Meson Tutorial

Für C++, Java, C#, LaTeX und Linux

Dr. Günter Kolousek

2018, 2019, 2020

Inhaltsverzeichnis

1 Überblick **3**

2 Installation **3**

3 Ein erstes Meson-Projekt **4**

4 Eigenes src-Verzeichnis **5**

5 Mehrere Source-Code-Dateien **5**

6 Verwendung von Headerdateien **7**

7 Angabe der C++ Version **8**

8 Spezifizieren der Projektversion und Ausgabe von Meldungen **9**

9 Setzen von Präprozessordefinitionen **9**

10 Explizites Spezifikation von Compileroptionen **10**
10.1 Setzen beim Aufruf von meson 10
10.2 Setzen in der Datei meson.build 10
10.3 Setzen mittels Optionen 11

11 Explizite Angabe des Compilers **11**

12 Konfigurationsdaten spezifizieren **11**

13 Übersetzen als Release- oder Debugversion **13**
13.1 Setzen beim Aufruf von meson 13
13.2 Setzen in der Datei meson.build 14
13.3 Setzen mittels Optionen 14

14 Angaben bzgl. Warnungen **14**
14.1 Setzen beim Aufruf von meson 14
14.2 Setzen in der Datei meson.build 15

14.3 Setzen mittels Optionen 15

15 Verwenden von Meson-Optionen 15

16 Verwenden von Threads 16

17 QtCreator mit meson verwenden 16

18 Erstellen und Verwenden einer "static library" 17

19 Erstellen und Verwenden einer "shared library" 18

20 Verwenden einer bestehenden "static" library 19

21 Verwenden einer bestehenden "shared" library 20

22 Verwenden von Unterverzeichnissen 21

23 Precompiled Header verwenden 22
23.1 Precompiled Header mit Visual Studio 23

24 Unit-Tests mit Catch 23

25 Unit-Tests mit doctest 25

26 Erstellen eines Coverage-Reports 27

27 Anzeigen eines Stacktrace beim Absturz eines C++-Programmes 28

28 Auslesen der Versionsinformationen aus Mercurial 29

29 Erstellen eines Releases 29

30 Installation im System durchführen 30
30.1 Ab Version 0.47 33

31 Java verwenden 33

32 Java mit Unit-Tests verwenden 34

33 C# verwenden 36

34 LaTeXverwenden 37
34.1 Variante mit "normalen LaTeX" 37
34.2 Variante mit `latexmk` 37

1 Überblick

Bei Meson handelt es sich um ein plattformübergreifendes Programm zum Generieren von Buildsystemen (d.h. es ist ein *meta-build system*, auch *build system generator* genannt). Es erzeugt aus der Beschreibungsdatei `meson.build` eines Projektes standardmäßig Angaben, die das gewählte Build-Tool zum Übersetzen des Projektes verwendet.

Als Auswahl stehen derzeit die Build-Tools Ninja, Visual Studio und XCode zur Verfügung, wobei Ninja als Default verwendet wird.

$$x = x^2$$

Hilfe zu Meson gibt es entweder auf der Homepage `http://mesonbuild.com` zu finden oder aber die Option `--help` kann weiterhelfen:

```
meson --help
```

Reicht diese Hilfe nicht aus, dann kann das vorliegende Dokument durchaus weiterhelfen. Es handelt sich bei diesem Dokument um eine Mischung aus Tutorial und Rezeptbuch. D.h. arbeite dieses Tutorial vom Anfang an durch bis du ein Gefühl für Meson erlangt hast. Dann reicht es, sich die weiteren Abschnitte nach Bedarf anzueignen.

2 Installation

Die Installation ist eigentlich sehr einfach, vorausgesetzt Python 3 und das Buildsystem Ninja ist am System installiert.

Ninja wird am besten mittels des Paketmanagers deiner Wahl installiert. Auf Linux-Systemen, die auf Manjaro oder Arch Linux basieren geht dies einfach mittels nachfolgendem Befehl:

```
sudo pacman -S ninja
```

Auf der Homepage von Ninja stehen auch ausführbare Programme für verschiedene Systeme zum Herunterladen bereit.

Meson selbst ist einfach über den Paketmanager von Python zu installieren. Systemweit und mit Systemadministratorrechten geht dies auf folgende Weise:

```
sudo pip install meson
```

zu installieren.

Hat man *keine* Rechte als Systemadministrator, dann kann man auch folgendermaßen vorgehen:

```
pip install --user meson
```

Meson wird dann in `~/.local` (bzw. unter Windows unter `%APPDATA%\Python`) installiert. Dann sollte allerdings die Umgebungsvariable `PATH` um das Unterverzeichnis `bin` ergänzt werden. Also sollte unter Linux `~/.local/bin` zum `PATH` hinzugefügt werden.

3 Ein erstes Meson-Projekt hello

Nehmen wir an, dass wir ein klassisches ”Hello World” Programm schreiben wollen und es demzufolge in der folgenden Art und Weise programmiert ist:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
}
```

Für unser erstes ”Meson-Projekt” wird dieses Programm in einem eigenen Verzeichnis, dem Projektverzeichnis (es wird auch als ”source root directory” bezeichnet), abgespeichert und in dieses Verzeichnis wird auch die folgende minimale `meson.build` Datei gespeichert:

```
project('hello', 'cpp')
executable('hello', 'hello.cpp')
```

Das bedeutet, dass unser Projekt den Namen ”hello” hat, es sich um ein `C++` Projekt handelt und das ausführbare Programm ebenfalls `hello` heißt und der entsprechende Source-Code in der Datei `hello.cpp` abgelegt ist.

Syntaktisch handelt es sich um zwei Funktionsaufrufe mit jeweils zwei Argumenten, die jeweils beide Stringtypen sind. Will man Kommentare hinzufügen, dann kann man dies wie in Python erreichen: Alles ab dem Rautezeichen bis zum Ende der Zeile wird als Kommentar gewertet.

Weiters legen wir noch das Verzeichnis an, in das das gesamte Projekt übersetzt werden soll (das ”build directory”). Für uns soll es `build` heißen (es kann prinzipiell jeden beliebigen Namen haben). Danach sieht unsere Verzeichnisstruktur folgendermaßen aus:

```
hello
  build
  hello.cpp
  meson.build
```

Danach wechselt man in das ”build”-Verzeichnis und startet `meson`:

```
cd build
meson ..
```

`meson` wird darauf die Datei `meson.build` im übergeordneten Verzeichnis lesen und im aktuellen Verzeichnis die notwendigen Dateien zum Übersetzen des Projektes anlegen.

Bis hierher wurde das Projekt konfiguriert (”configure step”). Jetzt geht es weiter mit dem Bauen des Projektes (”build step”).

Dazu wechselt man in das ”build”-Verzeichnis und startet `ninja`. Bei `ninja` handelt es sich um das eigentliche Tool zum Übersetzen des Projektes, für das `meson` die notwendigen Dateien erstellt hat:

```
ninja
```

Danach wird das Programm übersetzt und kann mittels `hello` gestartet werden (vorausgesetzt der Pfadname `.` ist in `PATH` enthalten, ansonsten mittels `./hello`).

Man muss nicht notwendigerweise in das Build-Verzeichnis (also `build`) wechseln, um das Projekt zu builden, denn man kann auch `ninja` direkt aus dem Projektverzeichnis auf folgende Art aufrufen:

```
ninja -C build
```

Werden Änderungen auch in der `meson.build` getätigt muss `meson` in der Regel **nicht** mehr händisch aufgerufen werden, ein weiterer Aufruf von `ninja` reicht, da `meson` von `ninja` in solch einem Falle selbsttätig neu aufgerufen wird!

Gratulation, erstes `meson` - Projekt erfolgreich erstellt.

4 Eigenes `src`-Verzeichnis hello2

Als nächsten Schritt wollen wir unsere Source-Dateien (ok, nur eine im Moment) in ein eigenes Verzeichnis verschieben und legen daher das Verzeichnis `src` an und verschieben `hello.cpp` nach `src`. Das erledigen wir in einem neuen Projekt `hello2`.

Der Verzeichnisbaum für das neue Projekt `hello2` im Verzeichnis `hello2` sieht jetzt folgendermaßen aus:

```
hello2
  build
  meson.build
  src
    hello.cpp
```

Bitte auch die `meson.build` an den neuen Projektnamen und and den geänderten Ort von `hello.cpp` anpassen:

```
project('hello2', 'cpp')
executable('hello', 'src/hello.cpp')
```

Das Erstellen des `build`-Verzeichnisses und auch das Übersetzen des Projektes funktioniert wieder wie im Abschnitt "Ein erstes Meson-Projekt".

5 Mehrere Source-Code-Dateien hello3

Nehmen wir einmal an, dass unser Beispielprojekt aus den Dateien `main.cpp` und `hello.cpp` besteht, wobei die Ausgabe unseres glorreichen "Hello world!" in eine eigene Funktion (!) `say_hello` in der Datei `hello.cpp` ausgelagert wird und diese Funktion in der Funktion `main`, die sich in `main.cpp` befinden soll, aufgerufen wird.

D.h. die Datei `hello.cpp` sieht so aus:

```
#include <iostream>
```

```
using namespace std;

void say_hello() {
    cout << "Hello world!" << endl;
}
```

In der Datei `main.cpp` wird lediglich die Funktion `say_hello` aufgerufen:

```
void say_hello();

int main() {
    say_hello();
}
```

Die angepasste Datei `meson.build` sieht folgendermaßen aus:

```
project('hello3', 'cpp')
executable('hello', ['src/main.cpp', 'src/hello.cpp'])
```

D.h., dass im zweiten Parameter von `executable` jetzt ein Array (wie eine Liste in Python) von Quellcodedateien angegeben wird. Wir sehen und halten fest, dass in Meson auch nur ein Wert angegeben werden, wenn ein Array von Werten erwartet wird!

Unter Umständen ist es übersichtlicher die Liste der Quellcodedateien in einer eigenen Variable abzuspeichern, wobei damit die `meson.build` Datei folgendermaßen aussieht:

```
project('hello3', 'cpp')

src = ['src/main.cpp', 'src/hello.cpp']

executable('hello', src)
```

Eine weitere Verbesserung der Übersichtlichkeit ist unter Umständen dadurch gegeben, dass man Schlüsselwortparameter von Meson verwendet:

```
project('hello3', 'cpp')

src = ['src/main.cpp', 'src/hello.cpp']

executable('hello', sources : src)
```

Mehr ist nicht zu tun!

Achtung: Es ist in Meson nicht möglich mittels globbing z.B. alle Dateien eines Verzeichnisses anzugeben, da dies von den Entwicklern ausdrücklich nicht gewünscht wird. Das hat zur Folge, dass man diese Liste selber warten muss. Konkret bedeutet das, dass man beim Anlegen einer neuen Datei diese auch in der Datei `meson.build` hinzufügen muss.

6 Verwendung von Headerdateien hello4

Nehmen wir an, das wir jetzt auch über eine Headerdatei `hello.h` verfügen, die die Schnittstelle unseres glorreichen Moduls `hello.cpp` enthält, nämlich den Prototypen der Funktion `say_hello`:

```
#ifndef HELLO_H
#define HELLO_H

void say_hello();

#endif
```

Diese Headerdatei gehört eindeutig in ein anderes Unterverzeichnis unseres Projektes. Hier bietet sich `include` an. Damit sieht unser Verzeichnisbaum jetzt folgendermaßen aus (wenn du ein neues Projekt im Verzeichnis `hello4` angelegt hast):

```
hello4
  build
  include
    hello.h
  meson.build
  src
    hello.cpp
    main.cpp
```

Um das Modul richtig zu implementieren, muss auch noch die Datei `hello.cpp` angepasst werden:

```
#include <iostream>

#include "hello.h"

using namespace std;

void say_hello() {
    cout << "Hello world!" << endl;
}
```

Letztendlich muss natürlich auch noch `main.cpp` angepasst werden:

```
#include "hello.h"

int main() {
    say_hello();
}
```

Das ist ja alles gut und schön, aber jetzt muss dem Compiler noch mitgeteilt werden *wo* die Header-Dateien liegen, sonst wirst du Fehlermeldungen bekommen.

Dazu gibt es die Meson-Funktion `include_directories`, die entweder ein Include-Verzeichnis oder wieder ein Array von Include-Verzeichnissen als Parameter erhält (auch mehrere 'positional arguments' sind möglich). Die fertige Datei `meson.build` sieht dann folgendermaßen aus:

```
project('hello4', 'cpp')

inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir)
```

Auch wieder sehr einfach, nicht wahr?

7 Angabe der C++ Version — hello5

Je nach verwendetem Compiler ist die standardmäßig eingestellte C++-Version nicht unbedingt die, die man in seinem Projekt verwenden will.

Man kann daher die benötigte C++ Version für das gesamte Meson-Projekt wie folgt angeben:

```
project('hello5', 'cpp',
        default_options : ['c_std=c11', 'cpp_std=c++11'])
```

Damit wird die Version C11 für die Programmiersprache C und die Version C++11 für C++ eingestellt.

Will man die C++ Version für ein bestimmtes Build-Target überschreiben, dann kann man dies folgendermaßen erreichen:

```
project('hello5', 'cpp',
        default_options : ['c_std=c11', 'cpp_std=c++11'])

inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir,
           override_options : ['cpp_std=c++17'])
```

Meson verwendet dafür "Optionen" (→ `default_options`, `override_options`), die im Abschnitt "Verwenden von Meson-Optionen" beschrieben werden.

8 Spezifizieren der Projektversion und Ausgabe von Meldungen hello6

Abgesehen von dem Projektnamen kann man bei dem Kommando `project` auch noch andere Informationen wie z.B. die Projektversion angeben. Das Format für die Projektversion ist prinzipiell frei, nur wird empfohlen Semantic Versioning zu verwenden.

Die angegebene Version des Projektes hat an sich keine besondere Bedeutung und hat reinen Dokumentationscharakter. Man kann allerdings mit Methoden des Objektes `meson` darauf zugreifen und eine entsprechende Nachricht ausgeben. Genauso sieht es auch mit der Angabe einer Projektlizenz aus:

```
project('hello6', 'cpp',
        license : ['proprietary', 'Boost'],
        version : '0.9.0',
        meson_version : '>0.46',
        default_options : 'cpp_std=c++17')

inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']

message('project name=' + meson.project_name())
message('project license=' + meson.project_license()[0] + ',' + meson.project_license()[1])

project_version = meson.project_version()

if project_version.version_compare('<1.0.0')
  warning('not production ready')
endif

message('project version=' + meson.project_version())
message('meson version=' + meson.version())

executable('hello',
           sources : src,
           include_directories : inc_dir)
```

Man sieht hier weiters, dass man normale Meldungen und auch Warnungen ausgeben kann. Auch sieht man, dass der eingebaute Datentyp String auch über eine Methode `version_compare` verfügt mit der man Versionsinformationen vergleichen kann. Diese Methode vergleicht Strings so, dass diese gemäß Semantic Versioning verglichen werden.

9 Setzen von Präprozessordefinitionen hello7

Das Setzen von Präprozessordefinitionen funktioniert wie unter "Explizites Spezifikation von Compileroptionen" beschrieben, da eine Präprozessordefinition genauso gesetzt werden kann.

Wir bauen die Datei `hello.cpp` folgendermaßen um:

```
#include <iostream>

#include "hello.h"

using namespace std;
```

```
void say_hello() {
    cout << "Hello " << MESSAGE << '!' << endl;
}
```

Weiters ändern wir die `meson.build` folgendermaßen ab:

```
project('hello7', 'cpp',
        default_options : 'cpp_std=c++17')

add_global_arguments('-DMESSAGE="world"', language : 'cpp')

inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir)
```

Natürlich kann auch jede der anderen Arten zum Setzen von Compileroptionen verwendet werden (siehe Abschnitt "Explizites Spezifikation von Compileroptionen").

10 Explizites Spezifikation von Compileroptionen

Will man der Compilersuite – also z.B. dem g++ oder dem `clang++` – Optionen beim Aufruf mitgeben, dann kann dies auf verschiedene Arten erreicht werden:

10.1 Setzen beim Aufruf von `meson`

Will man spezielle Compileroptionen schon beim ersten Aufruf von `meson` angeben, dann kann man dies über Umgebungsvariablen tun:

```
CXXFLAGS=-Wpedantic meson ..
```

Verwendest du die Shell `fish`, dann schaue bitte im Abschnitt "Explizite Angabe des Compilers" nach wie dies in der `fish` zu erreichen ist.

10.2 Setzen in der Datei `meson.build`

Will man Optionen direkt in `meson.build` setzen, dann hat man die Möglichkeit diese global oder für das aktuelle Projekt zu setzen oder für jedes Build-Target eigens:

```
add_global_arguments('-Wpedantic', language : 'cpp')
```

Soll dies noch in Abhängigkeit des verwendeten Compilers passieren, dann kann dies folgendermaßen erreicht werden:

```
if meson.get_compiler('cpp').get_id() == 'clang++'
  add_global_arguments('-fwriteable-strings', language : 'cxx')
endif
```

Allerdings ist es so, dass `add_global_arguments` nicht für Tests herangezogen wird. Außerdem sollten diese gemäß der Dokumentation *weder* für Debug und *noch* für Optimierungsflags verwendet werden!

Daher ist es meist sinnvoller `add_project_arguments` zu verwenden, da dann die angegebenen Argumente nur im aktuellen Projekt aber nicht in einem Subprojekt zur Verfügung stehen.

Man kann Optionen auch direkt bei einem Build-Target auf folgende Art und Weise angeben:

```
executable('hello', cpp_args : '-Wpedantic')
```

10.3 Setzen mittels Optionen

Diese Möglichkeit ist detailiert im Abschnitt "Verwenden von Meson-Optionen" beschrieben.

```
meson configure -Dcpp_args=-Wpedantic
```

11 Explizite Angabe des Compilers

Sind auf einem System mehrere Compiler installiert, dann will man manchmal einen dieser Compiler gezielt auswählen. Um einen speziellen Compiler einzusetzen, ist `meson` beim ersten Aufruf folgendermaßen zu starten:

```
CXX=clang++ meson ..
```

D.h. es ist für `C++` die Variable `CXX` und für `C` die Variable `CC` zu setzen (zumindest für den Aufruf von `meson`).

Verwendet man nicht `bash`, `zsh`,... sondern die *ausgezeichnete* Shell `fish`, dann sieht der Aufruf leicht anders aus, da in `fish` das Setzen einer Variable nur für den Aufruf eines Programmes etwas anders aussieht:

```
env CXX=clang++ meson ..
```

12 Konfigurationsdaten spezifizieren hello8

Da die explizitie Spezifikation von Präprozessor- bzw. Compileroptionen mühsam ist, besteht auch die Möglichkeit, Konfigurationsdateien anzulegen und diese zu verwenden.

Dazu wird mittels `configuration_data()` ein Objekt angelegt, das danach verwendet werden kann, Konfigurationsdaten (jeweils Schlüssel und Wert) mittels der Methode `set` zu setzen. Dieses Objekt mit den gesetzten Konfigurationsdaten kann danach verwendet werden, um mittels der Funktion `configure_file` aus einer Eingabedatei eine Ausgabedatei zu erzeugen. Die Ausgabedatei ist eine weitgehende Kopie der Eingabedatei nur, dass alle Schlüssel durch die Werte ersetzt worden sind. Die Schlüssel müssen gekennzeichnet sein, indem diese durch `@` eingeschlossen sind.

So sieht die Datei `meson.build` aus:

```
project('hello8', 'cpp',
        version : '1.0.0',
        default_options : 'cpp_std=c++17')

conf_data = configuration_data()
conf_data.set('version', meson.project_version())
conf_data.set('message', 'world')
configure_file(input : 'config.h.in',
               output : 'config.h',
               configuration : conf_data)

inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir)
```

Eine dazugehörige Eingabedatei `config.h.in`, die sich in diesem konkreten Fall direkt im Projektverzeichnis befinden soll, könnte so aussehen:

```
#define VERSION "@version@"
#define MESSAGE "@message@"
```

Die daraus erzeugte Datei `config.h` wird danach folgendermaßen aussehen:

```
#define VERSION "1.0.0"
#define MESSAGE "world"
```

Zu verwenden kann man dies indem man die Datei `hello.cpp` so gestaltet:

```
#include <iostream>

#include "hello.h"
#include "config.h"

using namespace std;

void say_hello() {
    cout << "Hello " << MESSAGE << '!' << endl;
}
```

Die Verwendung der Version könnte in der Datei `main.cpp` folgendermaßen aussehen:

```
#include <iostream>

#include "hello.h"
#include "config.h"

using namespace std;
```

```
int main() {
    say_hello();
    cout << VERSION << endl;
}
```

Mit einer aktuelle Version von Meson kann man anstatt von `configuration_data()` auch ein Dictionary verwenden:

```
configure_file(input : 'config.h.in',
               output : 'config.h',
               configuration :
                 {'version' : meson.project_version(),
                  'message' : 'world'})
```

13 Übersetzen als Release- oder Debugversion

Für die Angabe in welcher Art das Projekt übersetzt werden soll, stehen wieder mehrere Wege zur Verfügung.

Für die Art wie das Projekt übersetzt werden soll, gibt es die folgenden Angaben:

- `plain` ... keine speziellen Flags werden gesetzt; nur dann verwenden, wenn man alle Flags selber setzen will.
- `debug` ... zum Debuggen; keinerlei Optimierungen; das ist der Default
- `debugoptimized` ... zum Debuggen; etliche Optimierungen werden aktiviert
- `release` ... volle Optimierungen; keine Debuginformationen
- `minsize` ... "minimale" Größe, allerdings mit Debuginformationen; werden diese nicht benötigt, dann ist die Option `--strip` beim Aufruf von `meson` anzugeben.

13.1 Setzen beim Aufruf von `meson`

```
meson --buildtype=debug ..
```

13.2 Setzen in der Datei `meson.build`

Will man direkt in `meson.build` die Art des Übersetzen angeben, dann kann man dies folgendermaßen erreichen:

```
project('hello',
        sources : ['src/hello.cpp', 'main.cpp'],
        default_options : ['buildtype=debugoptimized'])
```

13.3 Setzen mittels Optionen

Diese Möglichkeit ist detailiert im Abschnitt Verwenden von Meson-Optionen beschrieben.

```
meson configure -Dbuildtype=release
```

14 Angaben bzgl. Warnungen warnlevels

Dem Compiler kann man in der Regel mitteilen wie viele Warnungen dieser anzeigen kann. Dies ist klarerweise abhängig von der verwendeten Programmiersprache und dem eingesetztem Compiler.

Meson bietet hier eine allgemeine Schnittstelle an:

14.1 Setzen beim Aufruf von `meson`

Beim Aufruf von `meson` kann mittels der Option `--warnlevel` entweder 1, 2 oder 3 angegeben werden. Standardmäßig ist die geringste Stufe, nämlich 1, vorausgewählt.

Nehmen wir folgendes Programm her:

```
#include <iostream>

using namespace std;


int main() {
    int i{};
    int* pi{&i};
    cout << (pi < 0) << endl;
}
```

Dieses wird in der Voreinstellung ohne eine Warnung übersetzen, obwohl es ziemlich sinnlos ist und wahrscheinlich vom Programmierer auch so nicht gewollt war.

Eine Änderung auf 3 mittels der Option `--warnlevel` beim Konfigurieren des Projektes sieht folgendermaßen aus:

```
meson .. --warnlevel 3
```

Beim Übersetzen werden wir jetzt unsere verdiente Warnung erhalten!

Sollen zusätzlich alle Warnungen als Fehler betrachtet werden, dann ergänzt man diesen Befehl folgendermaßen:

```
meson .. --warnlevel 3 --werror
```

14.2 Setzen in der Datei `meson.build`

Will man die Warnung von Haus aus auf eine höhere Stufe einstellen und auch alle Warnungen als Fehler betrachten, dann kann man dazu die Datei `meson.build` wie folgt ändern:

```
project('warnlevels', 'cpp',
        default_options : ['warning_level=3', 'werror=true'])
executable('hello', 'src/hello.cpp')
```

Man bemerkt, dass der Schlüssel `warning_level` leider vom Optionennamen in der Kommandozeilenschnittstelle abweicht!

14.3 Setzen mittels Optionen

Diese Möglichkeit ist detailiert im Abschnitt "Verwenden von Meson-Optionen" beschrieben.

```
meson configure -Dwarning_level=3 -Dwerror=true
```

15 Verwenden von Meson-Optionen

Meson unterscheidet zwischen built-in Optionen und benutzerdefinierten Optionen. Built-in Optionen sind unter Built-in options auf der Homepage von Meson beschrieben.

Benutzerdefinierte Optionen sind in einer Datei `=meson_options.txt=` im Rootverzeichnis des Projektes zu definieren (siehe Build options auf der Homepage von Meson).

Hier ein Beispiel für eine Datei `meson_options.txt`:

```
option('asio_include_dir', type : 'string', value : '/home/maxi/projects/asio/include/',
  description : 'the include dir of asio')
option('spdlog_include_dir', type : 'string', value : '',
  description : 'the include dir of spdlog')
option('clipp_include_dir', type : 'string', value : '',
  description : 'the include dir of clipp')
```

Innerhalb einer Datei `meson.build` kann man mittels `get_option` auf eine Option zugreifen:

```
include_directories([get_option('asio_include_dir')])
```

Den Wert einer Option kann man mittels des Kommandos `meson configure` im nachhinein noch ändern:

```
$ meson configure -Dspdlog_include_dir=/home/maxi/projects/spdlog/include/
```

16 Verwenden von Threads thread

Um Threads in einem C++ Programm verwenden zu können, muss die entsprechende Bibliothek hinzugefügt werden.

Plattformübergreifend funktioniert das auf folgende Art und Weise:

```
project('thread', 'cpp')
thread_dep = dependency('threads')
executable('hello', 'src/hello.cpp',
           dependencies : thread_dep)
```

Das entsprechende C++ Programm könnte folgendermaßen aussehen:

```
#include <iostream>
#include <thread>

using namespace std;

int main() {
    thread t{[]{ cout << "Hello"; }};
    t.join();
    cout << " world!" << endl;
}
```

17 QtCreator mit meson verwenden — hello_qtcreator

Meson wird von QtCreator nicht direkt unterstützt. Es gibt allerdings das Skript meson2ide.py, das aus einer Datei `meson.build` ein QtCreator-Projekt erstellt. Dazu ist folgendermaßen vorzugehen:

1. meson Projekt erstellen, d.h. der normale Ablauf wie z.B. in "Ein erstes Meson-Projekt" beschrieben.
2. `meson2ide.py` in das Projektverzeichnis kopieren. Ok, das ist nicht unbedingt notwendig, aber die weitere Beschreibung basiert darauf.
3. In deiner Lieblingsshell in das Projektverzeichnis wechseln und dort das Kommando `python2 meson2ide.py build` ausführen. Damit wird im Verzeichnis `build` das QtCreator-Projekt angelegt.
4. Jetzt kann der QtCreator gestartet werden, z.B. folgendermaßen: `qtcreator build`
5. Im QtCreator in den Projekteinstellungen, d.h. → Projects folgende Änderungen vornehmen:
   a) `Build` → `Build steps`: vorhandenen `Make`-Eintrag entfernen
   b) `Build` → `Build steps`: Einen neuen `Make`-Eintrag hinzufügen und in `Override /usr/bin/make` den Pfad von dem Executable von `ninja` einsetzen (z.B. `/usr/bin/ninja`)
   c) `Run` → `Run`: Das `Executable` des Projektes entsprechend setzen (also das auszuführende Programm)
6. Jetzt noch ein beherztes `File` → `Save All`, damit auch alles gespeichert ist.

Ab jetzt kann das Projekt im QtCreator "normal" übersetzt, gestartet und auch im Debugger entsprechend nachverfolgt werden.

Lediglich eine Kleinigkeit ist zu beachten: Wird eine Datei im QtCreator hinzugefügt (oder entfernt), dann bitte unbedingt auch die Datei `meson.build` anpassen!

18 Erstellen und Verwenden einer "static library" hello_static

Um eine statische Bibliothek zu erstellen, erzeugt man ein entsprechendes Build-Target mit der Funktion `static_library`. Um das ausführbare Programm mit der statischen Bibliothek zu linken, wird beim Erstellen des Programmes mit `executable` der Schlüsselwertparameter `link_with` verwendet:

```
project('hello_static', 'cpp',
        default_options : 'cpp_std=c++17')


inc_dir = include_directories('include')

hello_lib = static_library('hello',
                           sources : 'hello/hello.cpp',
                           include_directories : inc_dir)
src = ['src/main.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir,
           link_with : hello_lib)
```

Unter Unix-artigen Betriebssystemen heißt der Dateinamen der erstellten statischen Bibliothek aus diesem Beispiel `libhello.a`. Diese erstellte statische Bibliothek wird direkt zum Executable `hello` gelinkt.

Will man direkt Linkeroptionen mitgeben, dann kann man dies mit dem Schlüsselwertparameter `link_args` erreichen. Will man Linkeroptionen für das gesamte Projekt angeben, dann kann `add_global_link_arguments` bzw. `add_project_link_arguments` zum Einsatz kommen. Siehe Dokumentation.

Anstatt `static_library` kann auch nur `library` verwendet werden, dann hängt die Art der erstellten Bibliothek von der Meson Option `default_library` (siehe Abschnitt Verwenden von Meson-Optionen) ab. Siehe für genauere Informationen in der Meson-Dokumentation nach.

Weiters kann anstatt `static_library` auch `both_libraries` verwendet werden, die sowohl eine "static" als auch eine "shared" Bibliothek erstellt. Siehe für genauere Informationen in der Meson-Dokumentation nach.

19 Erstellen und Verwenden einer "shared library" hello_shared

Nehmen wir an, dass wir unsere fantastische Funktion `say_hello` in eine shared library verpacken wollen, damit wir diese in die ungezählten, zukünftigen, extrem wichtigen Projekte verwenden können.

An sich funktioniert dies wie im Abschnitt "Erstellen und Verwenden einer "static library"" nur dass anstatt von `static_library` die Funktion `shared_library` verwendet wird:

```
project('hello_shared', 'cpp',
        default_options : 'cpp_std=c++17')


inc_dir = include_directories('include')

hello_lib = shared_library('hello',
                           sources : 'hellolib/hello.cpp',
                           include_directories : inc_dir)
src = ['src/main.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir,
           link_with : hello_lib)
```

Unter Unix-artigen Betriebssystemen heißt der Dateinamen der erstellten dynamische Bibliothek aus diesem Beispiel `libhello.so`. Diese erstellte dynamische Bibliothek wird beim Starten des Executable `hello` gelinkt.

Mittels des Schlüsselwortparameters `soversion` kann man eine Version der dynamischen Bibliothek setzen. Diese wird herangezogen, um eine Art die (unter Meson) sogenannte ”soversion”-Version zu benennen. Die Auswirkung ist, dass unter Unix-artigen Betriebssystemen mit der Versionsangabe 1 der entsprechende Dateiname `libhello.so.1` sein wird. Unter Windows wird dieser `hello-1.dll` lauten. Damit kann man eine Versionierung seiner dynamischen Bibliotheken erreichen:

```
hello_lib = shared_library('hello',
                           sources : 'hellolib/hello.cpp',
                           include_directories : inc_dir,
                           soversion : '1')
```

Weiters gibt es noch die Möglichkeit eine Version gemäß ”Semantic Versioning” anzugeben. Unter Unix-artigen Betriebssystemen wird diese Angabe verwendet, um den Dateinamen entsprechend zu setzen und weiters wird ein entsprechender ”soname”-Dateiname als symbolischer Link angelegt.

```
hello_lib = shared_library('hello',
                           sources : 'hellolib/hello.cpp',
                           include_directories : inc_dir,
                           version : '1.0.0')
```

Damit wird unter Unix-artigen Betriebssystemen die dynamische Bibliothek `libhello.so.1.0.0` erzeugt und weiters ein entsprechender symbolischer Link `libhello.so.1`.

Fehlt diese Information, dann wird von Meson die Angabe `soversion` verwendet.

Anstatt `shared_library` kann auch nur `library` verwendet werden, dann hängt die Art der erstellten Bibliothek von der Meson Option `default_library` (siehe Abschnitt Verwenden von Meson-Optionen) ab. Siehe für genauere Informationen in der Meson-Dokumentation nach.

Weiters kann anstatt `static_library` auch `both_libraries` verwendet werden, die sowohl eine ”static” als auch eine ”shared” Bibliothek erstellt. Siehe für genauere Informationen in der Meson-Dokumentation nach.

20 Verwenden einer bestehenden ”static” libraryhello_static_extern

Handelt es sich um eine installierte Bibliothek, dann ist es einfach und man kann `dependency` verwenden, wie dies bei Verwendung von Threads im Abschnitt Verwenden von Threads gezeigt wurde.

Nehmen wir aber an, dass wir unsere fantastische Funktion `say_hello` in einer externen, d.h. von diesem Projekt unabhängig übersetzten, statischen Bibliothek haben. Das eingesetzte Betriebssystem ist ein Unix-artiges Betriebssystem wie z.B. Linux.

Diese externe Bibliothek wollen wir in unserem Meson-Projekt verwenden:

```
project('hello_static_extern', 'cpp',
        default_options : 'cpp_std=c++17')

inc_dir = include_directories('include')

cc = meson.get_compiler('cpp')
lib_hello = cc.find_library('hello',
                            dirs : ['/home/maxi/hello_static_extern'])

src = ['src/main.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir,
           dependencies : lib_hello)
```

Wichtig ist:

- Der keyword-Parameter `dirs` benötigt einen absoluten Pfad zu einem Verzeichnis. Es können auch mehrere Verzeichnisse angegeben werden, da es sich um eine Liste handelt.
- Die zu suchende Datei lautet: `lib` + erstes Argument von `find_library` + `.a`, also in unserem konkreten Fall `libhello.a`.
- Dies ist ein klarer Fall, um Meson-Optionen zu verwenden (siehe Abschnitt Verwenden von Meson-Optionen).

21 Verwenden einer bestehenden ”shared” library hello_shared_extern

An sich funktioniert dies genauso wie im Abschnitt Verwenden einer bestehenden ”static” library beschrieben, mit dem Unterschied, dass auch noch der Pfad bekannt gegeben werden muss unter

dem die Shared Library zur *Laufzeit* gefunden wird.

Hierfür gibt es mehrere Möglichkeiten:

- Die Library ist systemweit installiert. In diesem Fall wird die Bibliothek auch zur Laufzeit gefunden werden.
- Die Umgebungsvariable `LD_LIBRARY_PATH` wird um das entsprechende Verzeichnis erweitert, sodass zur Laufzeit die Library gefunden wird.
- Der Pfad wird explizit in das ausführbare Programm kodiert:

  ```
  project('hello_shared_extern', 'cpp',
          default_options : 'cpp_std=c++17')

  inc_dir = include_directories('include')

  cc = meson.get_compiler('cpp')
  lib_hello = cc.find_library('hello',
                              dirs : ['/home/maxi/hello_shared_extern'])

  src = ['src/main.cpp']

  executable('hello',
             sources : src,
             include_directories : inc_dir,
             dependencies : lib_hello,
             build_rpath : '..')
  ```

  Man sieht, dass der einzige Unterschied die Angabe des RPATH ist, der für das Buildverzeichnis gelten soll (`build_rpath`). Anstatt des relativen Pfadnamens kann auch ein absoluter Pfadname angegeben werden. Dies ermöglicht das Programm von jedem beliebigen Verzeichnis aus zu starten.

  Die Angabe eines absoluten Pfades ist nicht unbedingt nötig, da es sinnvoller ist, das Programm zu installieren und für diesen Fall gibt es die Möglichkeit den RPATH mittels `install_rpath` anzugeben, da der Wert von `build_rpath` beim Installieren entfernt wird!

Es ist allerdings zu beachten, dass der Name der Library im ausführbaren Programm von Meson "leider" in unserem Fall mit `libhello.so.1` festgelegt wird (zumindest bis Meson 0.50.1) und es über `find_library` keine Möglichkeit gibt die Versionsnummer festzulegen. D.h. es wird als Versionsnummer 1 fix kodiert. Das bedeutet, dass auch der Dateinamen so heißen muss. Es bietet sich hierfür an, dass man einen symbolischen Link anlegt:

```
ln -s libhello.so libhello.so.1
```

22 Verwenden von Unterverzeichnissen hello_modular

Bei größeren Projekten ist es sinnvoll, den Sourcecode in Unterverzeichnisse aufzuteilen. Z.B. kann dies sinnvoll sein, wenn man je Build-Target ein eigenes Unterverzeichnis zur Strukturierung ein-

setzen will.

Gehen wir der Einfachheit in diesem Beispiel davon aus, dass wir unser Hello-World-Beispiel aus ”Erstellen und Verwenden einer ”static library”” folgendermaßen unterteilen wollen:

```
hello_modular
  build
  meson.build
  io
    include
      hello.h
    meson.build
    src
      hello.cpp
  src
    main.cpp
```

Die Datei `main.cpp` sieht aus wie man es von dieser erwartet:

```
#include "hello.h"

int main() {
    say_hello();
}
```

Die Datei `meson.build` sieht jetzt folgendermaßen aus:

```
project('hello_modular', 'cpp',
        default_options : 'cpp_std=c++17')

subdir('io')

inc_dir = include_directories('include')
src = ['src/main.cpp']

executable('hello',
           sources : src,
           include_directories : inc_dir,
           link_with : io_lib)
```

Hier kann man sehen, dass mittels der Funktion `subdir` das Unterverzeichnis `io` eingebunden wird. Dazu muss dieses eine Datei `meson.build` aufweisen, die allerdings **keine** Funktion `project` aufrufen darf. Die Datei `meson.build` aus dem Verzeichis `io` sieht folgendermaßen aus:

```
inc_dir = include_directories('include')

src = ['src/hello.cpp']

io_lib = static_library('io',
                        sources : src,
                        include_directories : inc_dir)
```

Wir sehen, dass wir wieder das Include-Verzeichnis angeben, das allerdings in der *übergeordneten* `meson.build` verwendet wird. Das ist gut strukturiert, da der Inhalt des Verzeichnisses `include` jetzt genau die Schnittstelle dieses Moduls darstellt!

Weiters wird die statische Bibliothek `io_lib` definiert, die ebenfalls bei der Definition des Executable verwendet wird.

23 Precompiled Header verwenden precomp_header

In größeren Projekten ist Länge der Übersetzungszeit ein durchaus ernst zu nehmendes Problem. Ein Grund liegt darin, dass die Headerdateien immer wieder eingelesen werden und übersetzt werden müssen. Dem kann man mit vorkompilierten Headerdateien entgegegenwirken. Meson bietet dafür Unterstützung an.

Dazu muss man eine Headerdatei erstellen, die alle Headerdateien inkludiert, die vorkompiliert werden sollen. In unserem konkreten Fall werden diese im Unterverzeichnis `pch` in der Datei `hello_pch.h` gespeichert:

```
#include <iostream>

#include "hello.h"
```

Danach ist noch die Datei `meson.build` folgendermaßen zu erstellen:

```
project('precomp_header', 'cpp')
inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']
executable('precomp',
           sources : src,
           include_directories : inc_dir,
           cpp_pch : 'pch/hello_pch.h')
```

23.1 Precompiled Header mit Visual Studio

An sich funktioniert dies dort genauso, nur muss im Verzeichnis `pch` eine zusätzliche `cpp` Datei erstellt werden. Nennen wir diese `hello_pch.cpp`:

```
#if !defined(_MSC_VER)
#error "This file is only for use with MSVC."
#endif

#include "hello_pch.h"
```

Auch die Datei `meson.build` muss angepasst werden:

```
project('precomp_header', 'cpp')
inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/hello.cpp']
executable('precomp',
```

```
           sources : src,
           include_directories : inc_dir,
           cpp_pch : ['pch/hello_pch.h', 'pch/hello_pch.cpp'])
```

24 Unit-Tests mit Catch unittests_catch

Nehmen wir an, dass wir die folgende Funktion in der Datei `fact.cpp` testen:

```
#include "fact.h"

// pre: n > 0
int fact(int n) {
    int res{1};

    for (int i{1}; i <= n; ++i) {
        res *= i;
    }

    return res;
}
```

Unter der Annahme, dass wir die header-only Bibliothek Catch verwenden, könnte das entsprechende Testprogramm in der Datei `test1.cpp` folgendermaßen aussehen:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "fact.h"

TEST_CASE("Factorials are computed", "[factorial]") {
    REQUIRE(fact(0) == 1);
    REQUIRE(fact(1) == 1);
    REQUIRE(fact(2) == 2);
    REQUIRE(fact(3) == 6);
    REQUIRE(fact(10) == 3628800);
}
```

Die entsprechende `meson.build` sieht dann folgendermaßen aus:

```
project('unittests', 'cpp',
        default_options : 'cpp_std=c++17')

catch_dir = include_directories('/home/.../Catch/single_include')
inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/fact.cpp']

executable('fact',
           sources : src,
```

```
            include_directories : inc_dir)

test_src = ['tests/test1.cpp', 'src/fact.cpp']

test_exe = executable('test_exe',
                    sources : test_src,
                    include_directories : [inc_dir, catch_dir])

test('test1', test_exe)
```

Wir sehen, dass wir einen Test mit der Funktion `test` unter Angabe eines Testnamens als auch des entsprechenden Executables anlegen.

Alle Tests können durch folgendem Aufruf gestartet werden:

```
meson test
```

Ein bestimmter Test kann durch Angabe des Testnamens folgendermaßen ausgewählt werden:

```
meson test test1
```

Der relevante Teil der Ausgabe wird danach folgendermaßen aussehen:

```
ninja: Entering directory `/home/.../meson_tutorial/unittests/build'
ninja: no work to do.
1/1 unittests:suite1 / test1                OK       0.00 s

Ok:                    1
Expected Fail:         0
Fail:                  0
Unexpected Pass:       0
Skipped:               0
Timeout:               0

Full log written to .../unittests/build/meson-logs/testlog.txt
```

Mittels `meson test --list` werden alle definierten Tests ausgegeben.

Weiters ist es ab einer gewissen Größe sinnvoll die Tests zu Gruppen zusammenzufassen. Das geht so, indem man den einzelnen Tests mit dem Schlüsselwort `suite` eine oder mehreren Gruppierungsnamen zuordnet:

```
test_exe = executable('test_exe',
                      sources : test_src,
                      include_directories : [inc_dir, catch_dir])

test('test1', test_exe,
     suite : 'suite1')

test('test2', test_exe,
     suite : ['suite1', 'suite2'])
```

In diesem konkreten Fall besteht die `suite1` aus den Tests `test1` und `test2`, während die `suite2` nur aus dem Test `test2` besteht.

Der Aufruf aller Tests aus `suite1` kann jetzt so gestaret werden:

```
meson test --suite suite1
```

Manchmal will man Kommandozeilenparameter dem Prozess mitgeben oder für diese Prozess Umgebungsvariablen setzen. Dann kann dies auf folgende Art und Weise erreicht werden:

```
test('test3', test_exe2, args : ['first', 'second'])
test('test4', test_exe2, env : ['key1=value1', 'key2=value2'])
```

25 Unit-Tests mit doctest unittests_doctest

Nehmen wir an, dass wir die folgende Funktion in der Datei `fact.cpp` testen:

```
#include "fact.h"

// pre: n > 0
int fact(int n) {
    int res{1};

    for (int i{1}; i <= n; ++i) {
        res *= i;
    }

    return res;
}
```

Unter der Annahme, dass wir die header-only Bibliothek doctest verwenden, könnte das entsprechende Testprogramm in der Datei `test1.cpp` folgendermaßen aussehen:

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

#include "fact.h"

TEST_CASE("Factorials are computed") {
    CHECK(fact(0) == 1);
    CHECK(fact(1) == 1);
    CHECK(fact(2) == 2);
    CHECK(fact(3) == 6);
    CHECK(fact(10) == 3628800);
}
```

Die entsprechende `meson.build` sieht dann folgendermaßen aus:

```
project('unittests', 'cpp',
        default_options : 'cpp_std=c++17')
```

```
doctest_dir = include_directories('/home/.../doctest/doctest')
inc_dir = include_directories('include')
src = ['src/main.cpp', 'src/fact.cpp']

executable('fact',
           sources : src,
           include_directories : inc_dir)

test_src = ['tests/test1.cpp', 'src/fact.cpp']

test_exe = executable('test_exe',
                  sources : test_src,
                  include_directories : [inc_dir, doctest_dir])

test('test1', test_exe)
```

Wir sehen, dass wir einen Test mit der Funktion `test` unter Angabe eines Testnamens als auch des entsprechenden Executables anlegen.

Alle Tests können durch folgendem Aufruf gestartet werden:

```
meson test
```

Ein bestimmter Test kann durch Angabe des Testnamens folgendermaßen ausgewählt werden:

```
meson test test1
```

Der relevante Teil der Ausgabe wird danach folgendermaßen aussehen:

```
[2/3] Running all tests.
1/1 test1                                   OK       0.00 s

Ok:                    1
Expected Fail:         0
Fail:                  0
Unexpected Pass:       0
Skipped:               0
Timeout:               0

Full log written to /home/.../build/meson-logs/testlog.txt
```

Mittels `meson test --list` werden alle definierten Tests ausgegeben.

Weiters ist es ab einer gewissen Größe sinnvoll die Tests zu Gruppen zusammenzufassen. Das geht so, indem man den einzelnen Tests mit dem Schlüsselwort `suite` eine oder mehreren Gruppierungsnamen zuordnet:

```
test_exe = executable('test_exe',
                      sources : test_src,
                      include_directories : [inc_dir, doctest_dir])
```

```
test('test1', test_exe,
     suite : 'suite1')

test('test2', test_exe,
     suite : ['suite1', 'suite2'])
```

In diesem konkreten Fall besteht die `suite1` aus den Tests `test1` und `test2`, während die `suite2` nur aus dem Test `test2` besteht.

Der Aufruf aller Tests aus `suite1` kann jetzt so gestaret werden:

```
meson test --suite suite1
```

Manchmal will man Kommandozeilenparameter dem Prozess mitgeben oder für diese Prozess Umgebungsvariablen setzen. Dann kann dies auf folgende Art und Weise erreicht werden:

```
test('test3', test_exe2, args : ['first', 'second'])
test('test4', test_exe2, env : ['key1=value1', 'key2=value2'])
```

26 Erstellen eines Coverage-Reports

Das Erstellen eines Coverage-Reports mittels des Tools `gcov` und `gcovr` ist einfach zu erreichen.

Dazu sollte zuerst das Tool `gcov`, am Besten mit dem Paketmanagers des verwendeten Systems, installiert werden. Danach kann man `gcovr` mittels `sudo pip install gcovr` (oder `pip install --user gcovr`, wenn keine Administratorrechte vorhanden).

Das Konfigurieren des Projektes ist folgendermaßen durchzuführen:

```
meson .. -Db_coverage=true
```

Danach wird das Projekt normal übersetzt:

```
ninja
```

Jetzt wird das Programm gestartet und danach der Coverage-Report mittels dem folgendem Befehl erstellt:

```
ninja coverage
```

Damit wird ein Coverage-Report sowohl in Textform, in XML als auch in HTML erstellt. Will man nur einen speziellen Report, wie z.B. HTML dann kann man dies z.B. folgendermaßen erreichen (alternativ `coverage-xml` oder `coverage-text`):

```
ninja coverage-html
```

27 Anzeigen eines Stacktrace beim Absturz eines C++-Programmes

Bei `backward.hpp` und `backward.cpp` handelt es sich um eine Möglichkeit leicht einen Stacktrace bei einem Absturz des Prozesses anzeigen zu lassen.

Dafür sind folgende Angaben in der `meson.build` nötig:

```
# backward
# depends on binutils-dev; has to be installed seperately!
add_global_arguments('-DBACKWARD_HAS_BFD', language : 'cpp')
add_project_link_arguments('-lbfd', language : 'cpp')
# to surpress warnings about unknown pragrams!
add_global_arguments('-Wno-unknown-pragmas', language : 'cpp')
# /backward
```

Weiters ist noch die Datei `backward.hpp` in das Includeverzeichnis zu speichern und die Datei `backward.cpp` zu den Sourcen hinzuzufügen.

Ein einfacher Stacktrace kann dann folgendermaßen aussehen:

```
$ go
Hello world!
Stack trace (most recent call last):
#3    Object "", at 0xffffffffffffffff, in
#2    Object "/tmp/template_backward/build/go", at 0x5582da72472d, in _start
#1    Object "/usr/lib/libc.so.6", at 0x7f2523292222, in __libc_start_main
#0    Source "/tmp/template_backward/build/../src/main.cpp", line 12, in main [0x5582da7248
          9:     t.join();
         10:     cout << " world!" << endl;
         11:     int* p{nullptr};
      >  12:     cout << *p << endl;
         13: }
Segmentation fault (Address not mapped to object [(nil)])
fish: "go" terminated by signal SIGSEGV (Address boundary error)
```

28 Auslesen der Versionsinformationen aus Mercurial vcs

Meist ist es sinnvoller die Versionsinformation aus einem Versionsverwaltungssystem auszulesen (als direkt zu setzen). Meson bietet dafür eine allgemeine Unterstützung, die auf folgende Art für Mercurial genutzt werden kann:

```
project('vcs', 'cpp')

changeset = vcs_tag('hgchangeset',
                    input : 'src/version.h.in',
                    output : 'version.h',
                    command : ['hg', 'id', '-i'])

executable('hello', 'src/hello.cpp')
```

Damit wird die aktuelle changeset id ermittelt. Weiters wird der Inhalt der Datei `version.h.in` gelesen und der String `@CHANGESET@` durch die aktuelle changeset id ersetzt und das Ergebnis in eine Datei `version.h` geschrieben.

D.h. die Datei `version.h.in` könnte folgendermaßen aussehen:

```
const std::string changeset = "@CHANGESET@";
```

Eine daraus generierte Datei `version.h` könnte folgenden Inhalt haben:

```
const std::string changeset = "09875fe58a22";
```

Die verwendende Datei `hello.cpp` könnte so aussehen:

```
#include <iostream>

#include "version.h"

using namespace std;

int main() {
    cout << "changeset: " << changeset << endl;
}
```

29 Erstellen eines Releases hello_dist

Öfters wird von der aktuellen Version ein Archiv benötigt, das den aktuellen Stand der Sourcecodedateien enthält. Dabei wird die letzte Version aus dem Versionsverwaltungssystem (unterstützt werden Mercurial und git) geholt und in einem Archiv `<projectname>-<projectversion>.tar.xz` gespeichert, wobei der Projektname und die Projektversion direkt aus der `meson.build` genommen werden.

Dieses Archiv wird in einem Unterverzeichnis `meson-dist` abgespeichert und enthält keinerlei Metadaten aus dem Versionsverwaltungssystem. Weiters werden vorher etwaige Tests ausgeführt und danach eine Datei mit mit der SHA-256 Checksumme erstellt.

Dazu ist folgendermaßen vorzugehen:

Zuerst ist das Projekt anzulegen und dieses versionsverwalten.

Der Verzeichnisbaum könnte folgendermaßen aussehen (siehe Verwenden von Unterverzeichnissen]]):

```
hello_dist
  build
  io
    include
    meson.build
    src
  meson.build
  src
    main.cpp
```

In `meson.build` sollte der Funktionsaufruf `project` folgendermaßen aussehen:

```
project('hello_dist', 'cpp',
     default_options : 'cpp_std=c++17',
```

```
    version : '1.0')
```

Im Verzeichnis `hello_dist`:

```
hg init
hg add io
hg add meson.build
hg add src
hg commit -m "initial commit"
```

Jetzt kann schon das Releasearchiv mittels `ninja dist` erzeugt werden:

```
cd build
ninja dist
```

Dann wird das Archiv `hello_dist-1.0.tar.xv` und zuätzlich die Datei `hello_dist-1.0.tar.xz.sha256sum` im Verzeichnis `meson-dist` erzeugt. Fertig.

30 Installation im System durchführen

Die Installation eines Targets ist in Meson eine einfache Sache: einfach den Schlüsselwortparameter `install` wie folgt verwenden:

```
executable('hello', 'src/hello.cpp',
           install : true)
```

Danach reicht ein einfaches `ninja install` (vorausgesetzt man besitzt die entsprechenden Rechte) und das Executable wird im System an der ”richtigen” Stelle installiert.

Dies funktioniert in analoger Weise auch für die anderen Build-Targets, also z.B. `shared_library`.

Will man ein spezielles Verzeichnis angeben, dann geht dies mittels des Schlüsselwortparameters `install_dir`:

```
executable('hello', 'src/hello.cpp',
           install : true,
           install_dir : 'what/ever/dir')
```

Oft gibt es auch andere Dateien, die installiert werden müseen, wie z.B. Datendateien. Betrachten wir z.B. folgendes Hello-World Programm:

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    ifstream in{"data.txt"};
    string msg;
    in >> msg;
    in.close();
```

```
    cout << "Hello " << msg << "!" << endl;
}
```

Hier geht es offensichtlich darum, dass eine Datei `data.txt` zur Verfügung stehen muss, von der dann eine Zeile gelesen wird. Dafür kann man entweder `install_data` oder `configure_file` verwenden, denn beide verfügen über einen Schlüsselwertparameter `install_dir`. Allerdings ist `configure_file` für unsere Zwecke besser, da damit die angegebene Datei auch in das Buildverzeichnis kopiert wird und damit das Programm dort auch getestet werden kann.

Bis zur Version 0.46 von Meson muss man allerdings dafür ein `configuration_data`-Objekt verwenden (siehe Konfigurationsdaten spezifizieren), auch wenn man es gar nicht benötigt (ab Version 0.47 steht dafür ein Schlüsselwertparameter `copy` zur Verfügung):

```
project('hello_install', 'cpp')

conf_data = configuration_data()

configure_file(input : 'data/data.txt',
               output : 'data.txt',
               configuration : conf_data,
               install_dir : '/tmp/hello')

executable('hello', 'hello.cpp',
           install : true,
           install_dir : '/tmp/hello')
```

Jetzt reicht ein einfaches `ninja install` und sowohl das Executable als auch die angegebene Datendatei werden in das angegebene Verzeichnis kopiert.

Will man direkt das Installationsverzeichnis beim Installieren angeben, dann kann dies folgendermaßen erreicht werden:

```
project('hello_install', 'cpp')

conf_data = configuration_data()

configure_file(input : 'data/data.txt',
               output : 'data.txt',
               configuration : conf_data,
               install_dir : '/hello')

executable('hello', 'hello.cpp',
           install : true,
           install_dir : '/hello')
```

In einer bash-kompatiblen Shell funktioniert dies so:

```
DESTDIR=/tmp ninja install
```

In der fish so:

```
env DESTDIR=/tmp ninja install
```

In diesem Fall wird sowohl das Executable als auch die Datendatei in das Verzeichnis `/tmp/hello` installiert.

Geht es nicht nur um Targets oder um Datendateien sondern auch um die Installation einer Bibliothek, die von anderen Softwareentwicklern verwendet werden soll, dann müssen ja in der Regel auch Headerdateien oder Man-Dateien im System installiert werden. Unter der Voraussetzung, dass eine Headerdatei `hello.h` und eine Manpage-Datei `hello.1` vorhanden sind, dann kann man z.B. folgende Angaben machen:

```
install_headers('hello.h', subdir : 'hello') # -> include/hello/hello.h
install_man('hello.1') # -> share/man/man1/hello.1.gz
```

Hier sieht man auch, dass relative Pfade oder keinerlei Pfadangaben sich jeweils auf die entsprechenden standardmäßigen Installationspfade beziehen! D.h. damit werden die angegebenen Dateien in die entsprechenden Unterverzeichnisse (siehe obige Kommentarzeilen) in den standardmäßigen Installationspfaden installiert.

Will man ein eigenes Installationsskript angeben, dann kann man dies mit der Methode `add_install_script` des Objektes `meson` erreichen, das man mit Argumenten versorgen kann und das auf Umgebungsvariablen zugreifen kann (siehe Dokumentation).

30.1 Ab Version 0.47

Ab der Version 0.47 von Meson kann auch direkt mit dem Programm `meson` installieren:

```
env DESTDIR=/tmp meson install --only-changed
```

Wie hier zu sehen ist, gibt es auch eine zusätzliche Option `--only-changed`, sodass nur die geänderten Dateien kopiert werden.

Weiters gibt es auch die Möglichkeit einem Target einen `install_mode` zuzuweisen:

```
project('hello_install', 'cpp')

conf_data = configuration_data()

configure_file(input : 'data/data.txt',
               output : 'data.txt',
               configuration : conf_data,
               install_mode : 'rw-------',
               install_dir : '/tmp/hello')

executable('hello', 'hello.cpp',
           install : true,
           install_mode : 'rwx------',
           install_dir : '/tmp/hello')
```

Damit werden die Rechte gemäß POSIX eingestellt. Will man zusätzlich auch den Benutzer und die Gruppe änderen, dann ist der `install_mode` als Liste anzugeben:

```
executable('hello', 'hello.cpp',
           install : true,
           install_mode : ['rwx------', 'nobody', 'nobody'],
           install_dir : '/tmp/hello')
```

Damit wird zusätzlich der Benutzer und die Gruppe angegeben. Will man eine der Angaben gleich belassen, dann ist an dieser Stelle `false` anzugeben.

Will man das Konzept der `umask` einsetzen, dann geht dies mit einer zusätzlichen Kommandozeilenoption:

```
meson --install-umask=027 ..
```

Alternativ kann man diese auch in der `meson.build` angeben:

```
project('hello_install', 'cpp'
         default_options : ['install_umask=027'])
```

31 Java verwenden — hello_java

Meson unterstützt von Haus aus auch die Programmiersprache Java und im speziellen die Erzeugung von `.jar` Dateien:

```
project('hello_java', 'java')
jar('hello', 'src/HelloWorld.java',
    main_class : 'HelloWorld',
    java_args : ['-Xlint:unchecked'])
```

Man sieht hier auch, dass man genauso auch Argumente mitgeben kann (aber nicht muss).

Will man eine `jar`-Datei aus mehreren von einander abhängigen Java-Sourcecodedateien bauen, dann muss man das Verzeichnis angeben, das die abhängigen Java-Sourcecodedateien enthält:

```
project('hello_java', 'java')

inc_dir = include_directories('src')

jar('hello', ['src/Hello.java', 'src/HelloWorld.java'],
    main_class : 'HelloWorld',
    java_args : ['-Xlint:unchecked'],
    include_directories : inc_dir)
```

Dieses Beispiel geht von folgenden Java-Klassen aus:

```
public class Hello {
    String message() {
        return "Hello, World";
    }

    String message(String guy) {
        return "Hello, " + guy;
```

```
    }
}
```

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(new Hello().message());
    }
}
```

D.h. die Klasse `HelloWorld` ist von der Klasse `Hello` abhängig!

32 Java mit Unit-Tests verwenden hello_junit

Verwendet man die Bibliothek Junit 4 dann kann das folgendermaßen aussehen:

```
meson.build
src
  Hello.java
  HelloWorld.java
tests
  hamcrest-core-x.x.jar
  junit-4.xx.jar
  TestHelloWorld.java
```

Schauen wir uns diese Verzeichnishierarchie an und beginnen mit dem einfachen Teil, nämlich dem Verzeichnis `src`. Dieses enthält den Code, der zu testen ist. Das ist in unserem Fall die Klasse `Hello`:

```
public class Hello {
    String message() {
        return "Hello, World";
    }

    String message(String guy) {
        return "Hello, " + guy;
    }
}
```

Die Klasse `HelloWorld.java` ist die eigentliche Applikation, die für unsere Testsituation eigentlich unwichtig ist, aber der Vollständigkeit halber hier angegeben wird:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(new Hello().message());
    }
}
```

Im Verzeichnis `tests` befindet sich der Programmcode für das Testprogramm, das in unserem Fall folgendermaßen aussieht:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class TestHelloWorld {
    private Hello hello;

    @Before
    public void setUp() {
        hello = new Hello();
    }

    @Test
    public void test_default_message() {
        assertEquals(hello.message(), "Hello, World");
    }

    @Test
    public void test_custom_message() {
        assertEquals(hello.message("Bob"), "Hello, Bob");
    }
}
```

Weiters befindet sich im Verzeichnis `tests` die eigentlichen Jar-Dateien für JUnit.

Jetzt fehlt nur mehr `meson.build`:

```
project('hello_junit', 'java')

jre = find_program('java')
junit_files = ':'.join(['../tests/junit-4.12.jar', '../tests/hamcrest-core-1.3.jar'])

inc_dir = include_directories('src')

sources = ['src/Hello.java', 'src/HelloWorld.java']
jar('hello', sources,
    main_class : 'HelloWorld',
    java_args : ['-Xlint:unchecked'],
    include_directories : inc_dir)

# test
test_sources = ['src/HelloWorld.java', 'tests/TestHelloWorld.java']
jar('hello_tests', test_sources,
    main_class : 'org.junit.runner.JUnitCore',
    include_directories : inc_dir,
    java_args : ['-cp', junit_files])
```

```
test('test1', jre,
     args : ['-cp', junit_files + ':hello_tests.jar', 'org.junit.runner.JUnitCore',
             'TestHelloWorld'])
```

33 C# verwenden — hello_csharp

Die Verwendung von C# (nicht mit .NET Core, d.h. es muss ein Compiler `mcs` oder `csc` vorhanden sein) ist unkompliziert und funktioniert an sich wie die Verwendung von `C++`:

```
project('hello_csharp', 'cs')
executable('hello', 'src/hello.cs')
```

Weiters wird noch die entsprechende C#-Datei benötigt:

```
using System;

public class Prog {
    static public void Main () {
        Console.WriteLine("Hello world!");
    }
}
```

Es wird beim Builden das Executable `hello.exe` erzeugt (auch unter Linux – .NET eben). Unter Linux kann natürlich die Erweiterung ohne Gefahr an Leib und Leben entfernt werden (also z.B. `mv hello.exe hello`) und es funktioniert alles wie gehabt.

34 LaTeXverwenden — latex

34.1 Variante mit "normalen LaTeX"

Es gibt in Meson keine direkte Unterstützung für LaTeX, daher muss man ein sogenanntes custom target verwenden:

```
project('latex')

latex = find_program('xelatex')

pdf = custom_target('pdf',
                    build_by_default : true,
                    build_always : true,
                    input : ['src/test.tex'],
                    output : ['test.pdf'],
                    command : [latex, '-shell-escape', '@INPUT@'])
```

Hier wurde die Variante `xelatex`, das generell eine gute Wahl darstellt.

Trotzdem gibt wird auch hier der generellen Komplexität des Übersetzungsvorganges von LaTeXnicht Rechnung getragen: Meist muss ein Dokument ein paar Mal übersetzt werden oder es zusätzlich

`bibtex` und/oder `makeindex` aufgerufen werden. Ein Tool, das dies alles automatisch erledigt ist `latexmk`!

34.2 Variante mit `latexmk`

Auf Grund des notwendigen mehrmaligen Übersetzens (in Abhängigkeit von Inhaltsverzeichnis und Referenzen) bzw. des teilweise notwendigen Aufrufes von `bibtex` (oder `biber`) und/oder `makeindex` ist die Verwendung von latexmk anzuraten, das sich sowohl um die korrekte Anzahl der Aufrufe des LaTeX-Compilers als auch von `bibtex` und/oder `makeindex` kümmert. `latexmk` wird in der Regel mit der entsprechenden LaTeXDistribution (z.B. TeXLive) installiert.

Eine entsprechende Datei `meson.build` sieht folgendermaßen aus:

```
project('latexmk')

latexmk = find_program('latexmk')

pdf = custom_target('pdf',
                    build_by_default : true,
                    input : ['src/test.tex'],
                    output : ['test.pdf'],
                    command : [latexmk, '-pdf', '-pdflatex=xelatex --shell-escape %O %S',
                               '@INPUT@'])
```