Buildsysteme

- Automatisieren *Buildprozess* von Software-Projekten
- Für C++ speziell: Kompilieren von Objektdateien, Linken zu executables/libraries
- Grobe Kategorien:

  lowlevel Ausführen von build-Befehlen (wenn jeweilige Eingabe-Dateien sich ändern)
  Beispiele: Make, Ninja

  highlevel Automatismen zum Bestimmen der lokalen Begebenheiten (Orte von geteilten Bibliotheken, notwendige Compilerparameter, etc.)
  Eingebaute Lösungen für übliche Problemstellungen
  Generiert oft Spezifikation für ein lowlevel Buildsystem
  Beispiele: Make, GNU Autotools, Meson

Beispiel: Programmbibliothek

ld.h

```
#pragma once

double ld(double);
```

ld.cpp

```
#include <cmath>
#include "ld.h"

template<class T>
  constexpr T ln2_v
    = static_cast<T>(0.693147180559945309417232121458176568L);

double ld(double x) { return log(x) / ln2_v<double>; }
```

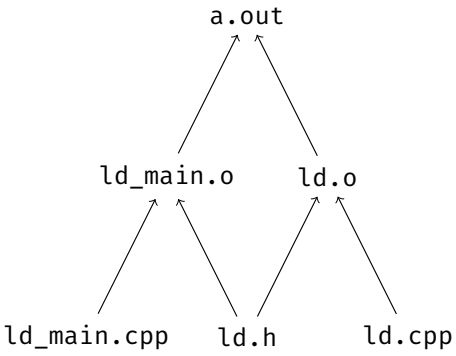
ld_main.cpp

```
#include <iostream>
#include <cerrno>
#include <cstring>

#include "ld.h"

using namespace std;

int main() {
  double x;
  cout << "x: ";
  cin >> x;
  errno = 0;
  cout << "ld(" << x << ") = " << ld(x) << endl;
  if (errno) cout << strerror(errno) << endl;
}
```

```
x: 8
ld(8) = 3
x: 0
ld(0) = -inf
Numerical result out of range
x: -4
ld(-4) = -nan
Numerical argument out of domain
```

Konzept: dependency graph

- *Dependency Graph* ist DAG von Dateien
- „*A* Nachfolger von *B*“ entspricht Aussage, dass *A* neu erzeugt werden muss, wann immer *B* geändert wird; *A* geht hervor aus *B*
- Ansatz:
  - Dependency Graph kodieren in Spezifikation für Buildsystem
  - Jede Datei hat optional Befehl wie sie erzeugt werden kann aus Vorgängern (sonst muss sie bereits existieren)
  - Buildsystem überprüft z.B. Modifikations-Zeitstempel und führt Befehle aus, wo Nachfolger jünger ist als einer der Vorgänger



Make

- Spezifikation als Datei, Name per Konvention `Makefile`
- ⇆ ist Tabulator, *nicht* Leerzeichen
- Aufruf als `make` für erstes Ziel
- Eine Regel pro erzeugtem Satz Dateien (*Ziel*), Eingabedateien (*Abhängigkeiten*) direkt nach Name erzeugter Datei, Befehle zur erzeugung darunter eingerückt
- `%` steht für mind. eines aber beliebig viele Zeichen
- `$<` ist erste Abhängigkeit, `$@` ist Ziel, `$^` ist alle Abhängigkeiten
- `.PHONY` ist Liste spezieller Regeln, die immer ausgeführt werden ohne Zeitstempel-Überprüfung

make_ld/Makefile

```
.PHONY: all
all: main

main: ld_main.o ld.o
⇆g++ -o $@ $^

%.o: %.cpp ld.h
⇆g++ -o $@ -c $<
```

Ninja

- Spezifikation als Datei, Name per Konvention `build.ninja`
- Aufruf als `ninja` baut alle Outputs die nicht auch Abhängigkeit sind
- `rules` spezifizieren Befehle zur Verwendung in `build`-Regeln
- `rules` tragen Variablen-Zuweisungen; können in `build`-Regeln überschattet werden
- Kanten im DAG entsprechen `build`-Regeln
- `build`-Regeln spezifizieren Abhängigkeiten (verfügbar in `rule` als `$in`); zusätzliche Abhängigkeiten mit `|`
- Keine patterns wie bei `make`

ninja_ld/build.ninja

```
CPPFLAGS = -Wall -Werror
CXXFLAGS = -std=c++14

rule cpp_compiler
  command = g++ $CPPFLAGS $CXXFLAGS -o
  ↪  $out -c $in
rule cpp_linker
  command = g++ $CPPFLAGS $CXXFLAGS -o
  ↪  $out $in

build main: cpp_linker ld_main.o ld.o
build ld.o: cpp_compiler ld.cpp | ld.h
build ld_main.o: cpp_compiler
↪  ld_main.cpp | ld.h
```

Ninja/GCC: Automatische Erkennung von includes

ninja_auto_ld/build.ninja

```
CPPFLAGS = -Wall -Werror
CXXFLAGS = -std=c++14

rule cpp_compiler
  deps = gcc
  depfile = $out.d
  command = g++ $CPPFLAGS $CXXFLAGS -MD -MF $out.d -o $out -c $in
rule cpp_linker
  command = g++ $CPPFLAGS $CXXFLAGS -o $out $in

build main: cpp_linker ld_main.o ld.o
build ld.o: cpp_compiler ld.cpp
build ld_main.o: cpp_compiler ld_main.cpp
```

Meson

- Eigene Syntax für Build-Spezifikationen (`meson.build`); Python-artig
- Eingebaute Funktionen, insb.:
  - `project` Zwingend erforderlich; spezifiziert Eigenschaften des gesamten Projekts, insb. name und (i.A. mehrere) Programmiersprache(n) die unterstützt werden sollen
  - `library` Kompiliert Programmbibliothek zu Objektdatei; liefert meson-object das verwendet werden kann insb. als Parameter anderer Funktionen
  - `executable` Kompiliert Übersetzungseinheit bis ausführbare Binärdatei; i.A. unter linken mit vorher übersetzten libraries
- `meson setup build` initialisiert *builddir*; Übersetzen mit `ninja -C build`

`meson_ld/meson.build`

```
project('ld', 'cpp')
ld_lib = library('ld_lib', 'ld.cpp')
executable('main', 'ld_main.cpp', link_with: ld_lib)
```

Einbinden von externen Abhängigkeiten

- In der Regel gute Idee vorgefertigte Lösungen für Probleme einzubinden statt alles selbst zu entwickeln (Z.B. Datenstrukturen aus STL)
- Woher Objekt- und Header-Dateien der benötigten Libraries beziehen?
  - Vom Betriebssystem Bereits bekannt, z.B. `iostream`
    Findet relevante Dateien auf Compiler-spezifische Weise
    Version/Verfügbarkeit von Abhängigkeiten Betriebssystem-abhängig
  - *Vendoring* Einbinden von *spezifischen Versionen* von Abhängigkeiten direkt in das Projekt
    Historisch: copy & paste
    Besser: Referenz (in einem *lockfile*) hinterlegen; vom Buildsystem beschaffen lassen

Meson: WrapDB

- Community unterhält Liste von Abhängigkeiten auf GitHub → die *Meson WrapDB*
- Erfordert Unterverzeichnis `subprojects`
- `meson wrap install gtest` installiert *googletest* unterhalb von `subprojects`
- Verwendung in `meson.build` mit `subproject` und `get_variable`

`meson_wrapdb/meson.build`

```
project('wrapdb', 'cpp')
gtest = subproject('gtest')
executable('testbin',
  'test.cpp',
  dependencies : gtest.get_variable('gtest_main_dep')
)
```