Programmierkurs
Vorlesung 5

Dr. Ole Klein

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

4. Dezember 2020

Standardbibliothek

- Maps

Sprachfeatures

- Keyword auto
- Control Flow in Schleifen

Bauen von realistischen Projekten

- Der C++-Kompilierprozess
- Mehrdateiprogramme
- CMake

Maps: Nachschlagewerke in C++

- `std::array` und `std::vector` speichern Listen fixer Länge:
  - Einträge addressiert über 0-basierten, konsekutiven Index
  - zulässige Indizes beschränkt auf `[0,size()-1]`
- Ungeeignet für
  - Listen mit "Löchern" in der Indexmenge
  - Negative Indizes
  - Assoziieren von Werten mit Schlüsseln, die keine ganzen Zahlen sind

Maps: Nachschlagewerke in C++

- `std::array` und `std::vector` speichern Listen fixer Länge:
  - Einträge addressiert über 0-basierten, konsekutiven Index
  - zulässige Indizes beschränkt auf `[0,size()-1]`
- Ungeeignet für
  - Listen mit "Löchern" in der Indexmenge
  - Negative Indizes
  - Assoziieren von Werten mit Schlüsseln, die keine ganzen Zahlen sind

- Lösung: Maps als Abbildung von Werten mit Typ `Key` auf Werte mit Typ `Value`:
  - `std::map<Key,Value>` speichert Einträge in sortierter `Key`-Reihenfolge.
  - `std::unordered_map<Key,Value>` speichert Einträge in zufälliger Reihenfolge, ist bei vielen Schlüsseln deutlich schneller.

Maps: Syntax I

- Benötigen `#include <map>` bzw. `<unordered_map>`
- Verwendung identisch, im folgenden nur für `std::map` gezeigt.
- Maps werden immer leer angelegt:

```
std::map <std::string,int> shopping_list;
```

- Einträge werden beim ersten Zugriff angelegt:

```
shopping_list["cookies"] = 3; // can be inefficient
shopping_list.insert({"cookies",3}); // better version
```

- Fehlende Einträge werden beim Abfragen mit dem Standardwert (bei Zahlen: 0) initialisiert:

```
shopping_list["biscuits"]; // returns 0
```

- Die Grösse der Map kann wieder mit `size()` bestimmt werden:

```
shopping_list.size(); // returns 2 (cookies and biscuits)
```

Maps: Syntax II

- Testen, ob ein Eintrag in der Map enthalten ist:

```
// returns 1, as there is 1 entry for key biscuits
shopping_list.count("biscuits");
// returns 0, as there is no entry for key crisps
shopping_list.count("crisps");
```

  Hier wird eine Anzahl zurückgegeben, weil die Bibliothek auch Multi-Maps enthält, die einem Schlüssel mehrere Einträge zuordnen können.

- Eintrag löschen:

```
// returns 1, because 1 element removed
shopping_list.erase("biscuits");
// returns 0, because 0 elements removed
shopping_list.erase("crisps");
```

- Map komplett leeren:

```
shopping_list.clear();
```

Maps: Iterieren

- Beim Iterieren möchten wir auf Schlüssel und zugeordneten Wert zugreifen können.
- Verbesserter `for`-Loop liefert Referenz auf `std::pair<const Key,Value>` zurück:

```
for (std::pair<const std::string,int>& entry : shopping_list)
  std::cout << entry.first << ": "
            << entry.second << std::endl;
```

- Bei Verwendung von `std::map` werden die Einträge nach aufsteigender `Key`-Reihenfolge sortiert abgelaufen.
- Bei Verwendung von `std::unordered_map` ist die Reihenfolge der Einträge zufällig.
- Für Fortgeschrittene: `std::map` basiert auf einem sortierten Binärbaum, `std::unordered_map` auf einer Hashtable.

Keyword auto

- Typnamen oft lang und umständlich:

```
for (const std::pair<std::string,int>& entry : shopping_list)
  std::cout << entry.first << ": "
            << entry.second << std::endl;
```

- Typ der Einträge ist Compiler bekannt
- `auto` *rät* Variablentypen basierend auf der rechten Seite der Zuweisung:

```
auto cookies = shopping_list["cookies"]; // auto -> int
```

  - Standardmäßig immer value type (kopiert Rückgabewert).
  - Nach Bedarf mit `&` und `const` qualifizieren.

- Beispiel:

```
for (auto& entry : shopping_list)
  std::cout << entry.first << ": "
  << entry.second << std::endl;
```

Control Flow in Schleifen

- Manchmal ist es nötig, eine Schleife vorzeitig zu verlassen. Hierfür gibt es das Keyword `break`:

```
for (int step = 0 ; step < steps ; ++step) {
  bool ok = do_step(step);
  if (not ok)
    break;  // leaves the loop, skipping later steps
}
```

- Manchmal ist es nötig, eine Iteration der Schleife vorzeitig zu beenden und direkt zur nächsten zu springen. Hierfür gibt es das Keyword `continue`:

```
for (auto& sweets : shopping_list) {
  if (sweets.second == 0) {
    // we don't really have those sweets in the list
    continue; // jump to next list entry
  }
  put_in_basket(sweets);
}
```

C++-Projekte jenseits kleiner Übungen

Echte C++-Projekte sind um ein vielfaches grösser als unsere bisherigen Programme in den Übungen. Dies bringt neue Herausforderungen mit sich:

- Code-Strukturierung: Der Quellcode ist auf mehrere Dateien aufgeteilt, die jeweils zusammenhängende Funktionen enthalten.
- Code-Reuse I: Für viele Funktionen wird auf externe Bibliotheken jenseits der Standardbibliothek zurückgegriffen (grafische Oberflächen, Netzwerk, . . .).
- Code-Reuse II: Wiederverwendbare Teile des eigenen Programms sollen als Bibliothek zur Verfügung stehen.

Wichtig hierfür:

C++-Projekte jenseits kleiner Übungen

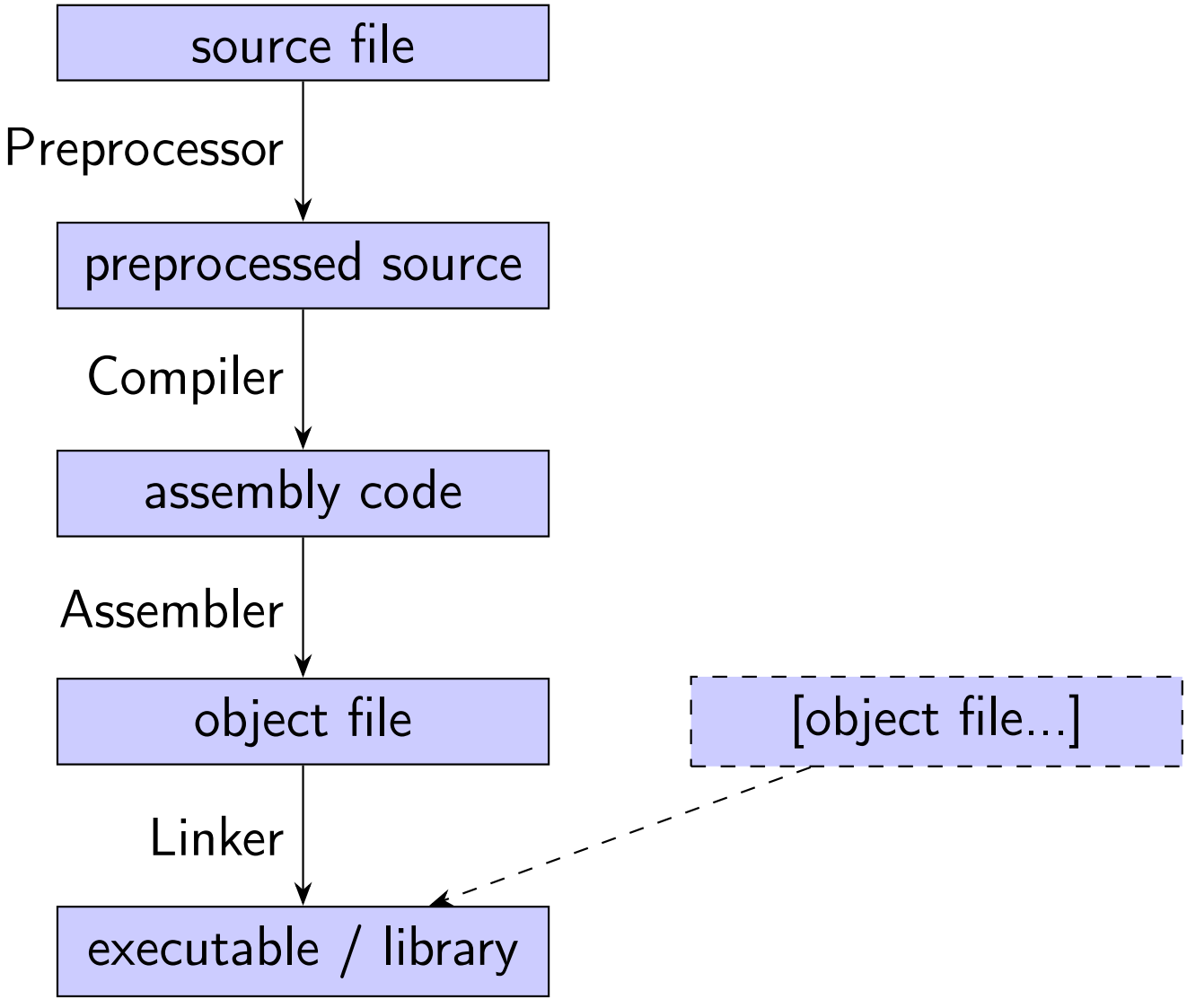
Echte C++-Projekte sind um ein vielfaches grösser als unsere bisherigen Programme in den Übungen. Dies bringt neue Herausforderungen mit sich:

- Code-Strukturierung: Der Quellcode ist auf mehrere Dateien aufgeteilt, die jeweils zusammenhängende Funktionen enthalten.
- Code-Reuse I: Für viele Funktionen wird auf externe Bibliotheken jenseits der Standardbibliothek zurückgegriffen (grafische Oberflächen, Netzwerk, ...).
- Code-Reuse II: Wiederverwendbare Teile des eigenen Programms sollen als Bibliothek zur Verfügung stehen.

Wichtig hierfür:

- Verständnis des Kompilier-/Buildprozesses
- Aufteilung von Code auf mehrere Dateien
- Verwaltung der Abhängigkeiten zwischen Dateien

Der C++-Kompilierprozess



Der Präprozessor

- Der C++-Präprozessor fügt Header-Dateien in Quellcode ein und expandiert Makros.
- Alle Zeilen, die mit `#` anfangen (sogenannte Direktiven), werden vom Präprozessor verarbeitet.
- Die wichtigsten Direktiven:
  - `#include <header>` fügt den Inhalt der Datei `header` an dieser Stelle ein.
  - `#include "header"` fügt den Inhalt der Datei `header` an dieser Stelle ein, sucht die Datei aber auch im aktuellen Verzeichnis.
  - `#define MACRO REPLACEMENT` definiert ein Makro: Immer, wenn nach dieser Zeile `MACRO` als alleinstehendes Wort auftaucht, wird es durch `REPLACEMENT` ersetzt.
  - Text zwischen `#ifdef MACRO` und `#endif` wird entfernt, wenn das angegebene Makro nicht definiert ist.
  - Text zwischen `#ifndef MACRO` und `#endif` wird entfernt, wenn das angegebene Makro definiert ist.
- Der Präprozessor kann mit `g++ -E` ausgeführt werden.

Der Compiler

- Der Compiler übersetzt den C++-Code in einfachere Befehle, die der Prozessor verstehen kann.
- Das Resultat dieses Schritts ist Assembly-Code, eine für Menschen lesbare Version der Maschinenbefehle.
- Assembly Code enthält keinerlei Variablennamen oder Schleifen mehr.
- Die Ausgabe des Compilers unterscheidet sich je nach Prozessor (Smartphone-Prozessoren verwenden andere Befehle als PCs).
- Der Compiler kann in diesem Schritt das Programm stark optimieren, wenn aktiviert (Option `-O2` oder `-O3`).
- Die Ausgabe des Compilers kann man mit `g++ -S` oder auf `https://godbolt.org` anschauen.

Der Assembler

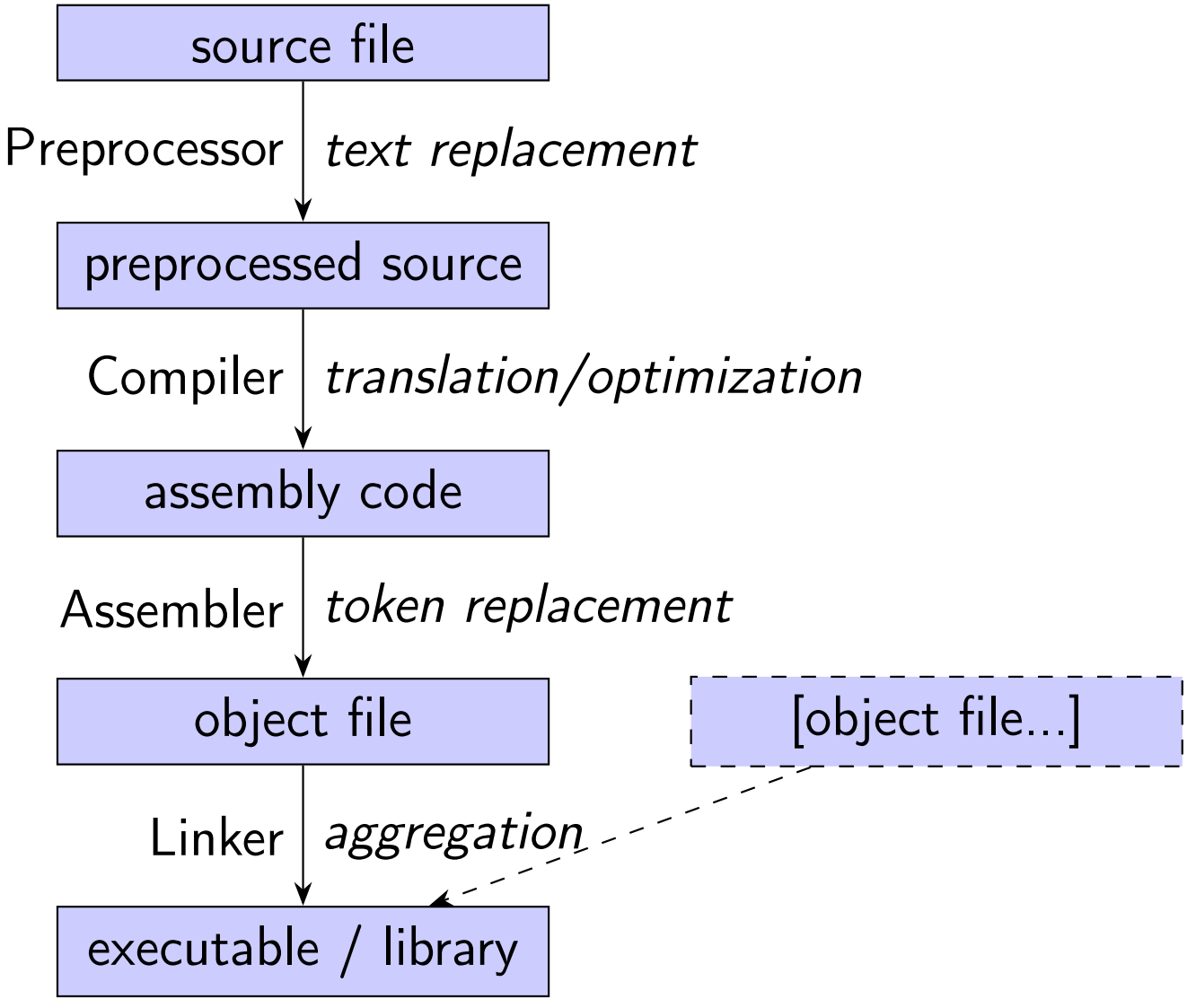
- Der Assembler verwandelt Assembly-Code in die binären Befehlscodes, die der Prozessor versteht.
- Der Assembler produziert sogenannte *object files* mit der Erweiterung `.o`.
- Achtung: das hat nichts mit den "Objekten" aus der objektorientierten Programmierung zu tun!
- Object files enthalten *object code*, das Endprodukt des Kompiliervorgangs im engeren Sinne.
- Um ein object file zu erzeugen, muss der Compiler mit der Option `-c` aufgerufen werden.

Der Linker

- Der Linker kombiniert den object code aus einem oder mehreren object files und Programmbibliotheken und erzeugt das Endprodukt des Build-Prozesses:
  - Ausführbare Dateien (executables), die von der Kommandozeile aufgerufen werden können.
  - Bibliotheken (libraries), die Funktionen enthalten und diese für andere Programme / Bibliotheken zur Verfügung stellen.
- Funktionen aus einigen Standardbibliotheken werden vom Linker automatisch gefunden, andere muss man explizit angeben.
- Linkeraufruf, um ein ausführbares Programm aus mehreren object files zu erzeugen:

```
g++ -o executable file1.o file2.o ...
```

Der C++-Kompilierprozess Revisited



Programme mit mehreren Dateien

```
double cube(double x)
{
  return x * x * x;
}
```

- Funktionen, die man mehrfach verwendet, sollte man in eine eigene Datei auslagern:
  - Einfache Wiederverwendbarkeit.
  - Bessere Übersichtlichkeit bei grösseren Programmen.
- Man benötigt meistens zwei Dateien:
  - Immer ein *header file*, das von anderen Dateien eingebunden werden kann und alle Funktionalität, die wir bereitstellen, *deklariert*.
  - Ein *implementation file*, das die eigentliche Implementierung enthält. Dies kann in manchen Situationen (siehe Templates) entfallen.

Deklaration vs. Definition

- Bevor man eine Funktion in C++ verwenden kann, muss sie deklariert werden.
- Eine Deklaration sagt dem Compiler nur, dass es eine Funktion mit einer bestimmten Signatur gibt.
- Deklarationen sind Funktionsköpfe, bei denen statt Code ein Semikolon folgt:

```
double cube(double x);
```

- Eine Definition enthält den eigentlichen Programmcode, wie bekannt.
- Eine Funktion darf beliebig oft deklariert werden, aber nur einmal definiert (one definition rule).
  - Deklaration → Header (`.hh`-Datei, der Inhalt taucht in jeder `.cc`-Datei auf, die den Header inkludiert).
  - Definition → Implementation (`.cc`-Datei).

Beispiel

cube.hh

```
// function for calculating the cube of a double
double cube(double x); // <-- declaration
```

cube.cc

```
#include "cube.hh" // preprocessor: replaced by declaration
double cube(double x) // <-- definition
{
  return x * x * x;
}
```

main.cc

```
#include <iostream>
#include "cube.hh" // also replaced by declaration
int main(int argc, char** argv)
{ // meaning of "cube" is clear thanks to declaration
  std::cout << cube(3.0) << std::endl;
}
```

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer Translation Unit (.cc-Datei).
  - langsam
  - problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
#ifndef CUBE_HH
#define CUBE_HH

// function for calculating the cube of a double
double cube(double x);

#endif // CUBE_HH
```

- Für den Namen des Guard-Makros nimmt man am besten den Dateinamen des Headers als Vorlage.

Header Guards

- Echte Programme inkludieren Header oft mehrmals in einer Translation Unit (.cc-Datei).
  - langsam
  - problematisch bei Makro-Definitionen
- Lösung: Header Guard

```
#ifndef CUBE_HH
#define CUBE_HH

// function for calculating the cube of a double
double cube(double x);

#endif // CUBE_HH
```

- Für den Namen des Guard-Makros nimmt man am besten den Dateinamen des Headers als Vorlage.

Hinweis

Alle Header-Dateien, die Sie in diesem Kurs schreiben, müssen einen Header-Guard haben!

Kompilieren des Projekts

```
g++ -Wall -std=c++14 -c cube.cc
g++ -Wall -std=c++14 -c main.cc
g++ -Wall -o example cube.o main.o
```

Probleme:

- Viel Tipparbeit
- Probleme bei späteren Änderungen:
  - Welche Datei inkludiert welche andere?
  - Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

Kompilieren des Projekts

```
g++ -Wall -std=c++14 -c cube.cc
g++ -Wall -std=c++14 -c main.cc
g++ -Wall -o example cube.o main.o
```

Probleme:

- Viel Tipparbeit
- Probleme bei späteren Änderungen:
  - Welche Datei inkludiert welche andere?
  - Was muss ich alles erneut ausführen, wenn ich eine Datei verändere?

$\Rightarrow$ Automatisierung des Prozesses durch Buildsysteme (make, CMake, qmake, autotools, . . .)

CMake

- CMake ist ein leistungsfähiges Buildsystem für Projekte in C und C++.
- Abhängigkeiten zwischen Programmen, Quell- und Headerdateien werden automatisch erkannt.
- CMake kann testen, ob das Betriebssystem bestimmte Features hat, und den Buildprozess daran anpassen.
- CMake unterstützt unterschiedliche Build-Konfigurationen:
  - Debug (für Entwicklung und Fehlersuche)
  - Release (generiert schnellere Programme für die spätere Nutzung: aktiviert Optimierung)
- CMake trennt sauber zwischen
  - Quellcode-Verzeichnis (enthält `.cc`-Dateien etc.)
  - Build-Verzeichnis (enthält alles, was automatisch generiert wird, z.B. Programme)
- CMake ist streng genommen ein Build System Generator, es erzeugt eine Konfiguration für andere Build Systems, die dann für das eigentliche Bauen verwendet werden.

CMakeLists.txt

- CMake wird über Dateien mit dem festen Namen `CMakeLists.txt` konfiguriert.
- Dateien beschreiben, wie das Projekt konfiguriert werden soll und welche Programme und Bibliotheken aus welchen `.cc`-Dateien gebaut werden sollen.
- Minimalbeispiel:

```
# Set minimum required CMake version
cmake_minimum_required(VERSION 3.5)
# Start project and set its name to ipk-demo
project(ipk-demo LANGUAGES CXX)

# Force compiler to run in C++14 mode
set(CMAKE_CXX_STANDARD 14)

# Create executable programs
add_executable(cube cubemain.cc cube.cc)
add_executable(calculator calcmain.cc basic.cc cube.cc)
```

CMake verwenden

- Im ersten Schritt erzeugt man mit CMake ein Buildsystem für `make`, indem man
  `cmake <pfad-zum-verzeichnis-mit-cmakelists.txt>`
  aufruft.
- Das Buildsystem muss in einem anderem Verzeichnis erzeugt werden als die Quelldateien.
- Eine gute Wahl ist das Unterverzeichnis `build/`:

```
mkdir build
cd build
cmake ..
```

- Wenn das Buildsystem existiert, startet man den eigentlichen Build-Prozess mit dem Befehl `make` im Verzeichnis, in dem man auch `cmake` aufgerufen hat (hier: `build/`).

CMake-Feintuning

Um genauer zu steuern, wie CMake ein Projekt baut, kann man dem CMake-Aufruf Variablen mitgeben:

```
1 cmake -DVARIABLE=VALUE <pfad>
```

Wichtige Variablen sind:

`CMAKE_CXX_COMPILER`	Der gewünschte C++-Compiler (g++, clang++, etc.)
`CMAKE_CXX_FLAGS`	Zusätzliche Flags für den Compiler, z.B. `-Wall` etc.
`CMAKE_BUILD_TYPE`	Build-Konfiguration (Release oder Debug)

Weitere Optionen kann man finden, indem man nach `cmake` im Build-Verzeichnis `ccmake .` aufruft, die Taste `t` drückt und dann durch die Liste blättert.

Wichtige CMake-Befehle

- Eine ausführbare Datei anlegen:

```
add_executable(<name> <.cc-Datei>...)
```

- Eine Bibliothek anlegen:

```
add_library(<name> <.cc-Datei>...)
```

- Target (executable oder library) gegen eine Bibliothek linken:

```
target_link_libraries(<target> PUBLIC <library>...)
```

- Unterstützung für automatische Tests aktivieren:

```
enable_testing()
```

- Test anlegen:

```
add_executable(calculator_test calculator_test.cc...)
add_test(NAME calculator_test COMMAND calculator_test)
```

Tests mit CMake

- Tests sind ein essentieller Bestandteil guter Programmierung!
- Tests prüfen, ob eine Funktion für bestimmte Inputs das erwartete Resultat produziert.
- In CMake ist ein Test ein normales Programm, das sich an folgende Konvention hält:
  - Wenn der Test erfolgreich war, gibt `main()` 0 zurück.
  - Wenn der Test fehlgeschlagen ist, gibt `main()` $0 < n < 127$ zurück.
- Vor dem Anlegen von Tests muss man `enable_testing()` aufrufen.
- Mit dem Befehl `ctest` führt CMake alle Tests aus und zeigt die Resultate auf der Konsole an.