

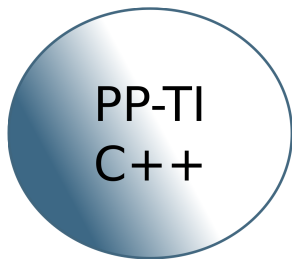


IMS  
Institut für Mikroelektronische Systeme  
Leibniz Universität Hannover



Leibniz  
Universität  
Hannover

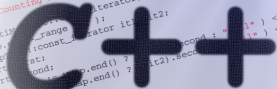
# Programmierpraktikum Technische Informatik (C++)



21.4.2022



# Einleitung



The C++ logo is prominently displayed in the center of the slide, overlaid on a circular, semi-transparent image of C++ source code. The code includes various map operations such as `insertPair`, `begin`, `end`, `count`, and `second`.

## Ansprechpartner

- Dr.-Ing. Markus Olbrich  
Tel.: 762-19661  
[markus.olbrich@ims.uni-hannover.de](mailto:markus.olbrich@ims.uni-hannover.de)
- Beratung per Forum und E-Mail

# Was ist C++?

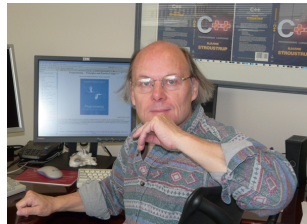
- Ursprünglich Erweiterung von C um objektorientierte Programmierung
- Ermöglicht gleichzeitig
  - **Systemprogrammierung:**  
Effiziente maschinennahe Programme
  - **Anwendungsentwicklung:**  
Objektorientierte Programmierung auf hoher Abstraktionsebene
- Rückwärtskompatibel zu C90
- Mittlerweile eigenständige Sprache
- Gegenüber C umfangreichere Bibliothek: C++-Standardbibliothek
- Aktuelle Version: C++20

# Entwicklung der Programmiersprache C++

1979: Bjarne Stroustrup

(\* 30. Dezember 1950 in Dänemark):

Beginn Entwicklung von C++ als Erweiterung von C  
(„C with Classes“)



- 1998: C++98 ISO-standardisiert, mit Standard Template Library (STL)
- 2011: C++11 Erste größere Erweiterung der Sprache
- 2014: C++14 Bugfixes und kleinere Erweiterungen
- 2017: C++17 Erweiterungen für C++14
- 2020: C++20 Erweiterungen für C++17 (u.a. Concepts)

## Beispiel in C, Java und C++

### ■ C

```
#include <stdio.h>
int main(void) {
    printf("Hallo Leute!\n");
}
```

### ■ Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hallo Leute!");
    }
}
```

### ■ C++

```
#include <iostream>
int main() {
    std::cout << "Hallo Leute!" << std::endl;
}
```

## Warum ist C++ so relevant?

- Laufzeit-/Energieeffizienz dauerhaft wichtig
  - Mobile Geräte
  - Serverfarmen
- C für komplexe Software zu unübersichtlich
- Java bietet oft nicht die nötige Kontrolle zur effizienten Implementierung
- C++ wird viel verwendet
  - Industrie
  - In der Uni viele Abschlussarbeiten

## Warum C++?

The going Word at **Facebook** is that **reasonably written C++ code just runs fast**, which underscores the enormous effort spent at optimizing PHP and Java code. Paradoxically C++ code is more difficult to write then in other languages, but **efficient code is a lot easier**

*Andrei Alexandrescu, Research Engineer at Facebook*

**Google** has one of the largest monolithic C++ codebases in the world. We have thousands of engineers working on **millions of lines of C++ code** every day.

*Google Engineering Tools Blog*



# Einige Vorteile C++ gegenüber C und Java

- Vorteile gegenüber C
  - OOP: Klassen, Vererbung etc.
  - Strikteres Typsystem
  - Generische Programmierung: Templates
  - Ausnahmebehandlung (Exceptions)
  - Umfangreichere Standardbibliothek
- Vorteile gegenüber Java
  - Keine JVM nötig
  - Höhere Laufzeit-Effizienz
  - Maschinennahe Programmierung möglich

# Literatur

- The C++ Programming Language (Bjarne Stroustrup)
- Effective Modern C++ (Scott Meyers)
- C++ Primer (Fifth Edition)
- The C++ Standard Library
- ISO/IEC 14882:2020
  - Offizieller C++20 Standard
- Für weitere Bücher:  
<http://stackoverflow.com/questions/388242>

# Onlinematerialien

- Referenz:  
<http://www.en.cppreference.com>
- C++ FAQ:  
<http://isocpp.org/faq>
- Frage- und Antwortplattform:  
<http://stackoverflow.com>
- Guidelines:  
<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- C++ Standard:  
<https://wg21.link/std20>
  - Inhaltlich identisch zum offiziellen C++20-Standard, nur redaktionelle Änderungen

## Aufgaben und Testate

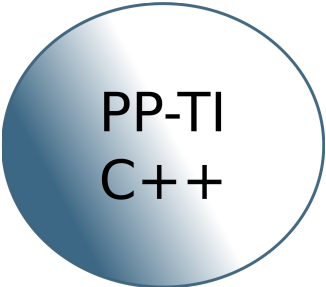
- Programmieraufgaben werden wöchentlich gestellt (Aufgabenzettel am Donnerstag)
- Jeder Teilnehmer bekommt eigenes Repository
- Lösung muss Dienstag bis 9 Uhr im Repository der Uni sein
- Insgesamt 10 Testate
- Warnings beim Compilieren führen zu Punktabzug

## Wann besteht man das Programmierpraktikum?

- Mindestens die Hälfte der Punkte in Testaten 1-5 und Hälfte in Testaten 6-10 und
- Zusätzliches Abschlusstest
  - Mehrere Termine, voraussichtlich beginnend am 21.7.2022
  - Programmieraufgabe in begrenzter Zeit lösen
  - Voraussichtlich im Institut



# Einführung in C++



PP-TI  
C++

# Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- In der Regel das erste Programm, das ein Programmierer schreibt
- Gibt den Text "Hello, world!" auf der Konsole aus
- Enthält trotz seiner Kürze bereits verschiedenste Sprachelemente

# Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Bindet die Ein- und Ausgabebibliothek von C++ ein
- `#include` bindet sogenannte Header-Dateien ein



## Header-Dateien

- Werden verwendet, um Funktionen und Typen für andere Kompilierungseinheiten sichtbar zu machen
- Konvention für Dateiendungen: \*.h, \*.hpp, \*.hh
- Ausnahme C++-Standardbibliothek: Header haben keine Dateiendung.
- Header der C-Standardbibliothek sind mit angepassten Namen in C++ verfügbar: `math.h` heißt beispielsweise `cmath`.
- `#include <...>` bindet Dateien aus Suchpfaden des Compilers ein.
- `#include "..."` sucht zunächst relativ zur einbindenden Datei, danach wie `#include <...>`.
- `#include` ist sog. Präprozessor-Direktive.

# Präprozessor

- Präprozessor liest Quelltext, verändert ihn und gibt den veränderten Text an den Compiler weiter.
- Präprozessor führt lediglich textuelle Ersetzungen durch.
  - Kann schwer zu debuggende Probleme mit dafür nicht ausgelegtem Code verursachen
- Präprozessor-Direktiven müssen in eigenen Zeilen stehen.
- `#include` fügt den Inhalt der angegebenen Header-Datei ein.
- Mehrfache Inklusion von Headern kann zu Problemen führen, falls Objekte in der Header-Datei definiert werden.

Präprozessor nur für die hier vorgestellten Anwendungsfälle verwenden!

# Präprozessor: Macros

- `#define Name Inhalt` definiert Macro.  
Jedes Auftreten von `Name` wird durch `Inhalt` ersetzt.
- Gängige Konvention: Macros komplett in Großbuchstaben
- `#if Expression ... #endif` entfernt Codezeilen zwischen `#if` und `#endif`, falls `Expression` nicht erfüllt ist.
- `#ifdef Name` ist Kurzform für `#if defined(Name)` und überprüft, ob `Name` ein definiertes Macro ist.
- Analog prüft `#ifndef Name`, ob `Name` nicht definiert ist
- Schutz vor mehrfacher Inkludierung (Include Guards):

```
#ifndef CUSTOM_STRING
#define CUSTOM_STRING
    // Content
#endif
```

## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- `/* Comment */, // Comment till end of line`
- Nicht schachtelbar!
- Dienen der Dokumentation
- Auskommentieren besser mit  
`#if false`  
`#endif`  
 Dies ist auch schachtelbar.

## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Definiert eine Funktion mit dem Namen `main`
- `main` ist die Funktion, die bei Ausführung des Programmes ausgeführt wird (wie in C)
- `main`-Funktion gibt einen Wert vom Typ `int` zurück und hat keine Argumente
  - Unterschied zwischen C++ und C: Leere Klammer gibt in C++ eine leere Parameterliste an, Entsprechung in C: `int main(void)`

# Primitive Datentypen

Name	Wertebereich	Beispiele
<code>bool</code>	<code>true, false</code>	<code>true</code>
<code>char</code>	*	<code>'a', '\n', 15</code>
<code>signed char</code>	-128 bis 127	-10
<code>unsigned char</code>	0 bis 255	15
<code>short</code>	-32768 bis 32767	17, -10
<code>unsigned short</code>	0 bis 65535	15
<code>int</code>	-32768 bis 32767	17, -10, 0x00ff
<code>unsigned int</code>	0 bis 65535	15u, 0xFF00u
<code>long</code>	$-2^{31}$ bis $2^{31} - 1$	17l, -10l
<code>unsigned long</code>	0 bis $2^{32} - 1$	15ul
<code>long long</code>	$-2^{63}$ bis $2^{63} - 1$	17ll, -10ll, 15ull
<code>unsigned long long</code>	0 bis $2^{64} - 1$	17ll, -10ll, 15ull
<code>float</code>	7 Dezimalstellen	3.14f, 1.35ef, -0.2E-7f
<code>double</code>	15 Dezimalstellen	3.14, 1.35e6, -0.2E-7

- Wertebereiche sind Mindestwerte für den Typ.
- `char` entspricht entweder `signed char` oder `unsigned char`.
- Konstanten, die mit 0x anfangen sind hexadezimal, solche die mit 0 anfangen sind oktal.
- Die Genauigkeit von `int` und `unsigned int` ist auf 32bit-Systemen meist  $-2^{31}$  bis  $2^{31} - 1$  bzw. 0 bis  $2^{32} - 1$ .
- Die Genauigkeit von `long` und `unsigned long` ist auf 64bit-Systemen meist  $-2^{63}$  bis  $2^{63} - 1$  bzw. 0 bis  $2^{64} - 1$ .

# Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Rumpf der Funktion steht in geschweiften Klammern
- Rumpf enthält Anweisungen ("Statements").

## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Anweisung ("Statement") ist ein Ausdruck ("Expression") gefolgt von Semikolon
- Statements können durch Verwendung von geschweiften Klammern zu Blöcken ("Compound Statements") zusammengefasst werden:

```
{
    statement1;
    statement2;
}
```

- Compound Statements sind überall erlaubt, wo Statements erlaubt sind.



## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Klassen und Funktionen können in C++ durch sogenannte Namespaces gruppiert werden. Diese ähneln den Packages aus Java.
- Auf den Inhalt eines Namespaces wird mit :: zugegriffen.
- Elemente der C++-Standardbibliothek liegen im Namespace std.
- Aus der C-Standardbibliothek geerbte Elemente sind in den std-Namespace integriert.

## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- `std::cout` ist der Standardausgabestream von C++.
- Mit dem `<<` Operator können beliebige Daten in den Stream geschrieben ("geschoben") werden. Diese können beliebige Typen haben, beispielsweise ist auch `std::cout<<4;` möglich.
- `std::endl` signalisiert einen Zeilenumbruch und flusht den Stream, wodurch dessen Inhalt in die Ausgabe geschrieben wird.
- Weitere Operatoren (z.B. `+`, `*`, `/`, ...) wie in C und Java. Siehe z.B. <http://en.cppreference.com>.

## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Zeichenketten (String Literale) stehen in Anführungszeichen.

## Ein einfaches Beispiel

```
#include <iostream>
/*A simple Hello World! program*/
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

- Ergebnisse einer Funktion werden mit `return` zurückgegeben.
- Beendet die Funktion und führt mit der Ausführung der aufrufenden Funktion fort.
- Im Fall von `main` wird das Programm beendet.
- Rückgabewert von 0 aus `main` signalisiert korrekte Ausführung des Programms, andere Rückgabewerte signalisieren Fehler.
  - Für `main` wird bei fehlendem `return` implizit 0 zurückgegeben.
  - Für alle anderen Funktionen ist dies ein Programmierfehler.

## Ein etwas komplexeres Beispiel

```
void printGreeting(std::string name, int border);  
int main()  
{  
    std::cout << "your name: " << std::flush;  
    std::string name;  
    std::cin >> name;  
    std::cout << std::endl;  
    printGreeting(name, 3);  
    return 0;  
}  
void printGreeting(std::string name, int)  
{  
    std::cout << name << std::endl;  
}
```

- Funktionen müssen vor ihrer Verwendung **deklariert** werden
- Ein alleinstehender Funktionskopf bildet eine **Forward-Deklaration**
- Eine Funktion kann in auf die Forward-Deklaration folgendem Quellcode verwendet werden, auch wenn sie noch nicht definiert ist

# Deklaration und Definition

- **Deklaration** teilt dem Compiler die Existenz von Funktionen/Typen/Variablen mit
  - Ermöglicht Nutzung des deklarierten Objektes
  - Nutzung nicht deklarerter Funktionen/Typen/Variablen führt zu Compilerfehlern
- **Definition** liefert die Implementation
  - Funktionsrumpf für Funktionen
  - Enthaltene Daten und Methoden für Typen
  - Speicherstelle für Variablen
  - Nutzung nicht definierter Funktionen/Variablen führt zu Linkerfehlern

## Ein etwas komplexeres Beispiel

```
void printGreeting(std::string name, int border);
int main()
{
    std::cout << "your name: " << std::flush;
    std::string name;
    std::cin >> name;
    std::cout << std::endl;
    printGreeting(name, 3);
    return 0;
}
void printGreeting(std::string name, int)
{
    std::cout << name << std::endl;
}
```

- Funktionen mit Rückgabetyt `void` geben nichts zurück. Sie müssen keine Return-Statements enthalten.
- Zur vorzeitigen Beendung ohne Rückgabe kann `return`; verwendet werden.
- Parameter, die in der Funktion nicht verwendet werden, müssen keinen Namen haben.

## Ein etwas komplexeres Beispiel

```
void printGreeting(std::string name, int border);  
int main()  
{  
    std::cout << "your name: " << std::flush;  
    std::string name;  
    std::cin >> name;  
    std::cout << std::endl;  
    printGreeting(name, 3);  
    return 0;  
}  
void printGreeting(std::string name, int)  
{  
    std::cout << name << std::endl;  
}
```

- `std::flush` bewirkt wie `std::endl` ein Rückschreiben der Ausgabe, allerdings ohne einen Zeilenumbruch einzufügen



## Ein etwas komplexeres Beispiel

```
void printGreeting(std::string name, int border);  
int main()  
{  
    std::cout << "your name: " << std::flush;  
    std::string name;  
    std::cin >> name;  
    std::cout << std::endl;  
    printGreeting(name, 3);  
    return 0;  
}  
void printGreeting(std::string name, int)  
{  
    std::cout << name << std::endl;  
}
```

- Variablen werden mit *type variable* definiert, beispielsweise `int foo;`.
- `std::string` ist der Standardtyp zur Darstellung von Zeichenketten.
- Komplexe Datentypen wie `std::string` werden automatisch initialisiert, primitive Typen nicht.

## Ein etwas komplexeres Beispiel

```
void printGreeting(std::string name, int border);
int main()
{
    std::cout << "your name: " << std::flush;
    std::string name;
    std::cin >> name;
    std::cout << std::endl;
    printGreeting(name, 3);
    return 0;
}
void printGreeting(std::string name, int)
{
    std::cout << name << std::endl;
}
```

- Analog zu `std::cout` kapselt `std::cin` den Standardeingabestream der Anwendung.
- Mit `>>` werden Daten aus dem Stream in Variablen geschoben.

## Ein etwas komplexeres Beispiel

```
void printGreeting(std::string name, int border);
int main()
{
    std::cout << "your name: " << std::flush;
    std::string name;
    std::cin >> name;
    std::cout << std::endl;
    printGreeting(name, 3);
    return 0;
}
void printGreeting(std::string name, int)
{
    std::cout << name << std::endl;
}
```

- `int foo; std::cin>>foo;` würde bei Eingabe 4 die Variable `foo` auf 4 setzen.
- Führende Leerzeichen werden ignoriert.
- Das Einlesen von Strings stoppt bei Leerzeichen, Tabulatoren und Zeilenumbrüchen.

# Großes Beispiel: Erweiterte Ausgabe der Anrede

```
void printGreeting(std::string name, int border)
{
    int lineWidth = name.size() + 2 * border;
    std::string outerLine(lineWidth + 2, '*');
    std::cout << outerLine << '\n';
    std::string emptyLine = '*' + std::string(lineWidth, ' ') + "*\n";
    for(int i = 0; i != 2 * border + 1; ++i)
    {
        if(i == border)
        {
            std::string borderLine = '*' + std::string(lineWidth, ' ') + "*\n";
            std::cout << borderLine;
        }
        else
        {
            std::cout << emptyLine;
        }
    }
    std::cout << outerLine << std::endl;
}
```

Zeilenumbruch

std::string-Objekte können durch Verwendung von +  
miteinander und mit Zeichenketten verknüpft werden

## Großes Beispiel: Erw

- Konstruktoren initialisieren komplexe Datentypen
- Konstruktoren können beliebige Parameter erhalten
- Initialisiert die Variable mit linewidth + 2 mal \*

```
void printGreeting(std::string name, int border)
{
    int lineWidth = name.size() + 2 * border;
    std::string outerLine(lineWidth + 2, '*');
    std::cout << outerLine << '\n';
    std::string emptyLine = '*' + std::string(lineWidth, ' ') + "*\n";
    for(int i = 0; i != 2 * border + 1; ++i)
    {
        if(i == border)
        {
            std::string borderString(border, ' ');
            std::cout << '*' << borderString << name << borderString << "*\n";
        }
        else
            std::cout << emptyLine;
    }
    std::cout << outerLine << std::endl;
}
```

# Konstrukturen

- Syntax für Konstruktoraufruf bei Variablendeklaration: `type name(parameter-list);`
- Definition mit `type name;` ruft parameterlosen Defaultkonstruktor auf
- **Achtung:** Expliziter Aufruf des Defaultkonstruktors mit `type name();` funktioniert nicht
  - Deklariert keine Variable vom Type `type`, sondern eine parameterlose Funktion, die einen `type` zurückgibt
- Temporäre Objekte können in Ausdrücken mit `type(parameter-list)` erstellt werden
  - `type()` ist korrekte Form für ein defaultkonstruiertes temporäres Objekt, `type` funktioniert nicht
- `foo a = foo();` hat gleichen Effekt wie `foo a;`, kann aber zusätzliche Kopieroperation erzeugen

# Großes Beispiel: Erweiterte Ausgabe der Anrede

```
void printGreeting(std::string name, int border)
{
    int lineWidth = name.size() + 2 * border;
    std::string outerLine(lineWidth + 2, '*');
    std::cout << outerLine << '\n';
    std::string emptyLine = '*' + std::string(lineWidth, ' ') + "*\n";
    for(int i = 0; i != 2 * border + 1; ++i)
    {
        if(i == border)
        {
            std::string borderString(border, ' ');
            std::cout << '*' << borderString << name << borderString << "*\n";
        }
        else
            std::cout << emptyLine;
    }
    std::cout << outerLine << std::endl;
}
```

# Bedingungen

```
if(condition)
    thenCode
else
    elseCode
```

- Wird *condition* zu **true** ausgewertet, wird *thenCode* ausgeführt, andernfalls *elseCode*
- **else** *elseCode* ist optional.
- Kann als **if(...)** ... **else if(...)** ... **else** ... verkettet werden
- *thenCode* und *elseCode* bestehen jeweils aus einem Statement, dieses kann auch ein Compound Statement sein
- Alternative Form: **if**(*init*, *condition*)  
Beispiel: **if**(**int** x=getX(); x >5){}



# Switch-Statements

```
switch (var) ← var==2
{
    case 1:
    case 2: ←
    case 3: ←
        foo(); ←
        break; ←
    case 4: return bar();
    default: foobar();
}
foo(); ←
```

- Ausführung springt zu dem **case**-Label mit dem Wert von **var**
- **default** Ziel, falls kein passender **case** existiert
  - Bei fehlendem **default** springt Ausführung an das Ende des **switch**
- Ausführung geht von dort linear weiter
- Mit **break** wird die Ausführung des **switch**-Statements abgebrochen

# Großes Beispiel: Erweiterte Ausgabe der Anrede

```
void printGreeting(std::string name, int border)
{
    int lineWidth = name.size() + 2 * border;
    std::string outerLine(lineWidth + 2, '*');
    std::cout << outerLine << '\n';
    std::string emptyLine = '*' + std::string(lineWidth, ' ') + "*\n";
    for(int i = 0; i != 2 * border + 1; ++i) ← Schleife
    {
        if(i == border)
        {
            std::string borderString(border, ' ');
            std::cout << '*' << borderString << name << borderString << "*\n";
        }
        else
            std::cout << emptyLine;
    }
    std::cout << outerLine << std::endl;
}
```

# Schleifen

- Schleifen dienen dazu, einen Codeblock mehrfach auszuführen
- Der Rumpf einer Schleife besteht immer aus einem (möglicherweise Compound) Statement
- while-Schleife

```
while( condition )  
    loop-body
```

- *loop-body* wird so lange ausgeführt, wie *condition* erfüllt ist
- *condition* wird am Anfang jeder Iteration getestet

- do-while-Schleife

```
do  
    loop-body  
while( condition )
```

- *condition* wird am Ende jeder Iteration getestet
- Mindestens eine Iteration

# Schleifen

## ■ for-Schleife

```
for(init; condition; next)  
    loop-body
```

## ■ Eine for-Schleife ist äquivalent zu:

```
init;  
while(condition)  
{  
    loop-body  
    next;  
}
```

# Schleifen

- Mit **break** können Schleifen vorzeitig beendet werden:

```
for(int i = 0; i != 20; ++i)
    if(i == 10)
        break;
```

- Die Schleife wird abgebrochen, wenn  $i==10$  gilt, obwohl die Schleifenbedingung immernoch erfüllt ist

- **continue** beendet die aktuelle Iteration und führt mit der nächsten fort

```
for(int i = 0; i != 20; ++i)
{
    foo();
    if(i >= 10)
        continue;
    bar();
}
```

- `foo` wird in jeder Iteration ausgeführt, `bar` nur in den ersten 10.

# Großes Beispiel: Erweiterte Ausgabe der Anrede

```
void printGreeting(std::string name, int border)
{
    int lineWidth = name.size() + 2 * border;
    std::string outerLine(lineWidth + 2, '*');
    std::cout << outerLine << '\n';
    std::string emptyLine = '*' + std::string(lineWidth, ' ') + "*\n";
    for(int i = 0; i != 2 * border + 1; ++i)
    {
        if(i == border)
        {
            std::string borderString(border, ' ');
            std::cout << '*' << borderString << name << borderString << "*\n";
        }
        else
            std::cout << emptyLine;
    }
    std::cout << outerLine << std::endl;
}
```

Variablen sind nur in gewissen Quellcodebereichen (Scopes) sichtbar

# Scopes

- Namen sind vom Zeitpunkt ihrer Definition an sichtbar
- Für einige Arten von Scopes kann von außerhalb auf enthaltene Namen durch Verwendung des Scope-Zugriffsoperators `::` zugegriffen werden
  - Namespaces
  - Klassen-Scopes gehören ebenfalls zu dieser Kategorie, werden aber erst später erklärt
- Für andere Scopes existieren enthaltene Namen nur bis zum Ende des Scopes
  - Compound Statements
  - Funktionsdefinitionen (enthalten Parameterdeklarationen und Funktionsrumpf)
  - Kontrollanweisungen erzeugen einen Scope, der die Anweisung selbst und das enthaltene Statement umfasst

## Scopes II

- Scopes sind hierarchisch geschachtelt
- Namen aus innerliegenden Scopes können Namen aus äußeren Scopes verdecken

```
int foo = 5;
int bar = 10;
{
    int bar = foo;
    for(int foo = 0; foo < bar; ++foo)
        std::cout << foo << "\n";
}
```

- Verdecken von Variablen macht den Code verwirrender
- Verdeckung von Variablennamen aus äußeren Scopes vermeiden



## Kompatibilität zu C Code

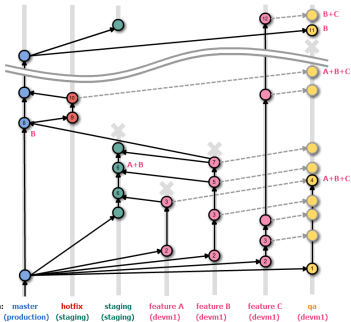
- C++ größtenteils rückwärtskompatibel zu C90
  - Primäre Inkompatibilität: Strengeres Typsystem erlaubt weniger implizite Konvertierungen
- Allerdings: **C++ ist nicht C**
  - Häufiger Fehler: C++ wie C mit Erweiterungen verwenden
  - Muss als eigene Sprache angesehen werden
- Schwierigkeit für Programmierer, die von C kommen

## Vorgehen für Übungsaufgaben

- Keine aus C bekannten Sprachelemente verwenden, die in dieser Veranstaltung noch nicht vorgestellt wurden
- Dies gilt insbesondere für:
  - C-Style Arrays
  - Pointer
  - structs
- C-Standardbibliothek nicht verwenden
- Einzige Ausnahmen:
  - Mathefunktionen aus `cmath`
  - Typen aus `cstdint` und `cstdint`
  - Absolutwertfunktionen aus `cstdlib`
    - Keine anderen Funktionen aus `cstdlib` verwenden
  - Assertions aus `cassert`



# Die Toolchain Teil 1



## Programmierungsumgebung

- Programmierung findet unter Ubuntu 20.04 LTS, 64Bit und Gcc 10.3.0 statt.
- Abgegebener Code muss in dieser Umgebung laufen.
- Übungsaufgaben können an eigenen Rechnern bearbeitet werden.
- Empfehlung: Ubuntu 20.04 LTS in virtueller Maschine
  - Oracle VirtualBox (<https://www.virtualbox.org>)
  - Benötigt Hardware-Virtualisierung Intel-VT bzw. AMD-V (ggf. im BIOS einschalten)
  - Nicht parallel mit Docker installieren
  - „Appliance“ downloaden unter <https://dei.spdns.de/cpp/ppti.html>
- Falls Sie keinen eigenen Rechner zur Verfügung haben, bitte melden.

# Version Control Systems (VCS)

- Verwaltet Dateien in einem Repository
- Erlaubt Zugriff auf vorhergehende Versionen
- Nutzen:
  - Wiederherstellung eines funktionierenden Zustandes
  - Herausfinden, welche Änderung einen Fehler ursprünglich erzeugt hat
  - Konflikte zwischen gleichzeitigen Änderungen verschiedener Entwickler identifizieren
  - Kann als Backup fungieren
  - Ungewollte Änderungen rückgängig machen
  - Dokumentation von Änderungen

## Einige Begrifflichkeiten

- Repository
  - Datenbank für die Versionshistorie der verwalteten Daten
- Commit
  - Diskreter Zustand in der Versionshistorie.
- Checkout
  - Eine Version aus dem Repository zur Bearbeitung extrahieren
- Arbeitsverzeichnis (Working Directory)
  - Aus dem Repository zur Bearbeitung ausgechecktes Verzeichnis
- Branch
  - Eigenständiger Entwicklungszweig
  - Entwicklung auf unterschiedlichen Branches ist voneinander isoliert
- Merge
  - Zusammenführen von verschiedenen Änderungssätzen

## Zentrales VCS (CVCS)

- Ein Repository, meist auf einem Server, enthält alle Daten
- Nutzer arbeitet auf einer lokalen Kopie der Daten
- Daten werden erst nach Commit auf den Server verwaltet
- Beispiele: CVS, SVN
- Ohne Netzwerkzugriff ist kein Arbeiten mit dem System möglich
- Branches global für alle sichtbar, daher höhere Schwelle Branches anzulegen
- Simultane Änderungen müssen gemergt werden, bevor sie im Repository verwaltet werden
  - Riskant aufgrund des hohen Fehlerpotentials komplexer Mergevorgänge

## Dezentrales VCS (DVCS)

- Jeder Nutzer hat lokal ein vollständiges Repository
- Einfaches Zusammenführen von Daten aus verschiedenen Repositories
- Branches zunächst nur lokal
- Häufig bessere Unterstützung von Merges als bei CVCS
- Entwicklung auch ohne Netzwerk möglich
- Daten werden bereits vor dem Merge verwaltet, bei Mergeproblemen ist die Wiederherstellung des Zustands vor dem Merge einfach möglich
- Beispiele: Git, Mercurial
- Hier: Verwendung von Git



# Konfiguration

- Git benötigt Namen und Email-Adresse für Commitinformationen:
 

```
git config --global user.name "Max Musterstudent"
git config --global user.email max@beispiel.de
```
- Vi ist standardmäßig Editor für Commitmessages  
(in gestellter VM wegen Einsteigerfreundlichkeit auf nano geändert):
 

```
git config --global core.editor nano
```
- push.default ändern (in gestellter VM bereits geschehen):
 

```
git config --global push.default simple
```

  - Nicht unbedingt notwendig, git gibt ohne allerdings Warnungen aus
- Konfiguration anzeigen: `git config --list`

## Repository anlegen

- Bestehendes Repository klonen:

```
git clone path-to-repo foldername
```

- *path-to-repo* ist üblicherweise *protocol://username@server/path*
- Für diese Veranstaltung: `git@gitlab.uni-hannover.de:ppti/2022/user.git`
- Lokales Repository wird in dem Ordner *foldername* angelegt
  - Um mit dem Repository arbeiten zu können muss erst in das Verzeichnis gewechselt werden (mit `cd foldername`).

oder

- Neues (leeres) Repository anlegen:

```
git init in dem Zielordner ausführen
```

Niemals `git init` **und** `git clone` für ein Repository verwenden

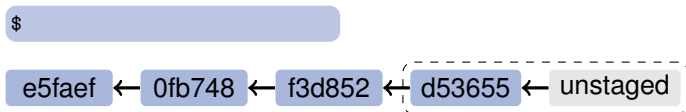
# Git Status

- `git status` zeigt den aktuellen Zustand des Arbeitsverzeichnisses an
- Übersichtlichere Ausgabe mit `git status -s`

```
$ git status
# On branch main
# 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:   foo.txt
#
$ git status -s
M foo.txt
```

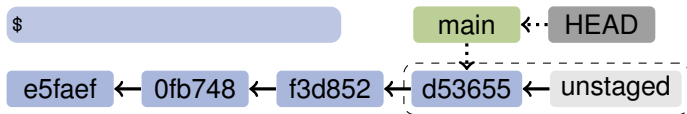
# Commits

- Ein Commit bildet den Zustand des Arbeitsverzeichnisses zum Commitzeitpunkt ab
  - Bezüglich der im Repository vermerkten Änderungen
  - Arbeitsverzeichnis kann Dateien enthalten, die nicht im Repository gespeichert werden sollen
- Commits werden in Git über ihren SHA-1 Hash identifiziert
- Ein Commit beinhaltet auch Informationen über seinen Vorgänger
  - Durch Nachverfolgung der Vorgänger können vorherige Zustände erreicht werden



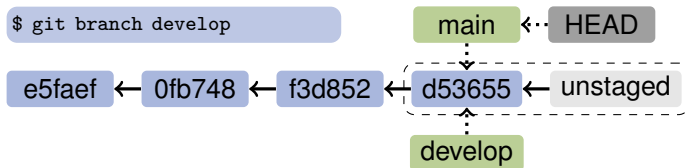
## Branches

- Branches elementar für kooperative Entwicklung
- Auch für einzelne Entwickler nützlich, um parallel an verschiedenen Komponenten zu arbeiten
- Branches in git sind lediglich Zeiger auf Commits
- Branchnamen können als Alias für Commits verwendet werden
- HEAD ist immer Alias für den aktuell ausgecheckten Commit



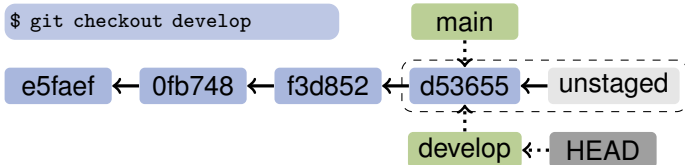
## git branch

- `git branch` zeigt alle Branches an
- `git branch branchname` legt einen neuen Branch an
  - Zeigt auf den aktuell ausgecheckten Commit



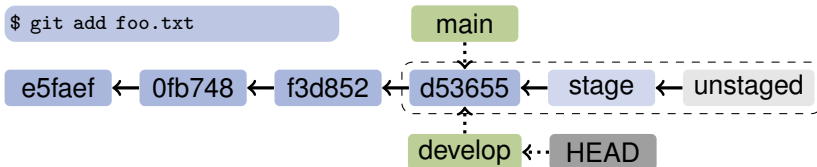
## git checkout

- `git checkout commit` checkt *commit* aus dem lokalen Repository ins Arbeitsverzeichnis aus.
- Verschiedene Optionen für *commit*:
  - Commit, identifiziert über SHA-1 Hash (eindeutiger Teilstring reicht aus)
  - Branchname
- `git checkout -b name` äquivalent zu `git branch name` gefolgt von `git checkout name`



## git add

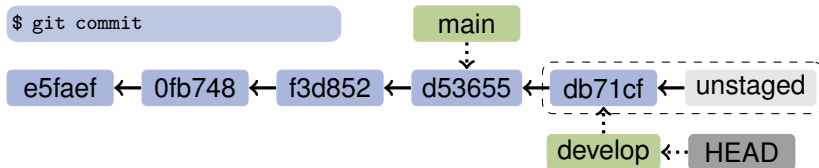
- Änderungen, die in einem Commit zusammengefasst werden sollen, müssen in git als solche markiert (staged) werden
  - Menge aller markierten Änderungen: Staging Area oder auch Stage
  - Git ignoriert Änderungen, die nicht im Stage oder im Repository sind
  - Achtung: Gestagete Änderungen sind nur für einen Commit vorgemerkt, sie werden noch nicht vom Repository verwaltet
- `git add filename` fügt *filename* zum Stage hinzu.





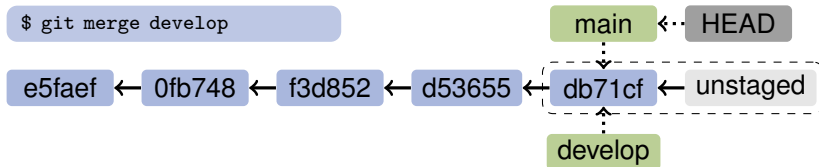
## git commit

- `git commit` führt die aktuell im Stage befindlichen Änderungen zu einem Commit zusammen
- Öffnet den Editor zum Schreiben einer Commitmessage
- `-a` fügt vor Commit alle Änderungen von bereits im Repo befindlichen Dateien dem Index hinzu
  - `git commit -a` normales Vorgehen, `git add` nur für neue Dateien notwendig



## git merge

- `git merge name` mergt den angegebenen Branch auf den aktuell aktiven
- Fast-forward-Merge, falls *name* strikter Nachfolger des aktuellen Branches ist
  - Lässt aktuellen Branch auf *name* zeigen
- Sonst: Erzeugt einen Mergecommit
- Im Fall von Konflikten: Manuell beheben, danach commiten



## Merge-Konflikte

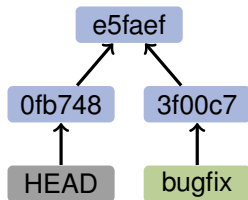
- Merge-Konflikte entstehen, wenn in den gemergten Branches unterschiedliche Änderungen an den selben Codepassagen vorgenommen wurden
- git fügt zwischen entsprechenden Markierungen beide Varianten ein und überlässt die Auflösung dem Entwickler

Inhalt HEAD:

```
foo!  
baar!  
qux!
```

Inhalt bugfix:

```
foo  
bar  
qux
```



## Merge-Konflikte

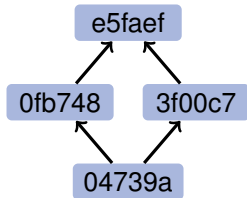
- Merge-Konflikte entstehen, wenn in den gemergten Branches unterschiedliche Änderungen an den selben Codepassagen vorgenommen wurden
- git fügt zwischen entsprechenden Markierungen beide Varianten ein und überlässt die Auflösung dem Entwickler

### Mergekonflikt:

```
foo!
<<<<<< HEAD
baar!
=====
bar
>>>>>> bugfix
qux!
```

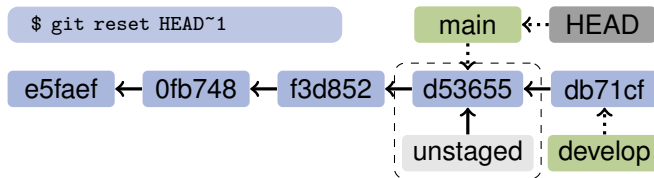
### manueller Merge:

```
foo!
bar!
qux!
```



## git reset

- `git reset commit` setzt den aktuellen Branch auf *commit*
  - *commit* kann wie immer auch ein Branch sein
  - Häufig: Relativ zu HEAD
  - HEAD~ Alias für HEAD~1, HEAD^^ für HEAD~2, ...
  - `git reset commitfile` wird nur für *file* ausgeführt
- Meist bei Mergeproblemen hilfreich
- `git reset HEAD`, um stage zurückzusetzen



## Remotes

- Als Remote-Repositories werden alle Repositories außer dem Lokalen bezeichnet
- Verwaltet über `git remote`
- `git remote` zeigt Liste aller bekannten Remotes
  - Das Repository, von dem geklont wurde, heißt `origin`
  - `git remote -v` zeigt auch Adressen der Remotes an
- `git remote add name url` macht das Repository unter *url* lokal als Remote *name* bekannt

```
$ git remote add repo git@gitlab.uni-hannover.de:repo.git
$ git remote
origin
repo
```

Schrittweise visualisiertes Beispiel nur in den Präsentationsfolien

## git fetch

- `git fetch repo` holt Branches aus dem Remote-Repository ins Lokale
  - Erzeugt Tracking-Banches, die den letzten bekannten Zustand des Branches im Remote-Repository darstellen, und updated bestehende
  - Name von Tracking-Banches: *repo/branch*
  - `git branch -r` zeigt Tracking-Banches an
  - Keine lokalen Veränderungen von Tracking-Banches möglich
- `git pull repo branch` entspricht `git fetch repo` gefolgt von `git merge repo/branch`
  - *branch* optional, wenn lokaler Branch Entsprechung im Remote-Repository hat
  - Wenn lokaler Branch aus *repo/branch* erstellt wurde oder umgekehrt
  - *repo* optional, default ist origin

```
$ git fetch origin
```

```
$ git merge origin/main
```

## git push

- `git push repo` überträgt Änderungen vom Lokalen ins Remote Repository
  - `repo` optional, ohne Angabe wird nach `origin` gepusht
  - Überträgt nur Branches mit Entsprechung auf dem Remote
  - Entsprechende Tracking-Banches werden auf den Stand des jeweiligen lokalen Branches geupdated
- `git push repo branch` pusht den lokalen Branch `branch` auf den remote Branch `branch`
  - Notwendig, um neue Branches in das Remote-Repository zu pushen
- **Wichtig:** Push erlaubt keine Änderungen der Repovergangenheit
  - Gepushte Änderungen nicht mehr mit `git reset` rückgängig machen
  - Sie wurden gewarnt!



# Guis

gitk: git

File Edit View Help

Local uncommitted changes, not checked in to index

**master** **remotes/origin/master** Merge branch 'maint'

**remotes/origin/main** Replace filepattern with pathspec for consistency

Update draft release notes to 1.8.2

Merge branch 'maint'

user-manual: Rewrite git-gc section for automatic packing

user-manual: Fix 'you - Git' -> 'you-Git' typo

user-manual: Fix 'http' -> 'HTTP' typos

user-manual: Fix 'both: so' -> 'both: so' typo

Merge branch 'sp/smart-http-content-type-check'

http\_request: reset "type" strbuf before adding

t5551: fix expected error output

Verify Content-Type from smart HTTP servers

Author	Date	Commit
Junio C Hamano <gitster@pobox.com>	2013-02-12 21	2013-02-12 21
Matthieu Moy <matthieu@openminded.org>	2013-02-12 10	2013-02-12 10
Junio C Hamano <gitster@pobox.com>	2013-02-11 05	2013-02-11 05
Junio C Hamano <gitster@pobox.com>	2013-02-11 05	2013-02-11 05
W. Trevor King <wking@redhat.com>	2013-02-10 16	2013-02-10 16
W. Trevor King <wking@redhat.com>	2013-02-10 16	2013-02-10 16
W. Trevor King <wking@redhat.com>	2013-02-10 16	2013-02-10 16
Junio C Hamano <gitster@pobox.com>	2013-02-11 05	2013-02-11 05
Jeff King <peff@peff.net>	2013-02-06 11	2013-02-06 11
Junio C Hamano <gitster@pobox.com>	2013-02-05 01	2013-02-05 01
Shawn Pearce <spearce@spearce.org>	2013-01-31 22	2013-01-31 22

SHA1 ID: 901fd180c9d19025bafefc34e131125628169bdd

Row 6 / 31918

Find next prev commit containing: Exact All fields

Search

Diff

Old version

New version

Lines of context: 3

Ignore space change

Line diff

Author: W. Trevor King <wking@redhat.com> 2013-02-10 16:10:27

Committer: Junio C Hamano <gitster@pobox.com> 2013-02-11 05:39:26

Parent: d42c7b3dc512675fae02abdf9449d48850aefb88 (user-manual: Fix 'you - Git' -> 'you-Git')

Child: d32805dce7bdc45a3e4045e999fc5d56e3b46a82 (Replace filepattern with pathspec for http\_request)

Child: e1ebf212377837d676b0b28a821c022b0a3e57f3 (Merge branch 'maint')

Branches: master, remotes/origin/main, remotes/origin/master, remotes/origin/next, remotes/origin/main

Follows: v1.8.1.3

Precedes:

user-manual: Rewrite git-gc section for automatic packing

This should have happened back in 2007 when 'git gc' learned about

Repository Edit Branch Commit Merge Remote Tools Help

Current Branch: master

Unstaged Changes

Documentation/git-add.

Documentation/user-manual

builtin/add.c

builtin/commit.c

http.c

http.h

po/de.po

po/git.pot

po/sv.po

po/vi.po

po/zh\_CN.po

remote-curl.c

t/lib-httpd/apache.conf

t/lib-httpd/broken-smart

Modified, not staged

File: po/vi.po

```
@@ -1,37 +1,38 @@
# Vietnamese translation for GIT-CORE.
# Copyright (C) 2012, Trần Ngọc Quân.
# Copyright (C) 2012-2013 Trần Ngọc Quân.
# This file is distributed under the same license as the git-core package.
# First translated by Trần Ngọc Quân <vnwildman@gmail.com>, 2012.
# First translated by Trần Ngọc Quân <vnwildman@gmail.com>, 2012-2013.
# Nguyễn Thái Ngọc Duy <pcclouds@gmail.com>, 2012.
#
msgid ""
msgstr ""
"Project-Id-Version: git-v1.8.0.1-347-gf94c3\n"
"Project-Id-Version: git-v1.8.1-476-gc3ae6\n"
"Report-Msgid-Bugs-To: Git Mailing List <git@vger.kernel.org>\n"
"POT-Creation-Date: 2012-11-30 12:40+0800\n"
"PO-Revision-Date: 2012-11-30 13:48+0800\n"
"POT-Creation-Date: 2013-01-25 12:33+0800\n"
"PO-Revision-Date: 2013-01-25 14:00+0800\n"
"Last-Translator: Trần Ngọc Quân <vnwildman@gmail.com>\n"
"Language-Team: Vietnamese <translation-team-vi@lists.sourceforge.net>\n"
"Language: vi\n"
"MIME-Version: 1.0\n"
```

Staged Changes (Will Commit)

Documentation/RelNotes

http-push.c

pe/TEAMS

t/lib-httpd.sh

t/t5551-http-fetch.sh

Commit Message:

New Commit

Amend Last Commit

Rescan

Stage Changed

Sign Off

Commit

Push

# Einrichtung der Programmierumgebung I

- In virtueller Box ein SSH-Schlüsselpaar erzeugen
  - In Termineal ("System Tools" -> "QTerminal") eingeben: `ssh-keygen -t ed25519`
  - Dabei keine Passphrase eingeben (immer Return drücken)
  - `ssh-keygen -c -f ~/.ssh/id_ed25519 -C "stefanie.studi@stud.uni-hannover.de"`
- Schlüssel konfigurieren
  - Datei `/.ssh/config` öffnen: `geany ~/.ssh/config`
  - In Datei schreiben (Beispiel):
 

```
Host gitlab.uni-hannover.de
  User stefanie.studi@stud.uni-hannover.de
  IdentityFile ~/.ssh/id_ed25519
```
- Öffentlichen Schlüssel (`~/.ssh/id_ed25519.pub`) auf GitLab hochladen
  - Bei GitLab des LUIS (<https://gitlab.uni-hannover.de>) anmelden
  - Bei "Edit Profile"->"SSH Keys" den *öffentlichen* Schlüssel (.pub) einfügen

## Einrichtung der Programmierumgebung II

- Git einrichten (Beispiel):
  - `git config --global user.name "Stefanie Studi"`
  - `git config --global user.email stefanie.studi@stud.uni-hannover.de`
- Clonen des eigenen Repos (am Beispiel-User ppti001):
  - `git clone git@gitlab.uni-hannover.de:ppti/2022/ppti001`
  - `cd ppti001`
- Remote Repo "ppti-common" hinzufügen
  - `git remote add common git@gitlab.uni-hannover.de:ppti/2022/ppti-common`
  - `git pull common main`

## Verwendeter Git Workflow (User ppti000)

Einmalige Befehle zur Einrichtung des lokalen Repositories:

```
$ git clone git@gitlab.uni-hannover.de:ppti/2022/ppti000
```

```
$ cd ppti000
```

```
$ git remote add common git@gitlab.uni-hannover.de:ppti/2022/ppti-common
```

Wöchentlich wiederholte Befehle:

Material zum aktuellen Übungsblatt herunterladen:

```
$ git pull common main
```

Alle Änderungen lokal speichern:

```
$ git commit -a
```

Lokale Änderungen hochladen:

```
$ git push origin
```

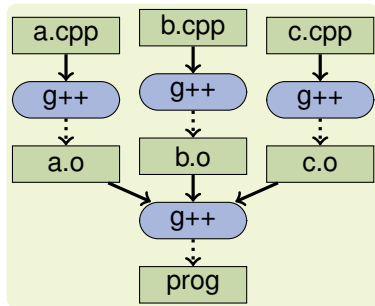
Schrittweise visualisiertes Beispiel nur in den Präsentationsfolien

## g++

- g++ ist C++-Compiler der Gnu Compiler Collection (GCC)
- Für diese Veranstaltung: Version 10.3.0
  - Standardversion von Ubuntu 20.04 LTS
  - Für andere Distributionen möglicherweise nicht im Standard Repository

## C++ kompilieren

- Die Kompilierung von C++-Code ist in mehrere Phasen unterteilt
- Für den Compileraufruf sind zwei Phasen relevant:
  - Kompilieren der einzelnen Sourcedateien (auch Compile-Units genannt) zu Objektdateien
  - Zusammenlinken der Objektdateien zu einer ausführbaren Datei



Trennung der Phasen sinnvoll  
bei vielen Sourcedateien. Geht  
aber auch in einem Rutsch.

# Compiler-Aufruf

- `g++ options [-c] [-o outfile] infiles`
  - `infiles`: Eingabe: Sourcedateien, Bibliotheken, Objektdaten
  - `-o outfile`: Ausgabedateiname (optional, default: `a.out`)
  - `-c`: Nur Kompilieren, kein Linken (optional)
  - `options`: siehe nächste Folie
  
- Beispiel: Kompilieren
  - `g++ -c -o programm.o programm.cpp`
  - Kompiliert `programm.cpp`
  - Ergebnis: Objektdatei `programm.o`
  
- Beispiel: Linken
  - `g++ -o programm p1.o p2.o`
  - Linkt `p1.o` und `p2.o`
  - Ergebnis: Ausführbares Programm `programm`

## Weiterführendes Material

- Git:
  - Kostenloses Ebook:  
<http://git-scm.com/documentation>
  - Online Tutorial:  
<http://try.github.com/>
  - Git Tutorial für SVN User:  
<https://git.wiki.kernel.org/index.php/GitSvnCrashCourse>
- g++
  - Dokumentation:  
<http://gcc.gnu.org/onlinedocs/>