# Deklarationen

*Deklarationen* vereinbaren Namen in der aktuellen Übersetzungseinheit. I.A. werden vereinbarte Namen nicht gleich mit einer Bedeutung versehen, d.h. bei Variablen ein initialer Wert bzw. bei Funktionen und Methoden ein Körper. Wird ein initialer Wert (auch implizit) bzw. ein Körper angegeben so ist diese Deklaration auch eine *Definition*. Nur Definitionen führen dazu, dass bei Ausführung des Programms Speicher zugewiesen wird und eine Initialisierung z.B. einer Variable durchgeführt wird, wenn nötig.

In C++ gilt die *One Definition Rule*; es darf jede Funktion, Klasse, Variable, etc. maximal einmal *definiert* werden. Davon nicht beeinträchtigt ist die Möglichkeit die selbe Funktion, Klasse, Variable, etc. mehrfach zu deklarieren (wobei die Regeln wann eine Deklaration einer Variable als Definition fungiert nicht trivial sind). Zusätzlich muss jedes Programm (insb. nicht jede Übersetzungseinheit, sondern erst das aus Objektdateien zusammengefügte Programm) eine Definition enthalten für jede *potentiell verwendete* Funktion, Klasse, Variable, etc.

Mit seiner Deklaration beginnt der *syntaktische Gültigkeitsbereich* (das *scope*) eines Namens. In den meisten Fällen erstreckt sich der syntaktische Gültigkeitsbereich eines Namens ab seiner Deklaration bis zum Ende des Blocks, der die Deklaration enthält (also bis zur „nächsten“ schließenden geschweiften Klammer). Für Parameter von Funktionen und Methoden ist der syntaktische Gültigkeitsbereich zunächst stets der gesamet Körper der Funktion bzw. Methode.
Wenn in verschachtelten Blöcken (z.B. bei Schleifen innerhalb von Funktionen) ein Name erneut deklariert wird, so handelt es sich hierbei um einen *anderen* Namen. Der syntaktische Gültigkeitsbereich des „äußeren“ Namens erstreckt sich dann nicht über den syntaktischen Gültigkeitsbereich des „inneren“ Namens. Man spricht hierbei von *Überschattung*.
Das Überschatten von Namen ist nicht per se ein Fehler, jedoch oft verwirrend und daher i.A. zu vermeiden.

## Speicherklassen

Die Speicherklasse einer Variablen bestimmt insb. wann Speicher für diese Variable zugewiesen wird, wann er wieder freigegeben wird, an welcher Stelle im Arbeitsspeicher (auf dem *heap* oder auf dem *stack*) der Speicher zugewiesen wird und wann die Initialisierung der Variable (bzw. des ihr zugewiesenen Speichers) erfolgt.

Die Speicherklasse einer Variable wird bei Ihrer Deklaration durch die Verwendung eines entsprechenden Schlüsselworts angegeben. Wird keine Speicherklasse explizit angegeben, so hat die Variable Speicherklasse *automatisch*.

**automatisch** Speicher wird zugewiesen wenn die Ausführung des Programms die Definition durchläuft. Der Speicher wird auf dem stack zugewiesen. Der Speicher wird freigegeben sobald die Ausführung des Programms das Ende des syntaktischen Gültigkeitsbereiches des Variablennamens erreicht.

**statisch** Wird eine Variable unter Verwendung des Schlüsselwortes `static` vereinbart so wird ihr Speicher am Start des Programms zugewiesen. Die Initialisierung dieses Speichers (also der Variable) geschieht jedoch erst sobald die Ausführung des Programms die Definition *das erste mal* durchläuft. Durchläuft die Ausführung des Programms die Definition erneut, so hat dies *keinerlei Auswirkung*. Der zugewiesene Speicher wird erst am Ende des Programms wieder freigegeben.

**dynamisch** Wird ein Wert unter Verwendung des `new`-Operators erzeugt, so hat dieser die Speicherklasse *dynamisch*. Es wird bei Auswertung des `new`-Operators Speicher auf dem heap zugewiesen und die Freigabe dieses Speichers muss manuell durch Verwendung des `delete`-Operators erfolgen.

> Von der Verwendung des `new`-Operators sollte abgesehen werden, stattdessen sollte das pattern *Resource Acquisition Is Initialization* (RAII) angewandt werden.

*Beispiel.* Wir implementieren eine Funkion `strtok`, die eine C-Zeichenkette durchläuft und mit jedem Aufruf der Funktion *die nächste* durch eine gegebene Menge von Trennzeichen abgetrennte Teilzeichenkette zurückgibt. Es wird hierfür eine Variable mit Speicherklasse statisch verwendet, welche sich die „aktuelle“ Position in der Zeichenkette zwischen Ausführungen der Funktion „merkt“.

strtok.cpp

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

bool is_delim(char c, const char* del) {
  for (int i = 0; del[i] != '\0'; i++)
    if (del[i] == c)
      return true;
  return false;
}
int strspn(const char* str, const char* del) {
  int res = 0;
  while (str[res] != '\0')
    if (is_delim(str[res], del))
      res++;
    else
      return res;
  return res;
}
int strcspn(const char* str, const char* del) {
  int res = 0;
  while (str[res] != '\0')
    if (!is_delim(str[res], del))
      res++;
    else
      return res;
  return res;
}

char* strtok(char* str, const char* del) {
  static char* buffer;
  if (str)
    buffer = str;

  buffer += strspn(buffer, del);

  if (*buffer == '\0')
    return nullptr;

  char* tokenBegin = buffer;

  buffer += strcspn(buffer, del);
  if (*buffer != '\0')
    *(buffer++) = '\0';
```

```
  return tokenBegin;
}

int main() {
  char input[] = "one + two * (three - four)!";
  const char delim[] = "! +- (*)";
  for (char* token = strtok(input, delim); token; token = strtok(nullptr, delim))
    cout << '"' << token << '"' << endl;
}
```

### Bindungswirkung von Namen (*linkage*)

Beim Zusammenfügen mehrerer Übersetzungseinheiten zu einem Programm können verwendete Namen sich entweder auf ein Objekt innerhalb der selben Übersetzungseinheit beziehen oder auf einen Namen der über Übersetzungseinheiten hinweg zugänglich sein soll.

Es wird daher die *Bindungswirkung* von Namen im Programm unterschieden, wobei diese sowohl von der Verwendung bestimmter Schlüsselwörter abhängt, sowie von der Position der Deklaration.

Mit dem Schlüsselwort `extern` deklarierte Namen haben stets *externe Bindung*.

Variablen mit Speicherklasse statisch haben stets *interne Bindung*.

Als `const` bzw. `constexpr` deklarierte Variablen haben per Voreinstellung interne Bindung, können jedch bei Verwendung des Schlüsselworts `extern` stattdessen externe Bindung haben.

Deklarationen im *global scope* (d.h. außerhalb sämtlicher Funktionen, Klassen und anderweitiger Blöcke) haben exteren Bindung.

Namen von Klassenkomponenten haben die selbe Bindungswirkung wie der Name der Klasse.

Alle Namen auf denen keine dieser Regel zutrifft haben *keine Bindungswirkung*.

**Keine Bindungswirkung** Der Name verweist nur innerhalb seines syntaktischen Gültigkeitsbereichs auf das gleiche sprachliche Objekt (d.h. die gleiche Funktion, Klasse, Variable, etc.)

**Interne Bindung** Der Name verweist innerhalb der ganzen Übersetzungseinheit auf das gleiche Objekt

**Externe Bindung** Der Name verweist innerhalb des ganzen Programms auf das gleiche Objekt

## Namensräume

Zur Verwaltung von Namen mit externer Bindung und zur Vermeidung von Namenskollisionen kennt C++ *Namensräume*. Bei einem Namensraum handelt es sich um eine Sammlung von Deklarationen unter einen gemeinsamen Präfix. Als Präfix fungiert hierbei der Name des Namensraumes abgetrennt vom Namen innerhalb des Namensraums mit dem scope resolution operator `::` (z.B. `std::string`).

Die Namen aus einem Namensraum können mithilfe einer `using namespace`-Deklaration auch unqualifiziert sichtbar (d.h. ansprechbar auch ohne Verwendung des scope resolution operator) gemacht werden (z.B. `using namespace std`).

Namensräume können verschachtelt werden. Es wird hierfür auf der linken Seite des scope resolution operators ein wiederrum qualifizierter Name verwendet, z.B. `std::numeric_limits<double>::epsilon`.

Neue Namensräume können deklariert werden, wie folgt:

[“`inline`”] “`namespace`” [⟨Name⟩] “{” {⟨Deklaration⟩} “}”

Wird ein Namensraum als `inline` deklariert, so werden die enthaltenen Namen zusätzlich auch an der Stelle der Deklaration des Namensraums bekannt gemacht (ähnlich zu `using namespace`).

Namensraum-Deklarationen ohne ⟨Name⟩ agieren wie Namensraum-Deklarationen mit einem, nicht näher spezifizierten, eindeutigem Namen *pro Übersetzungseinheit*. Zusätzlich verhalten sich Deklarationen unbenannter Namensräume stets so als wären sie `inline`. Unbenante Namensräume haben konsequenterweise stets interne Bindung.

Namensräume im global scope (und somit auch die darin deklarierten Namen) haben externe Bindung.

Namensräumen kann ein zusätzlicher (i.d.R. kürzerer) Name gegeben werden durch Deklarationen eines *namespace alias*, wie folgt:

“`namespace`” ⟨Name⟩ “`=`” ⟨Name⟩ “`;`”

## Buildsysteme

Bei einem *Buildsystem* handelt es sich um Software mit der Zielsetzung den *Buildprozess* von Software-Projekten zu automatisieren. Speziell für C++ besteht der Buildprozess daraus Übersetzungseinheiten zu Objektdateien zu kompilieren und die resultierenden Objektdateien zu ausführbaren Binärdateien zu linken. Buildsysteme enthalten oft jedoch auch weitere Funktionalität. Z.B. zum Verwalten von externen Abhängigkeiten (Programmbibliotheken), zum Generieren von Quellcode aus für die konkrete Problemstellung angebrachteren Spezifikationen/bereichsspezifischen Programmiersprachen (DSLs; z.B. Parser- bzw. Lexer-Generatoren) oder das Zusammenfügen der kompilierten Build-Produkte zu installierbaren Softwarepaketen.

Unter einen *lowlevel* Buildsystem verstehen wir Software, die sich primär mit dem Ausführen konkret spezifizierter Schritte eines Buildprozesses beschäftigt. Beispiele von lowlevel Buildsystemen sind Make und Ninja.

Abgegrenzt davon verstehen wir unter einem *highlevel* Buildsystem Software, die zusätzlich auch umfangreiche Standardlösungen für oft auftretende Situationen bereitstellt und somit konzisere Spezifikationen ermöglicht. Weitere Eigenschaft von highlevel Buildsystemen ist die automatische Bestimmung und Anpassung an lokale Begebenheiten, wie die Verwaltung von bzw. das lokalisieren von Abhängigkeiten und adaptive Voreinstellungen für zu verwendene Compilerparameter. Beispiele von highlevel Buildsystemen sind Make[1], GNU Autotools und Meson.

[1] Make stellt diverse eingebaute Standardregeln zur Verfügung um insb. C Software-Projekte mit einem Minimum an Konfiguration bauen zu können. Die Existenz dieser Standardregeln ist wenig bekannt.

*Beispiel* (Programmbibliothek). Wir stellen eine kleines Software-Projekt zusammen, welches im Folgenden als Beispiel wieder aufgegriffen wird.

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
```

## Abhängigkeitsgraph

Buildsysteme agieren auf dem *Abhängigkeitsgraph* (*dependency graph*) des zu übersetzenden Projekts. Der Abhängigkeitsgraph ist ein azyklischer gerichteter Graph auf den Quelldateien und Übersetzungsprodukten (Objektdateien, ausführbare Binärdateien, etc.). Eine Datei $A$ ist Nachfolger von $B$ im Graph, wenn $A$ neu erzeugt werden muss (z.B. kompiliert) wann immer $B$ geändert wird. Wir sagen $A$ gehe hervor aus $B$.

*Beispiel* (Abhängigkeitsgraph). Wenn wir die Übersetzungseinheit, die das Hauptprogramm enthält (`main.cpp`), mit der Objektdatei für `ld.cpp` linken und die ausführbare Binärdatei `main` nennen, ergibt sich für das kleine Software-Projekt der Abhängigkeitsgraph in Abbildung 1.

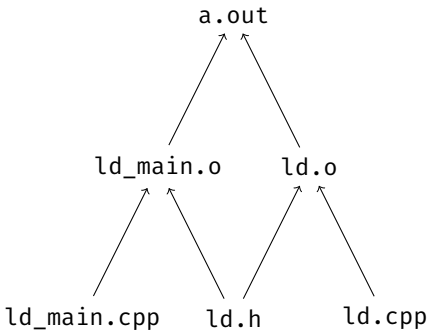


Abbildung 1: Abhängigkeitsgraph für das kleine Software-Projekt

Für Zwecke der Verwendung in einem Buildsystem müssen die Kanten des Abhängigkeitsgraphen zusätzlich annotiert werden mit Befehlen, die ausgeführt werden sollen, wann immer ein Vorgänger im Graph geändert wurde. Üblicherweise vergleicht das Buildsystem bei jeder Ausführung die Modifikations-Zeitstempel aller Dateien, die zu Knoten im Abhängigkeitsgraph korrespondieren, mit den Modifikations-Zeitstempeln der jeweiligen direkten Nachfolger. Wenn vorgefunden wird, dass ein Nachfolger einen Vorgänger hat, der neuer ist als er selbst, so wird der mit der entsprechenden Kante assoziierte Befehl ausgeführt.

Das gewünschte Verhalten ist, dass wann immer eine Datei im Projekt vom Benutzer verändert wird, ein darauf folgender Aufruf des Buildsystems *genau* jene Buildprodukte neu erstellt, die von der Änderung betroffen sind. Besonders im Rahmen großer Software-Projekte ist zudem die Parallelität der Ausführung der Build-Befehle und die Geschwindigkeit mit der Abhängigkeitsgraph durchlaufen werden kann ausschlaggebend für die Qualität des Buildsystems.

## Make

Wir betrachten Make als lowlevel-Buildsystem was der üblichen Verwendung in moderner Softwareentwicklung entspricht.

Bei Make liegt die Spezifikation des Abhängigkeitsgraphen und der assoziierten Befehle in Form einer Datei vor. Name der Datei ist per Konvention `Makefile`.

*Beispiel.* Wir formulieren ein Makefile für das kleine Software-Projekt.

make_ld/Makefile

```
.PHONY: all
all: main

main: ld_main.o ld.o
	g++ -o $@ $^

%.o: %.cpp ld.h
	g++ -o $@ -c $<
```

Ein Makefile besteht aus Regeln.

Jede Regel beginnt mit dem Namen der Datei die bei Ausführung der Regel erzeugt wird. Im Abhängigkeitsgraphen entspricht das einer Familie von Kanten zu dem gelisteten Nachfolger. Es folgt nach einem Doppelpunkt eine Leerzeichen-separierte Aufzählung der Abhängigkeiten der Regel. Im Abhängigkeitsgraphen entspricht das einer Menge Vorgängern für die von jedem Vorgänger jeweils eine Kante ausgeht zum gelisteten Nachfolger.
Es folgen beliebige viele *mit einem Tabulator-Zeichen eingerückte* Zeilen von Befehlen, die ausgeführt werden sollen, wann immer die Regel angewandt wird. Im Abhängigkeitsgraphen entspricht das dem Folgen aller durch die Regel spezifizierten Kanten gleichzeitig.

Innerhalb von Regeln können Variablen verwendet werden indem dem Namen der Variable ein Dollar-Zeichen vorgestellt wird. Der Inhalt der Variable (stets eine Zeichenkette) wird dann bei Auswertung der Regel eingefügt. Automatische Variablen, die stets definiert sind, sind insb.:

- `$@` für den Namen der Zieldatei (links vom Doppelpunkt), der für die Auswertung der Regel „verantwortlich“ ist
- `$^` für die Namen aller Abhängigkeiten der Regel
- `$<` für den Namen der ersten Abhängigkeit der Regel

Regeln können als *pattern* definiert werden. Hierfür kann das Prozent-Zeichen `%` im Namen der erzeugten Datei vorkommen. Die Regel wird dann immer in Betracht gezogen, wenn eine Datei erzeugt werden soll, deren Name zum so definierten *Muster* passt, wobei `%` für beliebig viele Zeichen steht. Im der Liste von Abhängigkeiten kann `%` ebenfalls vorkommen. Es wird dann jeweils die konkrete Zeichenkette eingesetzt für die `%` bei Auswertung der Regel im Namen der zu erzeugenden Datei steht.

Die spezielle „Zieldatei“ `.PHONY` wird nicht tatsächlich erzeugt sondern vom Buildsystem betrachtet als wäre sie stets älter als alle ihre Abhängigkeiten. Durch Angabe einer Zieldatei als Abhängigkeit von `.PHONY` kann also erreicht werden, dass die Regel für diese Zieldatei immer ausgeführt wird, wann immer sie in Betracht gezogen wird. Dies wird verwendet um Regeln zu definieren, die nicht wirklich Zieldateien entsprechen, sondern nur als Sammlung von Abhängigkeiten bzgl. Befehlen fungieren. Würde eine derartige Regel nicht als *phony* deklariert und später eine Datei mit gleichem Namen angelegt, könnte dies zu unerwatetem Verhalten führen.

Beim Aufruf von Make mit dem Befehl `make` ohne Parameter wird die Datei mit Namen `Makefile` betrachtet und die erste Regel in dieser Datei (Regeln deren Name mit einem Punkt beginnt werden hierbei zunächst ignoriert) ausgeführt. Natürlich werden hierfür erst die Abhängigkeiten dieser Regel (neu) gebaut, falls das nötig ist.

Die Definition einer phony Regel namens `all` als erste Regel ist üblich und dient dazu, dass ein einfacher

Aufruf von `make` alle zum Software-Projekt gehörigen Build-Produkte erstellt.

## Ninja

Ninja ist explizit als lowlevel Buildsystem konzipiert mit der Zielsetzung die ausufernde und obig nur Ansatzweise und skizzenhaft aufgeführte Syntax und Funktionalität von Make auf einen minimalen Satz zu reduzieren. Angedacht ist es dann die explizit weniger mächtigen Spezifikationen für ninja mit einem separaten highlevel Buildsystem (hier später Meson) automatisch aus einer möglichts konzisen und idiomatischen Spezifikation zu generieren.

Auf bei Ninja liegt die Spezifikation in Form einer Datei vor. Name der Datei ist per Konvention `ninja.build`.

*Beispiel.* Wir formulieren eine `ninja.build`-Datei für das kleine Software-Projekt.

`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.build`-Dateien teilen sich i.d.R. in drei Sektionen: die Definition von Spezifikations-weiten Voreinstellungen für einen Satz von Variablen, die Definition von _rule_s – Familien von Befehlen die durch Substitution von Variablen aus gemeinsamen Vorlagen entstehen und schließlich die Spezifikation des Abhängigkeitsgraphs.

Die Substitution von Variablen funktioniert i.W. identisch mit Make durch Vorstellung eines Dollar-Zeichens `$`.

Regeln beginnen mit dem Schlüsselwort `rule` gefolgt von einem Nutzer-gewähltem Bezeichner für die Regel. Es folgt dann ein jeweils eingerückter Satz von Belegungen von Variablen mit Werten. Manche Variablen tragen hierbei spezielle Bedeutung. Insb. gibt `command` den Befehl an, der bei Ausführung der Regel ausgeführt werden soll.

Die Spezifikation des Abhängigkeitsgraphen geschieht in Form eines Satzes von `build`-Regeln. Jede `build`-Regel beginnt mit dem Schlüsselwort `build` gefolgt von einer Leerzeichen-separierten Liste von Dateien, die bei Ausführung der Regel erzeugt werden (die *outputs*). Es folgt ein Doppelpunkt und der Name einer Regel. Danach folgt eine Leerzeichen-separierte Liste von Abhängigkeiten. Optional können *implizite Abhängigkeiten* nach einem vertikalen Strich `|` angegeben werden.

Die outputs stehen der Regel in der Variable `out` zur Verfügung, die nicht-impliziten Abhängigkeiten in der Variable `in`.

### Automatische Erkennung von Präprozessor include-Direktiven

`gcc` unterstützt spezielle Kommandozeilenparameter um beim Ausführen des Präprozessors zusätzlich eine separate Datei anzulegen (Dateiendung per Konvention `.d`) in der festgehalten wird, welche anderen Dateien zusätzlich betrachtet wurden. Insb. mit `#include` eingebundene Dateien tauchen dann namentlich in der `.d`-Datei auf.

Ninja unterstützt es direkt nach Ausführung des compilers die dabei erzeugte `.d`-Datei auszuwerten. Ninja speichert diese Information dann in einer versteckten Datei `.ninja_deps`. Bei zukünftigen Ausführungen liest Ninja automatisch stets auch die `.ninja_deps`-Datei und fügt dort spezifiizerte implizite Abhängigkeiten dem Abhängigkeitsgraph hinzu. So kann insgesamt erreicht werden, dass, auch ohne explizite Angabe welche header-Dateien eine Übersetzungseinheit inkludiert, Ninja die Übersetzungseinheit neu kompiliert, wann immer eine der inkludierten header-Dateien angepasst wird.

*Beispiel.* Wir übergeben an `gcc` in der Regel `cpp_compiler` geeignete Kommandozeilenparameter um eine `.d`-Datei zu erzeugen. Wir instruieren zudem ninja (durch setzen der Variablen `deps` und `depfile`) die so erzeugte `.d`-Datei auch auszuwerten. Wir können dann auf die explizite Angabe der header-Datei `ld.h` als implizite Abhängigkeit von `ld.o` und `ld_main.o` verzichten.

`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

Meson ist ein highlevel Buildsystem. Durch Erkennung der lokalen Begebenheiten und Interpretation einer wesentlich abstrakteren Spezifikation in einer eigenen (nicht-Turing-vollständigen[2]) Sprache eine Spezifikation für Ninja erezugt. Die eigentliche Ausführung der Build-Befehle geschieht dann mit Ninja.

Name der Datei, die die Meson-Spezifikation für ein Projekt enthält, ist per Konvention `meson.build`.

Ein Auszug aus der Grammatik der Spezifikationssprache von Meson:

⟨Identifier⟩ → `/[a-zA-Z_]/` {`/[a-zA-Z_0-9]/`}
⟨Literal⟩ → `/[1-9]/` {`/[0-9]/`}
| "'" ⟨String⟩ "'"
| "[" [⟨Expression⟩ {"," ⟨Expression⟩}] "]"

[2] https://de.wikipedia.org/wiki/Turing-Vollständigkeit

⟨AssignStmt⟩ → ⟨Identifier⟩ "=" ⟨Expression⟩
⟨PostfixExpr⟩ → ⟨Literal⟩ | ⟨Identifier⟩
| ⟨PostfixExpr⟩ "[" ⟨Expression⟩ "]"
| ⟨FunExpr⟩
| ⟨Identifier⟩ "." ⟨FunExpr⟩
⟨FunExpr⟩ → ⟨Identifier⟩ "(" [⟨PosArgs⟩] [⟨KWArgs⟩] ")"
⟨PosArgs⟩ → ⟨Expression⟩ {"," ⟨Expression⟩}
⟨KWArgs⟩ → ⟨Identifier⟩ ":" ⟨Expression⟩ ["," ⟨KWArgs⟩]

Hierbei sei eine ⟨Expression⟩ ein Ausdruck von ⟨PostfixExpr⟩ unter Verwendung der Operatoren `-`, `+`, `*`, `/`, `%`, `==`, `!=`, `>`, `<`, `>=`, `<=`, `in`, `not in`, `and`, `or` und `not` mit konventionellen Bindungsstärken und runder Klammern.

Im Unterschied zu C++ kennt die Meson-Spezifikationssprache keine Vereinbarungen von Variablen. Neue Variablen können ohne vorherige Vereinbarung einfach zugewiesen werden.

Für unsere Zwecke relevante in Meson eingebaute Funktionen (können also als ⟨Identifier⟩ verwendet werden ohne vorher definiert zu werden) sind:

**`project`** Ein Aufruf der Funktion `project` ist in jeder Meson-Spezifikation zwingend erforderlich.
Positionale Parameter (⟨PosArgs⟩) sind ein string der als Name des Projekts als Ganzes verwendet wird und dann beliebig viele weitere string-Parameter die die in diesem Projekt verwendete Programmiersprachen auflisten – für unsere Zwecke nur der string `'cpp'`.

**`library`** Jeder Aufruf der Funktion `library` führt dazu, dass später eine Übersetzungseinheit kompiliert wird in eine Objektdatei. Der Rückgabewert der Funktion enthält insbesondere die notwendigen Informationen um die so kompilierte Objektdatei wiederfinden zu können. Es genügt daher im Folgenden den Rückgabewert eines Aufrufs in eine neue Variable zu speichern und den Wert dieser Variable dann an andere eingebaute Funktionen geeignet zu übergeben um zu erreichen, dass beim linken einer ausführbaren Binärdatei die erzeugte Objektdatei mit gelinkt wird.
Positionale Parameter sind ein string der als Name der library verwendet wird und die Namen von beliebig vielen Quellcode-Dateien (bei uns i.d.R. Dateiendung `.cpp`)
Es werden insb. die selben Schlüsselwort-Parameter (⟨KWArgs⟩) unterstützt wie bei `executable`.

**`subproject`** Wertet die Meson-Spezifikation des als string-Parameter gegebenen Unterprojekt-Namens aus. Es wird erwartet, dass das Unterprojekt sich in einem Ordner mit dem gleichen Namen wie der Inhalt des string-Parameters unterhalb des Unterordners `subprojects` des Projekts befindet.
Es wird ein Wert zurückgegeben auf den sich insb. die folgende Methode anwenden lassen:

**`get_variable`** Nimmt als Parameter den Namen einer Variablen die in der Meson-Spezifikation des Unterprojektes definiert wurde und liefert ihren Wert zurück.

**`test`** Fügt eine ausführbare Binärdatei der Menge von Tests des Projektes hinzu.
Positionale Parameter sind ein Name als string und der Rückgabewert eines Aufrufs von `executable`.

**`dependency`** Sucht auf betriebsysstemabhängige Weise nach der als string-Parameter übergebenen externen Abhängigkeit und liefert eine Referenz darauf als Rückgabewert.

**`declare_dependency`** Liefert einen Wert zurück, der sich zum Übergeben an den `dependencies`-Parameter von `executable` und `library` eignet. Relevante Schlüsselwort-Parameter sind:

**`link_with`** Nimmt einen einzelne oder ein Array von Rückgabewerten von Aufrufen der Funktion

`library` und sorgt dafür, dass bei Verwendung des Rückgabewertes von `declare_dependency` in z.B. `executable` die gegebenen Werte behandelt werden als wären auch sie direkt mit `link_with` an den Aufruf von `executable` übergeben worden.

**`include_directories`** Nimmt einen einzelnen oder ein Array von strings die Unterverzeichnisse des Projekts bezeichnen. Bei Verwendung des Rückgabewertes von `declare_dependency` in z.B. `executabe` werden dann alle hier angegeben Unterverzeichnisse geeignet an den compiler übergeben, sodass deren Inhalt mit durchsucht wird, wann immer der Präprozessor bei Auflösung einer `#include`-Direktive (mit spitzen Klammern) nach einer Datei sucht.

**`executable`** Jeder Aufruf der Funktion `executable` führt dazu, dass spëter eine ausführbare Binärdatei erzeugt wird. Positionale Parameter sind ein string der als Name der ausführbaren Binärdatei verwendet wird (bei uns bisher oft `a.out`, üblicher sind jedoch sprechendere Namen) und die Namen von beliebig vielen Quellcode-Dateien (`.cpp`) die die Übersetzungseinheit bilden aus der die ausführbare Binärdatei kompiliert werden soll. Relevante Schlüsselwort-Parameter sind:

**`link_with`** Nimmt einen einzelne oder ein Array von Rückgabewerten von Aufrufen der Funktion `library`

**`dependencies`** Nimmt einen einzelne oder ein Array von Rückgabewerten von Aufrufen der Funktionen `dependency` oder `declare_dependency`

*Beispiel.* Wir formulieren eine `meson.build`-Datei für das kleine Software-Projekt.

`meson_ld/meson.build`

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

Meson platziert seine generierte `ninja.build`-Datei und die build-Produkte in einem Unterverzeichnis des Projektverzeichnisses. Meson bezeichnet dieses Unterverzeichnis als das *builddir*. Wir nennen es per Konvention `build`.

Folgende `meson`-Befehle sind für unsere Zwecke relevant und können im Projektverzeichnis verwendet werden:

**`meson setup build`** Wertet die `meson.build`-Datei initial aus und erstellt das builddir mit Namen `build`

**`ninja -C build`** Wertet die `meson.build`-Datei erneut aus falls sie sich verändert hat und aktualisiert die resultierende `ninja.build`-Datei. Danach werden auch die build-Produkte und alle ihre Abhängigkeiten neu gebaut, soweit erforderlich. Build-Produkte entstehen hierbei insb. durch alle Aufrufe der eingebauten Funktionen `executable` und `library`.

**`meson test -C build`** Führt alle an Aufrufe der eingebauten Funktion `test` übergebenen ausführbaren Binärdateien aus nachdem sie, soweit erforderlich, neu gebaut wurden.

### Meson WrapDB

Die Meson community verwaltet eine Sammlung von quelloffenen Projekten die entweder standardmäßig selbst mit Meson kompiliert werden oder für die die Meson community eigene Meson-Spezifikationen geschrieben hat. Aus dieser, *WrapDB* genannten, Sammlung können einfach von der Kommandozeile Projekte als subproject des eigenen Projektes hinzugefügt werden. Man spricht von *vendoring*.

Vor der ersten Installation eines subprojects aus der WrapDB muss zunächst im Projektverzeichnis das Unterverzeichnis `subprojects` manuell angelegt werden.

Danach kann durch einen Befehl der Form "`meson`" "`wrap`" "`install`" ⟨ProjektName⟩, also z.B. `meson wrap install gtest`, das Projekt mit Namen ⟨ProjektName⟩ in der WrapDB im Internet nachgeschlagen und unterhalb von subprojects heruntergeladen werden.

Danach kann es in der Meson-Spezifikation des eigenen Projektes mit `subproject` verwendet werden.