabe: bis Montag, 13:00 Uhr

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?
- Was versteht ihr unter folgendem Shell-Kommando?

```
g++ -Wall -O3 test.cc | grep "error:|warning:" > log.txt
```

1. Bahnhof

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?
- Was versteht ihr unter folgendem Shell-Kommando?

```
g++ -Wall -O3 test.cc | grep "error:|warning:" > log.txt
```

1. Bahnhof
2. Es wird g++ ausgeführt und "error..." nebeneinander ausgegeben

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?
- Was versteht ihr unter folgendem Shell-Kommando?

```
g++ -Wall -O3 test.cc | grep "error:|warning:" > log.txt
```

1. Bahnhof
2. Es wird g++ ausgeführt und "error..." nebeneinander ausgegeben
3. Es werden nur Fehler und Warnungen des Compilers in eine Datei geschrieben

Bestandsaufnahme

- Wer hat schon einmal Linux / macOS verwendet?
- Wer hat schon einmal die Kommandozeile verwendet?
- Wer hat schon einmal programmiert?
  Javascript? Java? C#? C? C++? Python?
- Was versteht ihr unter folgendem Shell-Kommando?

```
g++ -Wall -O3 test.cc | grep "error:|warning:" > log.txt
```

1. Bahnhof
2. Es wird g++ ausgeführt und "error..." nebeneinander ausgegeben
3. Es werden nur Fehler und Warnungen des Compilers in eine Datei geschrieben

Bestandsaufnahme

- Was gibt folgender C++-Code aus?

```
auto v = std::array<int>{1,2,3,4};
for (auto x : v)
  x *= x;
std::cout << std::accumulate(v.begin(),v.end(),0)
<< std::endl;
```

1. accumulate und endl

Bestandsaufnahme

- Was gibt folgender C++-Code aus?

```
auto v = std::array<int>{1,2,3,4};
for (auto x : v)
  x *= x;
std::cout << std::accumulate(v.begin(),v.end(),0)
<< std::endl;
```

1. accumulate und endl
2. 1,2,3,4

Bestandsaufnahme

- Was gibt folgender C++-Code aus?

```
auto v = std::array<int>{1,2,3,4};
for (auto x : v)
  x *= x;
std::cout << std::accumulate(v.begin(),v.end(),0)
<< std::endl;
```

1. accumulate und endl
2. 1,2,3,4
3. 30

Bestandsaufnahme

- Was gibt folgender C++-Code aus?

```
auto v = std::array<int>{1,2,3,4};
for (auto x : v)
  x *= x;
std::cout << std::accumulate(v.begin(),v.end(),0)
<< std::endl;
```

1. accumulate und endl
2. 1,2,3,4
3. 30
4. 10

Warum UNIX / Linux?

Die meisten Arbeitsgruppen in Mathematik, Physik und Informatik verwenden zumindest in Teilen Linux. Daher werden sich die meisten von Ihnen früher oder später damit auseinandersetzen müssen.

Linux ist mit Abstand das am häufigsten genutzte Unix. Über mit Linux betriebene Webserver werden nicht nur große Teile des Internet zur Verfügung gestellt, grob 95% der 500 schnellsten Rechner der Welt basieren auf Linux, und alle unter den zehn schnellsten. Wer mit Rechnern wissenschaftlich arbeiten will, kommt an Unix nicht vorbei.

Darüber hinaus ist Linux durch seine offene Natur und die vielen Open-Source-Komponenten eine hervorragende Spielwiese für angehende Informatiker. Für jedes Programm, das auf einem normalen Linux-Desktop installiert ist, können Sie den Quellcode herunterladen und lernen, wie es programmiert wurde!

Was ist UNIX?

Im engeren Sinne Unix (1969), ein Betriebssystem, das viele Funktionen einführte, die in heutigen Betriebssystemen selbstverständlich sind.
Im weiteren Sinne jedes Betriebssystem, das sich an die UNIX-Interfaces und -Spezifikation hält:

- Free,Net,OpenBSD, basierend auf einer Unix-Variante von der UC Berkeley
- macOS, das auf Teilen von FreeBSD und NetBSD basiert
- iOS, aus macOS hervorgegangen
- illumos/OpenIndiana, basierend auf Solaris und System V
- Linux (1991), streng genommen kein UNIX, aber weitestgehend kompatibel
- Android, von Google stark modifiziertes Linux
- Viele embedded systems, oft auf Basis von BSD oder Linux, in Netzwerkroutern, Fernsehern, Robotern, Autos, Raketen, ...

Linux-Installation

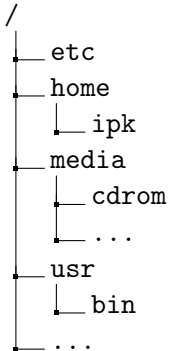
- Fester Teil des Moduls ist die Entwicklung von Kompetenz im Umgang mit
  - Linux-Distribution(en)
  - Shell
- Dazu benötigen Sie natürlich Zugang zu einem solchen System
- Möglichkeiten für Sie:
  - Linux/macOS evtl. bereits vorhanden
  - Für Interessierte/Fortgeschrittene: Linux parallel zu Windows installieren
  - Für Nutzer von Windows 10: Windows Subsystem for Linux (WSL) installieren
  - Ansonsten: Installation einer virtuellen Maschine (VM)

Installationsanleitungen

Anleitungen (WSL und VM) sind auf der Webseite zu finden

Dateisystem I

- Dateien liegen in Verzeichnissen
- Verzeichnistrenner unter Unix:
  `home/user` statt `C:\Users\Name`
- Groß- und Kleinschreibung unter Linux relevant:
  `test.txt` $\neq$ `Test.txt`
- Linux kennt keine Laufwerksbuchstaben, alle Verzeichnisse haben eine gemeinsame Wurzel `/`



Dateisystem II

- Jeder Benutzer hat ein home directory für eigene Dateien, normalerweise in `/home/<username>`
- Normale Benutzer haben keine Schreibrechte außerhalb ihres home directories
- Der Administrator (heißt unter UNIX root) darf überall lesen und schreiben
- Jedes Programm hat ein Arbeitsverzeichnis (working directory)
- Dateizugriffe immer relativ zu diesem Verzeichnis
- Arbeitsverzeichnis kann gewechselt werden
- Spezielle Verzeichnis-Namen

  `.` Das aktuelle Verzeichnis

  `..` Das übergeordnete Verzeichnis

  `/` Das Wurzel-Verzeichnis

  `~` Das Home Directory des aktuellen Benutzers (funktioniert nicht in jedem Kontext)

Shell I

- Linux verfügt über eine graphische Oberfläche mit der es sich ähnlich bedienen lässt wie die bekannten Desktops von Windows und macOS.
- Es gibt eine Vielzahl an Benutzeroberflächen. Auch welche die Windows nachahmen, macOS nachahmen oder gänzlich neue Wege gehen.
- Für Aufgaben wie Programmieren ist es jedoch weiterhin sinnvoll, sich mit dem schwarz-weißen Terminal-Fenster und der darin laufenden Shell zu befassen:
  - Automatisierung von monotonen und repetitiven Arbeiten
  - Mit etwas Übung ist man bei vielen Aufgaben schneller als in der graphischen Oberfläche (GUI)
  - Viele UNIX-Programme haben keine GUI und können direkt nur über die Shell aufgerufen werden, z.B. der C++-Compiler, den wir zum programmieren in diesem Kurs brauchen

Shell II

Möglichkeiten für erste Schritte mit der Shell:

- Linux / macOS, falls vorhanden (klar!)
- die Windows PowerShell (eingeschränkte Kompatibilität)
- ein Shell-Emulator online

Online Shells

bellard.org Virtuelles Fedora 29 (Fabrice Bellard)
https://bellard.org/jslinux/vm.html?cpu=riscv64&url=fedora29-riscv-2.cfg&mem=256

copy.sh Virtuelles ArchLinux (Fabian Hemmer)
https://copy.sh/v86/?profile=archlinux

Shell III

```
[ipk@vm ~] _
```

- Wichtige Informationen am Anfang der Zeile (prompt)
  - Benutzername
  - Rechnername
  - aktuelles Verzeichnis
- Ausgabe des vollen Pfades mit `pwd`:

```
[ipk@vm ~] pwd
/home/ipk/
```

- Shell beenden mit `exit`:

```
[ipk@vm ~] exit
<Fenster schließt sich>
```

Kommandos in der Shell

```
[ipk@vm ~] cmd -sv --opt --op2 arg1 arg2 ...
```

- Die meisten Kommandos haben ein einheitliches Interface
  - Am Anfang steht der Name des Kommandos (der Dateiname des Programms)
  - Danach folgen Optionen (beginnen mit `"-"`)
  - Am Ende stehen die Argumente ohne `"-"`
- Argumente bestimmen, worauf das Kommando angewendet wird
- Optionen verändern, wie das Kommando arbeitet. Wichtige Optionen haben oft einen langen Namen und eine Abkürzung aus einem Buchstaben
  - Lange Optionen beginnen mit `"--"`
  - Kurze Optionen beginnen mit `"-"` und können gruppiert werden, z.B. `"-rf"`

Hilfe zu Kommandos

- Die meisten Kommandos geben eine kurze Übersicht der erlaubten Optionen und Argumente aus, wenn man sie falsch benutzt:

```
[ipk@vm ~] rm
usage: rm [-f | -i] [-dPRrvW] file ...
       unlink file
```

- Fast alle Kommandos geben mit der Option `"--help"` einen Hilfetext aufs Terminal aus
- Für genauere Informationen gibt es die man pages, die man mit dem Befehl `man` aufruft

```
[ipk@vm ~] man gcc
```

- In der man page bewegt man sich mit den Pfeiltasten und verlässt sie mit der Taste `"q"`

Verzeichnis wechseln I

- Verzeichnis wechseln mit `cd <name>` (change directory):

```
[ipk@vm ~] cd Documents
[ipk@vm Documents]
```

- Neues Verzeichnis wird im prompt angezeigt
- `cd` erzeugt UNIX-typisch nur bei Fehlern eine Ausgabe

```
[ipk@vm Documents] cd nonesuch
-bash: cd: nonesuch: No such file or directory
[ipk@vm Documents]
```

- `cd ..` kehrt ins übergeordnete Verzeichnis zurück

```
[ipk@vm Documents] cd ..
[ipk@vm ~]
```

Verzeichnis wechseln II

- `cd` unterstützt auch zusammengesetzte Pfade

```
[ipk@vm ~] cd Documents/c++
[ipk@vm c++]
```

- Man kann auch absolute Pfade verwenden, die mit / beginnen und unabhängig vom aktuellen Verzeichnis sind:

```
[ipk@vm c++] cd /etc/sysconfig
[ipk@vm sysconfig]
```

- `cd` ohne Argumente wechselt immer ins home directory

```
[ipk@vm sysconfig] cd
[ipk@vm ~]
```

Dateien auflisten

- `ls` (list) zeigt die Dateien im aktuellen Verzeichnis an

```
[ipk@vm c++] ls
helloworld    helloworld.cc
```

- Dateien, die mit einem Punkt beginnen, werden normalerweise nicht angezeigt. Dies kann man mit der Option `-a` ändern:

```
[ipk@vm c++] ls -a
.             ..             .versteckt
helloworld    helloworld.cc
```

- Auch `ls` akzeptiert einen Pfad als Argument

```
[ipk@vm c++] ls ~/Documents
c++
```

- `ls -l` (long) zeigt zusätzliche Informationen wie Besitzer, Zugriffsrechte, Dateigröße etc. an

Dateien kopieren und verschieben

- `cp` (copy) kopiert, `mv` (move) verschiebt Dateien

```
[ipk@vm c++] cp original copy
[ipk@vm c++] ls
copy     original
```

- Man kann auch mehrere Dateien gleichzeitig kopieren. In diesem Fall muss das Ziel ein Verzeichnis sein

```
[ipk@vm c++] mkdir subdir
[ipk@vm c++] mv original copy subdir
[ipk@vm c++] ls
subdir
[ipk@vm c++] ls subdir
copy     original
```

- Mit der Option `-r` (recursive) kopiert man Unterverzeichnisse samt Inhalt, beim Verschieben ist diese Option nicht erforderlich.

Dateien löschen

- `rm` (remove) löscht Dateien und Verzeichnisse

```
[ipk@vm c++] rm subdir/copy
[ipk@vm c++] ls subdir
original
```

- Mit der Option `-r` (recursive) löscht `rm` ein Verzeichnis mit allen Inhalten

```
[ipk@vm c++] rm -r subdir
[ipk@vm c++] ls
```

Warnung

`rm` fragt nicht nach, bevor die Dateien gelöscht werden, und in der Shell gibt es keinen Papierkorb! Gelöschte Dateien können nicht wiederhergestellt werden!

Dateien bearbeiten

- Es gibt Editoren, die im Textmodus im Terminal arbeiten (`vim`, `nano`, `emacs`,...), aber wir werden in dieser Vorlesung Editoren mit GUI verwenden
- Im Pool sind die Editoren `gedit` und `kate` installiert, in der VM `geany` und `qtcreator`, bei der WSL-Anleitung werden Sie `geany` installieren
- Sie können den Editor entweder wie gewohnt per Doppelklick auf eine Datei im Dateimanager starten oder direkt über die Shell

```
[ipk@vm c++] geany helloworld.cc &
[ipk@vm c++]
```

- Beim Starten von GUI-Programmen in der Shell ist es sinnvoll, ein `"&"` ans Ende des Befehls zu setzen. Ansonsten ist die Shell blockiert, bis Sie das gestartete Programm wieder beenden.

Dateien anzeigen

- `cat` zeigt den Inhalt von Dateien im Terminal an

```
[ipk@vm c++] ls
file1 file2
[ipk@vm c++] cat file2 file1
line in file2
line in file1
another line in file1
```

- `cat` kann bei langen Dateien unübersichtlich werden. `less` öffnet die Dateien in einem Viewer ähnlich zu dem für die man pages

```
[ipk@vm c++] less file1
```

  - Navigieren mit Pfeiltasten und `"q"` zum Beenden
  - Leertaste, um einen Bildschirm weiter zu springen
  - `"/"` + Suchwort + Enter, um zu suchen
  - `"n"`, um zum nächsten Vorkommen des Suchworts zu springen

Dateien durchsuchen

- `grep <pattern> <datei>...` durchsucht Dateien nach einem Pattern

```
[ipk@vm c++] grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
10:operator:x:11:0:operator:/root:/sbin/nologin
```

- Das Pattern kann ein einfaches Wort sein, man kann aber auch nach komplizierten Ausdrücken mit Hilfe sogenannter Regular Expressions suchen

I/O Streams

- UNIX-Programme kommunizieren mit dem System über sogenannte I/O (Input/Output) streams
- Streams sind eine Einbahnstraße — man kann aus ihnen entweder lesen oder Daten in sie schreiben
- Beim Start hat jedes Programm drei offene Streams

stdin	Die Standardeingabe liest User-Eingaben von der Konsole, ist verbunden mit file descriptor 0
stdout	An die Standardausgabe werden normale Ergebnisse des Programms ausgegeben, ist verbunden mit file descriptor 1
stderr	An die Standardfehlerausgabe werden diagnostische Meldungen wie Fehler ausgegeben, ist verbunden mit file descriptor 2

Umleiten von I/O Streams I

- Normalerweise sind alle Standardstreams mit dem Terminal verbunden
- Manchmal kann es sinnvoll sein, diese Streams in Dateien umzuleiten
- stdout wird mit "> datei" in einer Datei gespeichert

```
[ipk@vm ~] ls > files
[ipk@vm ~] cat files
file1
file2
files
```

Die Ausgabedatei wird angelegt, bevor der Befehl ausgeführt wird, daher taucht sie selbst auf

- Fehlermeldungen werden weiterhin im Terminal angezeigt

```
[ipk@vm ~] ls missingdir > files
ls: missingdir: No such file or directory
[ipk@vm ~] cat files
[ipk@vm ~]
```

Umleiten von I/O Streams II

- stdin wird mit "< datei" aus einer Datei gelesen

```
[ipk@vm ~] cat # ohne Argument gibt cat stdin nach stdout aus
Eingabe am Terminal^D # (CTRL+D) beendet die Eingabe
Eingabe am Terminal
[ipk@vm ~] cat < files
 file1
 file2
 files
```

- stderr wird mit "2> datei" in einer Datei gespeichert

```
[ipk@vm ~] ls missingdir 2> error
[ipk@vm ~] cat error
ls: missingdir: No such file or directory
```

Kompilieren von C++-Programmen

- C++-Programme müssen vor dem Ausführen kompiliert (= in Maschinensprache übersetzt) werden
- Der Standard-C++-Compiler unter Linux heißt g++
- Oft wird auch gerne stattdessen `clang++` aus dem LLVM-Projekt verwendet, da dieser verständlichere Fehlermeldungen generiert
- Sie sollten immer die Option `-Wall` verwenden, damit der Compiler Sie bei Problemen warnt, die zwar legales C++ sind, aber wahrscheinlich von Ihnen so nicht gewollt sind
- Beim Ausführen des erstellten Programms muss `./` vorangestellt werden

```
[ipk@vm ~] g++ -Wall -o test test.cc
[ipk@vm ~] ./test
```

Fragen bis jetzt?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?
- Dropbox mit mehreren Leuten benutzt?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?
- Dropbox mit mehreren Leuten benutzt?
- Dabei Dateien verloren, weil zwei Leute gleichzeitig gespeichert haben?

Was ist ein Version Control System

Ein Version Control System

- Speichert Schnapschüsse (Commits) eines Verzeichnisses (mit Unterverzeichnissen)
- Speichert für jeden Commit eine Beschreibung, was sich geändert hat und wer es geändert hat
- Ermöglicht es, für Textdateien genau anzuzeigen, wie sich zwei Versionen einer Datei unterscheiden
- Speichert (oft) eine Kopie der Daten auf einem Server
  - Datensicherung
  - Datenaustausch mit anderen Computern, Entwicklern
- Erstellt Commit nur auf explizite Anfrage
  - Keine kaputten Versionen committen (kompiliert nicht etc.)
  - Commit-Beschreibung muss eingegeben werden
- Unterstützt dabei, gleichzeitige Änderungen von mehreren Entwicklern zusammenzuführen (merge)

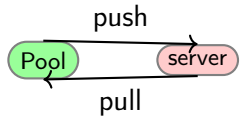
git — Distributed Version Control System

- Kommandozeilenprogramm zum Verwalten von (ursprünglich) text-basierten Dateien
- Entwickelt 2005 von Linus Torvalds für die Quellen des Linux-Kernels
- Extrem schnell
- Verwaltet einige der grössten Codebasen weltweit:
  - Linux-Kernel ($>$ 25 Mio. Zeilen, $>$ 10.000 Commits / Version)
  - Windows (300 GB, 3,5 Mio. Dateien)
- Kostenloses Repository-Hosting, z.B. GitHub, Bitbucket, GitLab
- Unterstützung für Binärdaten (Bilder, Musik, Gitter, Matrizen etc.) mit git-lfs
- Inzwischen de-facto Industriestandard

Use Cases

Warum machen wir das ganze hier?

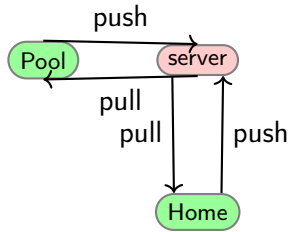
- Wichtig, um später an realen Softwareprojekten mitarbeiten zu können
- Braucht eine gewisse Eingewöhnungsphase
- Konsequente Verwendung von Versionskontrolle hilft beim Strukturieren der Programmierarbeit
- Einfache Möglichkeit, zusammen an Übungen zu arbeiten und diese abzugeben.
- Dokumentation von Versuchen und Diskussion mit den Tutoren



Use Cases

Warum machen wir das ganze hier?

- Wichtig, um später an realen Softwareprojekten mitarbeiten zu können
- Braucht eine gewisse Eingewöhnungsphase
- Konsequente Verwendung von Versionskontrolle hilft beim Strukturieren der Programmierarbeit
- Einfache Möglichkeit, zusammen an Übungen zu arbeiten und diese abzugeben.
- Dokumentation von Versuchen und Diskussion mit den Tutoren



Glossar

Repository Datenbank mit allen Informationen über Dateiversionen in einem Projekt, liegt im versteckten Verzeichnis `.git` im obersten Projektverzeichnis.

Commit Globaler Schnappschuss aller Projektdateien mit einer Beschreibung der Änderungen zur vorherigen Version.

Branch Eine Abfolge von Commits, die einen Entwicklungszweig abbilden. Ein Repository kann mehrere Branches enthalten. Der Standard-Branch heißt `master`.

Tag Ein dauerhafter Name für einen Commit, z.B. für ein Release.

Globale Konfiguration

Zuerst sollten wir zwei Dinge einrichten:

- git möchte wissen, wer wir sind:

```
git config --global user.name "Lorenz Braun"
git config --global user.email "lorenz.braun@ziti.uni-heidelberg.de"
```

- Für Git-Status in der Kommandozeile folgende Zeilen in `~/.bash_profile` einfügen:

```
export PS1='[\u@\h \W$(__git_ps1 " (%s)")]\$ '
export GIT_PS1_SHOWDIRTYSTATE=1
```

Getting Started

Zum Anlegen des Repositories ins oberste Projektverzeichnis wechseln und dann:

```
[git-tutorial]$ git init
Initialized empty Git repository in /home/oklein/git-tutorial/.git/
[git-tutorial (master #)]$
```

Damit existiert das Repository, aber es gibt noch keinen Commit:

```
[git-tutorial (master #)]$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        input.cc
        input.hh

nothing added to commit but untracked files present (use "git add" to track)
[git-tutorial (master #)]$
```

- Kopieren Sie beliebige Dateien der ersten Wochen in den Ordner, wenn Sie die Befehle ausprobieren möchten

Änderungen hinzufügen

git speichert nur Änderungen, von denen wir ihm explizit erzählen:

```
[git-tutorial (master #)]$ git add input.cc
[git-tutorial (master +)]$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   input.cc

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        input.hh

[git-tutorial (master +)]$
```

Änderungen committen

Jetzt können wir auch noch input.hh hinzufügen und einen Commit erzeugen:

```
[git-tutorial (master +)]$ git add input.hh
[git-tutorial (master +)]$ git commit
[master (root-commit) 1bb9ef8] Added input files
 2 files changed, 25 insertions(+)
 create mode 100644 input.cc
 create mode 100644 input.hh
[git-tutorial (master)]$ git status
On branch master
nothing to commit, working tree clean
[oklein@ipk git-tutorial (master)]$ git log
commit 1bb9ef87e4c235cc72e07009fc48cefb38df6154 (HEAD -> master)
Author: Ole Klein <ole.klein@iwr.uni-heidelberg.de>
Date:   Fri Dec 1 10:55:17 2017 +0100

    Added input files
  [oklein@ipk git-tutorial (master)]$
```

Commits

- Commits enthalten
  - einen Snapshot aller Dateien,
  - den Erstellungszeitpunkt,
  - Namen und Inhalt vom Autor der Änderungen und von der Person, die den Commit erstellt hat,
  - eine Beschreibung der Änderungen (Changelog),
  - Eine Liste mit Verweisen auf die Eltern-Commits.
- Commits werden durch einen Hash (eine Prüfsumme) ihres Inhalts identifiziert, z.B. `1bb9ef87e4c235cc72e07009fc48cefb38df6154`.
- Commit-Hashes können abgekürzt werden, solange sie eindeutig sind.
- Commits können nicht verändert werden:
  anderer Inhalt $\Rightarrow$ anderer Hash.

Weiterarbeiten

Wir verändern die Datei `input.hh` und speichern die veränderte Datei:

```
[git-tutorial (master *)]$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   input.hh

no changes added to commit (use "git add" and/or "git commit -a")
[git-tutorial (master *)]$
```

Zur Erinnerung: Änderungen müssen wir git immer mitteilen!

```
[git-tutorial (master *)]$ git add input.hh
[git-tutorial (master +)]$ git commit
[master bab868d] Added comment
 1 file changed, 2 insertions(+)
[git-tutorial (master)]$
```

Unterschiede anzeigen

Anzeigen, was der aktuelle Commit verändert hat:

```
[git-tutorial (master)]$ git show master
commit bab868dd7e345f1b660157a8bd4519cad175733d (HEAD -> master)
Author: Ole Klein <ole.klein@iwr.uni-heidelberg.de>
Date:   Fri Dec 1 11:06:27 2017 +0100

    Added comment

diff --git a/input.hh b/input.hh
index 3b74a4a..3bef25c 100644
--- a/input.hh
+++ b/input.hh
@@ -4,6 +4,8 @@
 #include <string>
 #include <istream>

+// Reads from input until EOF and returns the
+// result as a string.
 std::string read_stream(std::istream& input);

 #endif // INPUT_HH
[git-tutorial (master)]$
```

Tipp

Diffs mit grafischen Hilfsprogrammen (z.B. meld) oder einfach auf dem Server anschauen!

Branches

- Ein Branch ist ein Entwicklungszweig innerhalb eines Repositories.
- Repositories können beliebig viele Branches enthalten.
- `git status` sagt einem, auf welchem Branch man sich befindet.
- Ein Branch zeigt auf einen Commit.
- Wenn man einen neuen Commit erstellt, speichert er den aktuellen Commit als Vater und der Branch zeigt danach auf den neuen Commit.

Branches: Befehle

- Branch `playground` erstellen:

```
git branch playground
```

- Branches auflisten:

```
[git-tutorial (master)]$ git branch
* master
  playground
[git-tutorial (master)]$
```

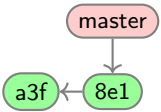
- Branch wechseln:

```
[git-tutorial (master)]$ git checkout playground
Switched to branch 'playground'
[git-tutorial (playground)]$
```

- Änderungen von anderem Branch importieren (merge):

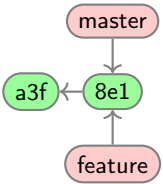
```
git merge other-branch
```

Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

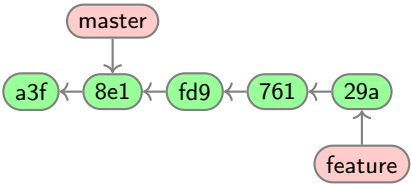
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- Branch anlegen

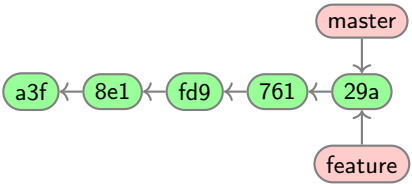
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- Branch anlegen
- Auf Branch arbeiten

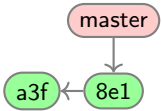
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

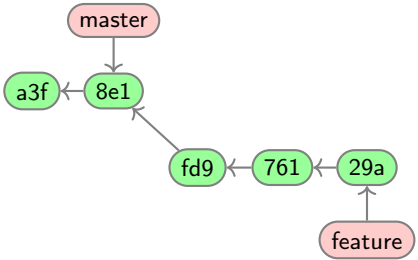
- Branch anlegen
- Auf Branch arbeiten
- Keine neuen Commits in master $\Rightarrow$ fast-forward

Merging II



Der realistische Fall: Gleichzeitige Änderungen

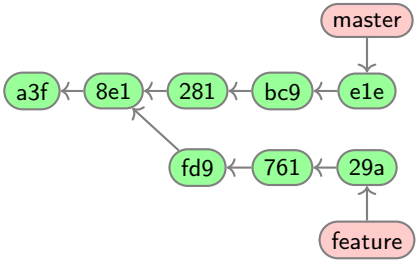
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten

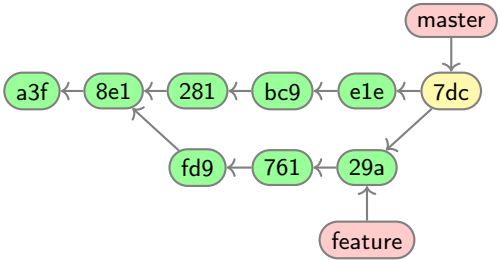
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert

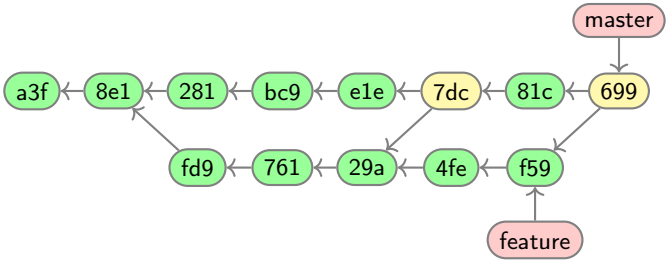
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert
- Änderungen nach master mergen
  - Erzeugt Merge-Commit
  - Bei Konflikten muss manuell nachgeholfen werden

Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert
- Änderungen nach master mergen
  - Erzeugt Merge-Commit
  - Bei Konflikten muss manuell nachgeholfen werden

Git — Wichtige Befehle

init	Leeres Repository anlegen
clone	Bestehendes Repository klonen
add	Änderungen für Commit registrieren
commit	Commit erstellen
log	Verlauf ansehen
diff	Änderungen ansehen
branch	Branches anlegen, auflisten, löschen
checkout	Branch wechseln
merge	Branches zusammenführen
pull	Neue Änderungen herunterladen
push	Neue Änderungen hochladen

```
git help <Befehl> für Hilfe!
```

Credential Caching in git

Es gibt zwei Möglichkeiten, das ständige Eingeben von Passwörtern in git zu umgehen:

- Erstellen eines SSH-Schlüssels, Hochladen des Schlüssels auf den Server (über das Webinterface) und aktivieren des SSH agents (siehe Tutorials im Internet, SSH Agent hängt von der Linux-Distribution ab)
- Aktivieren des Credential Cache:

```
git config --global credential.helper cache
```

  Durch dieses Kommando wird git auf dem lokalen Rechner so konfiguriert, dass Benutzername und Passwort für 15 min gespeichert werden

Empfehlung

- Spielerische Einführung https://ohmygit.org/
- Anmeldung auf https://gitlab.com
- Projekt *IPK2023* im lokalen Namensraum anlegen
- 2FA **NICHT** aktivieren (ansonsten: Token als Passwort verwenden)
- Jede Übung in
  - separaten Branch
  - oder separaten Ordner

Fragen bis jetzt?