

Programmierkurs

Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

November 9, 2018

C++-Grundlagen

Wiederholung

Namespaces

Versionskontrolle mit git

Übersicht

Einführung

Branches

Merging

Mehrere Repositories

Cheat Sheet

Wiederholung: Mehrere Funktionen

- ▶ Funktionen können **nicht** ineinander verschachtelt werden
- ▶ Damit eine Funktion von einer anderen Funktion aus aufgerufen werden kann, muss sie vor der aufrufenden Funktion stehen

```
int error(x)
{
    return square(x); // compile error, place after square
}

int square(int x)
{
    return x * x;
}

int cube(int x)
{
    return square(x) * x;
}
```

Fragen / Unklarheiten in C++ bis jetzt?

Namespaces und Zusammenarbeit

- ▶ Alice schreibt eine Bibliothek mit einer Funktion `greeting()`, die Sie wie folgt verwenden können:

```
#include <alice.hh>

int main(int argc, char** argv)
{
    greeting(); // aus alice.hh
}
```

Namespaces und Zusammenarbeit

- ▶ Alice schreibt eine Bibliothek mit einer Funktion `greeting()`, die Sie wie folgt verwenden können:

```
#include <alice.hh>

int main(int argc, char** argv)
{
    greeting(); // aus alice.hh
}
```

- ▶ Bob schreibt auch eine Bibliothek mit einer Funktion `greeting()`:

```
#include <bob.hh>

int main(int argc, char** argv)
{
    greeting(); // aus bob.hh
}
```

Kombinieren mehrerer Bibliotheken

Wenn man mehrere verschiedene Bibliotheken verwendet, die beide Symbole mit gleichem Namen definieren, kommt es zu

Namenskollisionen:

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
    // Compile-Fehler!
    // Welches greeting() soll aufgerufen werden?
    greeting();
}
```

Kombinieren mehrerer Bibliotheken

Wenn man mehrere verschiedene Bibliotheken verwendet, die beide Symbole mit gleichem Namen definieren, kommt es zu **Namenskollisionen**:

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
    // Compile-Fehler!
    // Welches greeting() soll aufgerufen werden?
    greeting();
}
```

Problematisch, man kann ja nicht voraussehen, welche Funktionsnamen andere Entwickler sich ausdenken!

Namespaces

- ▶ Namespaces ermöglichen es Bibliotheken, eigene Namensräume zu definieren.
- ▶ Um auf ein Symbol `func()` aus dem namespace `mylib` zuzugreifen, stellt man den namespace gefolgt von `::` vor den Namen des Symbols:

```
mylib::func()
```

- ▶ Namespaces können auch ineinander verschachtelt werden:

```
mylib::mysublib::func()
```

- ▶ C++ liefert eine [Standardbibliothek](#) als Teil des Compilers mit. Alle Funktionalität der Standardbibliothek befindet sich in namespace `std`.

Namespaces: Alice und Bob

Mit namespaces läßt sich das Problem des uneindeutigen Funktionsnamens auflösen:

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
    alice::greeting();
    bob::greeting();
}
```

Namespaces selber anlegen

- ▶ Namespaces sind ein spezielles Scope ausserhalb einer Funktion, das durch das Keyword **namespace** eingeleitet wird:

```
namespace alice {  
  
    void greeting()  
    {  
        std::cout << "Hello Alice!" << std::endl;  
    }  
  
} // end namespace alice
```

- ▶ Da ein **namespace**-Scope oft sehr lang ist, sollte man hinter die schliessende Klammer einen kurzen Kommentar schreiben, was hier eigentlich geschlossen wird.
- ▶ Innerhalb des **namespace**-Scope muss man für andere Funktionen im gleichen namespace den Namen des namespace nicht davorschreiben.

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 Bilder 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, Übungen im Kurs zu sichern und zwischen Pool-Rechnern und eigenem Computer zu synchronisieren

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, Übungen im Kurs zu sichern und zwischen Pool-Rechnern und eigenem Computer zu synchronisieren

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, Übungen im Kurs zu sichern und zwischen Pool-Rechnern und eigenem Computer zu synchronisieren

Struktur

- ▶ Mischung aus Erklärungen und Demos
- ▶ Wird auf dem nächsten Übungsblatt vertieft

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 "Steffen Müthing"  
git config --global user.email  
↪ "steffen.muething@iwr.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 I

Zum Anlegen des Repositories ins oberste Projektverzeichnis wechseln und dann:

```
[git-tutorial]$ git init
Initialized empty Git repository in /Users/smuething/tmp/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 #)]$
```

Änderungen hinzufügen

git speichert nur Änderungen, von denen wir im 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
[smuething@muh109-191 git-tutorial (master)]$ git log
commit 1bb9ef87e4c235cc72e07009fc48cefb38df6154 (HEAD -> master)
Author: Steffen Müthing <muething@dune-project.org>
Date:   Fri Dec 1 10:55:17 2017 +0100

    Added input files
[smuething@muh109-191 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: Steffen Müthing <muething@dune-project.org>
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)]$
```

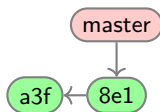
- ▶ Branch wechseln:

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

- ▶ Änderungen von anderem Branch importieren (merge):

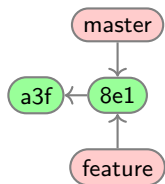
```
git merge other-branch
```

Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

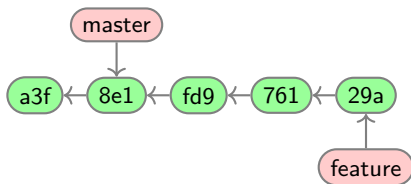
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- ▶ Branch anlegen

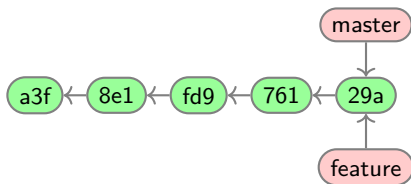
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- ▶ Branch anlegen
- ▶ Auf Branch arbeiten

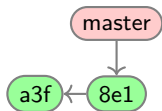
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

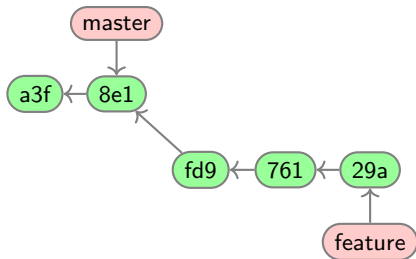
- ▶ Branch anlegen
- ▶ Auf Branch arbeiten
- ▶ Keine neuen Commits in master \Rightarrow fast-forward

Merging II



Der realistische Fall: Gleichzeitige Änderungen

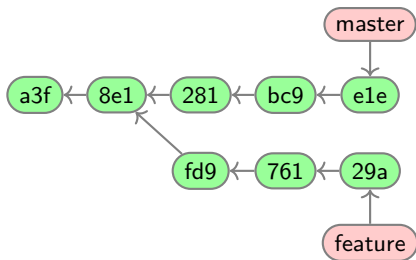
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- ▶ Branch erstellen und darauf arbeiten

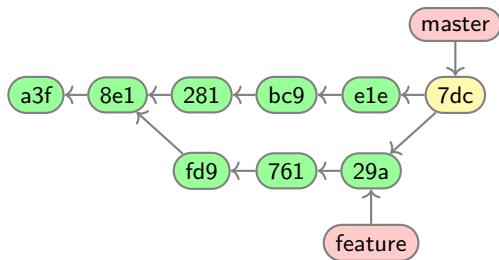
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- ▶ Branch erstellen und darauf arbeiten
- ▶ Master wird verändert

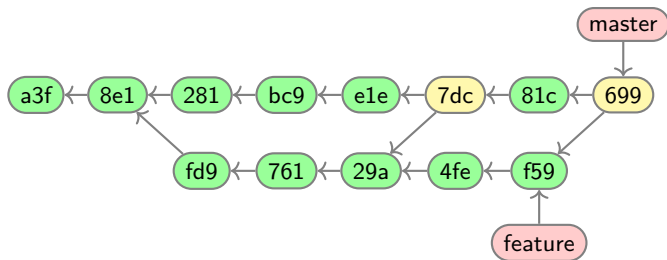
Merging II



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

Mehrere Repositories I

- ▶ git kann Änderungen zwischen Repositories synchronisieren.
- ▶ Zusätzliche Repositories heißen `remote`.
- ▶ Branches aus einem Remote-Repository bekommen den Namen des Repositories vorangestellt.
- ▶ Man kann ein Remote-Repository `klonen`, um den Inhalt zu bekommen:

```
[folder]$ git clone https://gitlab.dune-project.org/core/dune-common.git
Cloning into 'dune-common'...
remote: Counting objects: 54485, done.
remote: Compressing objects: 100% (13788/13788), done.
remote: Total 54485 (delta 40840), reused 54059 (delta 40531)
Receiving objects: 100% (54485/54485), 12.83 MiB | 19.15 MiB/s, done.
Resolving deltas: 100% (40840/40840), done.
[folder]$
```

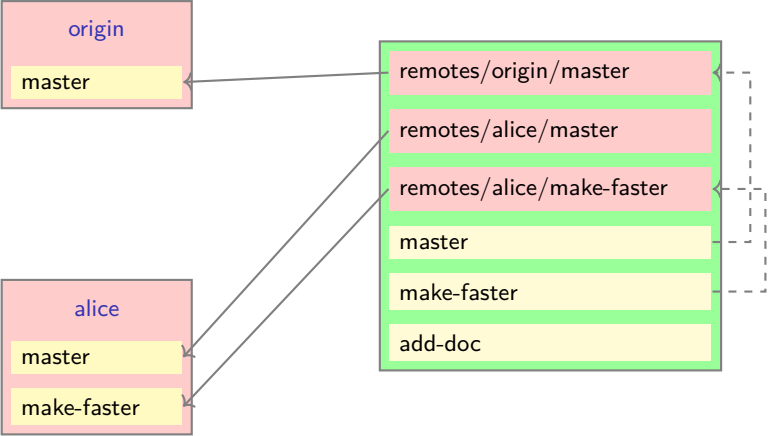
Mehrere Repositories II

- ▶ Man kann Branches von Remote-Repositories nicht direkt auschecken.
- ▶ Git legt beim ersten checkout einen tracking branch an:

```
[dune-common (master)]$ git checkout releases/2.6  
Branch 'releases/2.6' set up to track remote branch 'releases/2.6' from 'origin'.  
Switched to a new branch 'releases/2.6'  
[dune-common (releases/2.6)]$
```

- ▶ Neue Änderungen können mit `git pull` heruntergeladen und gemergt werden.
- ▶ Eigene Änderungen können mit `git push` hochgeladen werden.
 - ▶ Wenn jemand anderes vorher Änderungen hochgeladen hat: Fehler!
 - ▶ Lösung: Änderungen erst mit `git pull` herunterladen und integrieren, dann nochmal pushen.

Mehrere Repositories - Schema



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!