C/C++ Materialpaket (Level C)
01b_TOOL – Toolchain

Prof. Dr. Carsten Link

Zusammenfassung

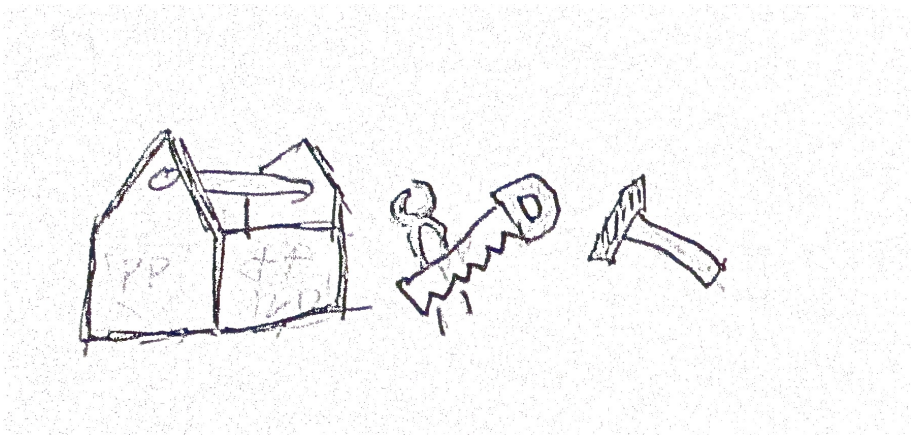


Abbildung 1: Verwendung der C++-Werkzeuge

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können die für die C/C++-Programmierung übliche Toolchain bedienen.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- Programme, welche aus mehreren Quelltextdateien bestehen, übersetzen lassen
- ein C/C++-Programm, welches aus mehreren Quelltextdateien besteht, erstellen
- dieses Programm modifizieren, so dass sich ein anderes Verhalten ergibt
- die nötigen Werkzeuge und deren Aufgaben benennen

2 Konzepte

2.1 Motivation

Betrachten Sie kurz die folgende *Rube Goldberg machine*[2]:

Aufgrund Ihrer Erfahrung und Ihres Wissens über physikalische Zusammenhänge und Vorgänge können Sie sich erschließen, wie diese Maschine arbeitet. Hätten Sie keine physikalischen Erfahrungen, so wäre es Ihnen nicht möglich zu ergründen, was in der Abbildung passiert.

In diesem Materialpaket werden Komponenten vorgestellt, die en Komponenten eine Rube-Goldberg-Machine ähneln: jedes löst eine Teilaufgabe und zusammengenommen wird ein Ziel erreicht. Auch wenn Sie im Verlauf der Veranstaltung

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

[2] en.wikipedia: Rube Goldberg is best known for his popular cartoons depicting complicated gadgets performing simple tasks in indirect, convoluted ways. The cartoons led to the expression "Rube Goldberg machines" to describe similar gadgets and processes.

[3] Quelle: https://media.rubegoldberg.com/site/wp-content/uploads/2017/11/2018-RB.INTRO_.November.pdf

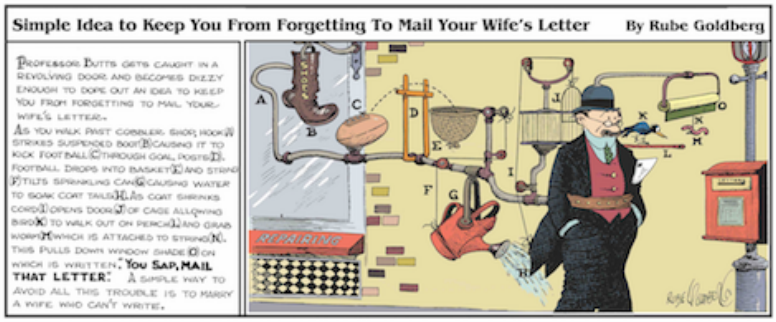


Abbildung 2: Eine Rube Goldberg Maschine[3]

nicht mit jeder Komponente und den entsprechenden Zwischenerzeugnissen in Berührung kommen, sollten Sie diese kennen. Insbesondere wenn Fehler auftreten und sie deren Ursache identifizieren müssen.

2.2 Übersetzen eines einfachen C-Programmes

Das nachfolgende Programm verfügt über zwei Funktionen: `line()` und `main()`. Letzere wird beim späteren Programmstart automatisch ausgeführt.

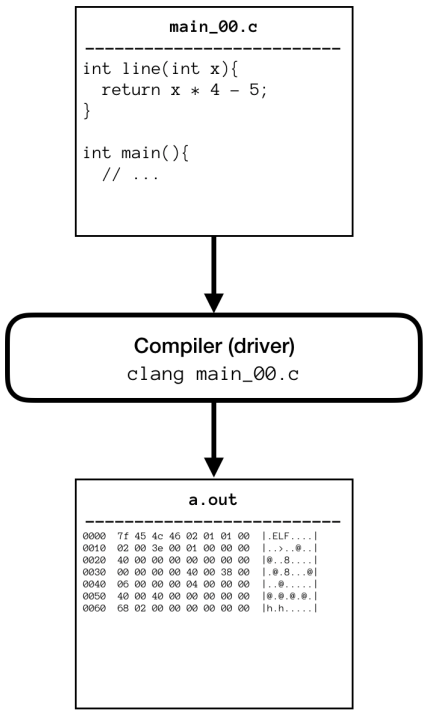
```
1  int line(int x){
2    return x * 4 - 5;
3  }
4
5  int main(){
6    int y = 0;
7    int x = 0;
8    while ( x < 10 ){
9      y = line(x);
10     // TODO: print values
11     x++;
12   }
13   return 0;
14 }
```

Oben ist der Inhalt der Datei `main_00.c` zu sehen. Diese Datei kann nun mit dem Kommando `clang main_00.c` zu einem ausführbaren Programm übersetzt werden:

```
bash$ ls
main_00.c
bash$ clang main_00.c
bash$ ls
a.out  main_00.c
```

Anhand der beiden `ls`-Ausgaben ist zu sehen, dass die Datei `a.out` neu entstanden ist. Sie wurde von `clang`- erzeugt und ist ausführbar. Da in dem Programm keine Ausgabe enthalten ist, ist beim Start mit `./a.out` nichts zu sehen. Da wir uns mit der Toolchain befassen, soll dies keine Rolle spielen.

In der nachfolgenden Abbildung ist zu sehen, wie aus dem Quellcode eine ausführbare Datei wird:



Im obigen Beispiel sind viele Schritte verborgen, da `clang` als Compiler Driver fungiert hat und mehrere Komponenten der Toolchain angetrieben hat[4]. Diese Komponenten werden in den nächsten Unterkapiteln vorgestellt.

2.3 Toolchain-Komponenten

Die Werkzeugkette für Programmierer von C/C++-Programmen und Bibliotheken besteht aus diesen Komponenten:

- Editor
- Preprocessor (Präprozessor)
- Compiler
- Assembler

[4]Umgangssprachlich wird hier oft von Compiler gesprochen. Dies ist aber nur im Sinne von *pars pro toto* richtig.

- Linker (Binder)
- Loader
- Debugger
- Compiler-Treiber
- Build Tool
- Integrated Development Environment (IDE)

In den nachfolgenden Kapiteln werden die wichtigsten dieser Komponenten vorgestellt.

2.4 Toolchain-Komponente Editor

Wichtige Optionen der Editorstufe:

- Zeilennummern
- Übersicht über geöffnete Dateien
- Tabs vs. Spaces
- Dark vs. light mode

Häufige Fehler der Editorstufe:

- vor dem Neucompilieren wurde nicht abgespeichert
- Copy-Paste-Fehler: das Copy-Paste-Modify innerhalb des eigenen Quelltextes wurde nicht vollständig umgesetzt
- andere Kopie einer Datei im Fenster
- Ungültige Zeichen wurden aus PDF oder Webseiten kopiert. Oft sind diese Zeichen unsichtbar
- Bezeichner und Kommentare in einer anderen Sprache als der *Lingua Franca*
- Unübersichtliche Formatierung (Einrückung)
- Verwendung ungeeigneter Schriftarten (Unterscheidbarkeit von `iItl1O0`) oder Farbschemata

2.5 Toolchain-Komponente Präprozessor

Der Präprozessor läßt sich mit `clang -E` einzeln aufrufen. Ein Beispielprogramm dazu (`main_10.c`):

```
1  #define SLOPE 7
2  #define OFFSET -3
3  #define MIN 0
4  #define MAX 20
5
6  int line(int x){
7    return x * SLOPE + OFFSET;
8  }
9
10 int main(){
```

```
11    int y = 0;
12    int x = MIN;
13    while (x<MAX){
14      y = line(x);
15      // TODO: print values
16      x++;
17    }
18    return 0;
19  }
```

Der Präprozessor erzeugt die Datei `main_10.i` (Aufruf: `clang -o main_10.i -E main_10.c`):

```
1   int line(int x){
2     return x * 7 + -3;
3   }
4
5   int main(){
6     int y = 0;
7     int x = 0;
8     while (x<20){
9       y = line(x);
10
11      x++;
12    }
13    return 0;
14  }
```

Mit `diff --color=always main_10.c main_10.i` (oder `meld main_10.c main_10.i`) lassen sich die Unterschiede zwischen den beiden Datein aufzeigen:

```
1   7c14
2   <   return x * SLOPE + OFFSET;
3   ---
4   >   return x * 7 + -3;
5   12,13c19,20
6   <   int x = MIN;
7   <   while (x<MAX){
8   ---
9   >   int x = 0;
10  >   while (x<20){
```

Mit `diff -y main_10.c main_10.i` (oder `meld main_10.c main_10.i`) lassen sich die Unterschiede zwischen den beiden Datein nebeneinander aufzeigen:

```
1   int line(int x){                          int line(int x){
2     return x * SLOPE + OFFSET;            |   return x * 7 + -3;
3   }                                         }
```

```
4
5  int main(){                                int main(){
6    int y = 0;                                 int y = 0;
7    int x = MIN;                         |     int x = 0;
8    while (x<MAX){                       |     while (x<20){
9      y = line(x);                               y = line(x);
10     // TODO: print values                |
11     x++;                                       x++;
12   }                                          }
13   return 0;                                  return 0;
14 }                                        }
```

Oben ist zu sehen, dass der Präprozessor die mit `#define` deklarierten Makros (`MIN`, `MAX`, `SLOPE`, `OFFSET`) durch ihre Werte ersetzt hat. Das ist hilfreich, da dann Konstante Werte an einer zentralen Stelle im Quellcode eingestellt werden können. Weitere Präprozessor-Makros helfen, den Code eines größeren Programmes in mehrere Dateien aufzuteilen (`#include`) oder unterschiedlich Varianten des Programmes zu erzeugen (`#ifdef`).

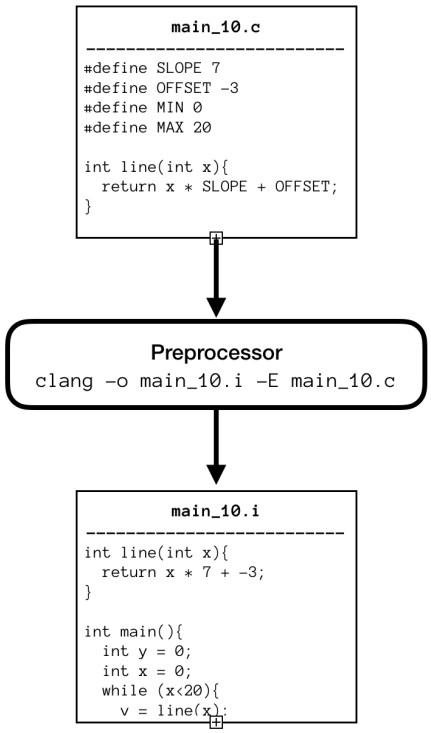
Wichtige Optionen der Präprozessorstufe:

- Suchpfade für Headerdateien: `-I ../path/to/headers` oder `-I path1/:path2/`
- Makrodefinitionen: `-DDEBUG` oder `-DVERSION=3`

Häufige Fehler der Präprozessorstufe:

- mehrfache `#include` derselben Datei
- Datei von `#include` ist nicht zu finden (falsch geschrieben, in anderem Verzeichnis)
- ungewünschte Ersetzungen von Makros (`#define`)

In der nachfolgenden Abbildung ist Präprozessorstufe zu sehen:



2.6 Toolchain-Komponente Compiler

Der Compiler ist das Kernstück der Toolchain.

Hier ein Auszug aus der oben vollständig gegebenen C-Datei `main_10.i`, die vom Präprozessor erzeugt wurde:

```
1 int line(int x){
2   return x * 3 + -3;
3 }
```

```
clang -S main_10.i
```

Hier ein Auszug aus der sich ergebenden Assembly-Datei `main_10.s`, in der der Assembly-Code der Funktion `line()` zu sehen ist:

```
1 line:                                   # @line
2         .cfi_startproc
3 # %bb.0:
4         pushq   %rbp
5         .cfi_def_cfa_offset 16
6         .cfi_offset %rbp, -16
7         movq    %rsp, %rbp
```

```
8         .cfi_def_cfa_register %rbp
9         movl    %edi, -4(%rbp)
10        imull   $7, -4(%rbp), %edi
11        addl    $-3, %edi
12        movl    %edi, %eax
13        popq    %rbp
14        .cfi_def_cfa %rsp, 8
15        retq
16 .Lfunc_end0:
17        .size   line, .Lfunc_end0-line
18        .cfi_endproc
```

Bei Assembly-Code handelt es sich um menschenlesbaren Maschinencode. Für jemanden, der nicht Assembly programmiert, sind oben jedoch einzig die beiden Konstanten `7` und `-3` erkennbar.

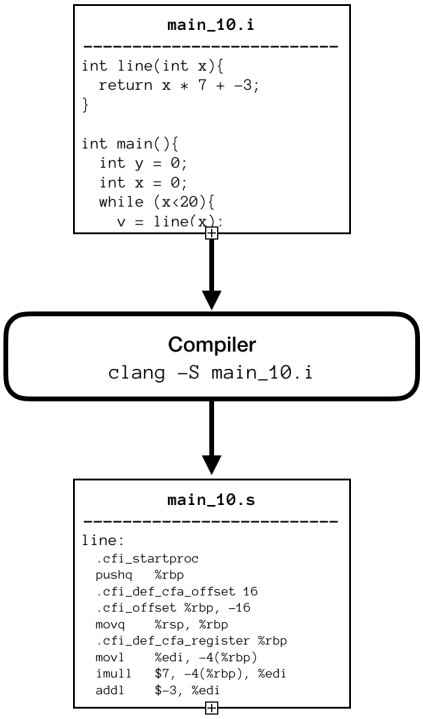
Wichtige Optionen der Compilerstufe:

- Auswahl eines der unterstützten C++-Sprachstandards: `-std=c++17`
- Aktivierung von Warnings: `-Wall`, `-Werror`, `-Wextra`
- Auswahl einer Optimierungsstufe: `-O0` bis `-O3`
- Aktivierung von Debug-Informationen: `-g`

Häufige Fehler der Compilerstufe:

- Syntaxfehler
- Typfehler
- …

In der nachfolgenden Abbildung ist Compilerstufe zu sehen:



2.7 Toolchain-Komponente Assembler

Der Assembler übersetzt mit `clang -c main_10.s` den menschenlesbaren Maschinencode in CPU-lesbaren Maschinencode. Hier ein Auszug aus der sich ergebenden Datei (`hexdump -vC main_10.o`).

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  40 02 00 00 00 00 00 00  |........@.......|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 09 00 01 00  |....@.....@.....|
00000040  55 48 89 e5 89 7d fc 6b  7d fc 07 83 c7 fd 89 f8  |UH...}.k}.......|
00000050  5d c3 66 2e 0f 1f 84 00  00 00 00 00 0f 1f 40 00  |].f...........@.|
00000060  55 48 89 e5 48 83 ec 10  c7 45 fc 00 00 00 00 c7  |UH..H....E......|
00000070  45 f8 00 00 00 00 c7 45  f4 00 00 00 00 83 7d f4  |E......E......}.|
00000080  14 7d 16 8b 7d f4 e8 b5  ff ff ff 89 45 f8 8b 45  |.}..}.......E..E|
00000090  f4 83 c0 01 89 45 f4 eb  e4 31 c0 48 83 c4 10 5d  |.....E...1.H...]|
```

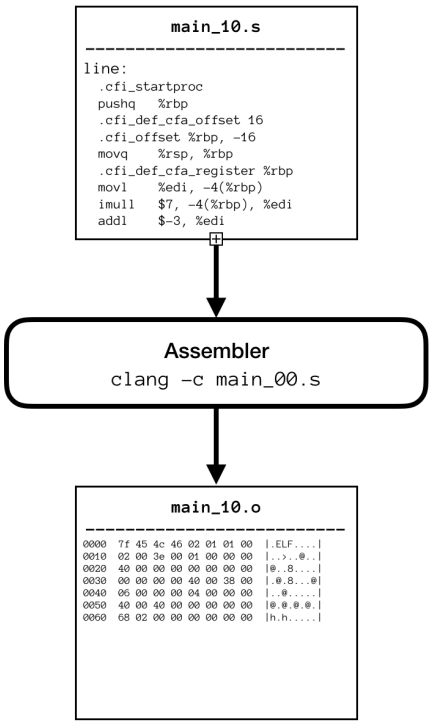
Es lässt sich nun nichts mehr erkennen, das auf das ursprüngliche C-Programm hindeutet. Allerdings lassen sich aus Objektdateien (`.o`) mit dem Kommando `objdump` wieder Informationen entlocken (Auszug von `objdump -d main_10.o`):

```
main_10.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <line>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   6b 7d fc 07             imul   $0x7,-0x4(%rbp),%edi
   b:   83 c7 fd                add    $0xfffffffd,%edi
   e:   89 f8                   mov    %edi,%eax
  10:   5d                      pop    %rbp
  11:   c3                      retq
  12:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  19:   00 00 00
  1c:   0f 1f 40 00             nopl   0x0(%rax)

0000000000000020 <main>:
  20:   55                      push   %rbp
  // ...
```

Häufige Fehler der Assemblerstufe:

- keine, da der Assembler meist korrekt arbeitet und mögliche Fehler bereits vom Compiler abgefangen werden

In der nachfolgenden Abbildung ist Assemblerstufe zu sehen:



2.8 Toolchain-Komponente Linker

Werden dem Compiler Driver `clang` eine oder mehrere Objektdatein übergeben, so wird der Linker veranlasst, diese zu einer ausführbaren Datei zu verbinden. In unserem Beispiel `clang main_10.o`.

Da die erzeugte Datei wieder binär ist, werden mit `objdump -d a.out` Informationen extrahiert (ein Auszug):

```
1  a.out:     file format elf64-x86-64
2
3  Disassembly of section .text:
4
5  0000000000401020 <_start>:
6    401020:   31 ed                    xor    %ebp,%ebp
7    // ...
8
9  0000000000401110 <line>:
10   401110:   55                       push   %rbp
11   401111:   48 89 e5                 mov    %rsp,%rbp
12   401114:   89 7d fc                 mov    %edi,-0x4(%rbp)
13   401117:   6b 7d fc 07              imul   $0x7,-0x4(%rbp),%edi
```

```
14   40111b:   83 c7 fd                add    $0xfffffffd,%edi
15   40111e:   89 f8                   mov    %edi,%eax
16   401120:   5d                      pop    %rbp
17   401121:   c3                      retq
18   401122:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
19   401129:   00 00 00
20   40112c:   0f 1f 40 00             nopl   0x0(%rax)
21
22 0000000000401130 <main>:
23   401130:   55                      push   %rbp
24   // ...
25
26 00000000004011e4 <_fini>:
27   4011e4:   48 83 ec 08             sub    $0x8,%rsp
28   // ...
```

Es ist zu sehen, dass zu der Funktion `line` noch weitere Funktionen hinzugekommen sind: `_start` und `_fini`. Diese sorgen für die Initialisierung beim Programmstart, den Aufruf von `main()` und das Aufräumen beim Programmende.

Der Vorteil, der sich ergibt, ausführbare Programme aus `.o`-Dateien zusammen zu binden ist der, dass die Objektdateien aus Quellcode in verschiedenen Programmiersprachen stammen können. So kann ein C++-Programm Funktionen nutzen, welche in Assembly, C, Pascal, Fortran oder einer anderen Programmiersprache programmiert wurden. Um mehrere `.o`-Dateien einfacher handhaben zu können, lassen sich diese in einer Archiv-Datei zusammenfassen (`.a` oder `.lib`[5]).

Wichtige Optionen der Linkerstufe:

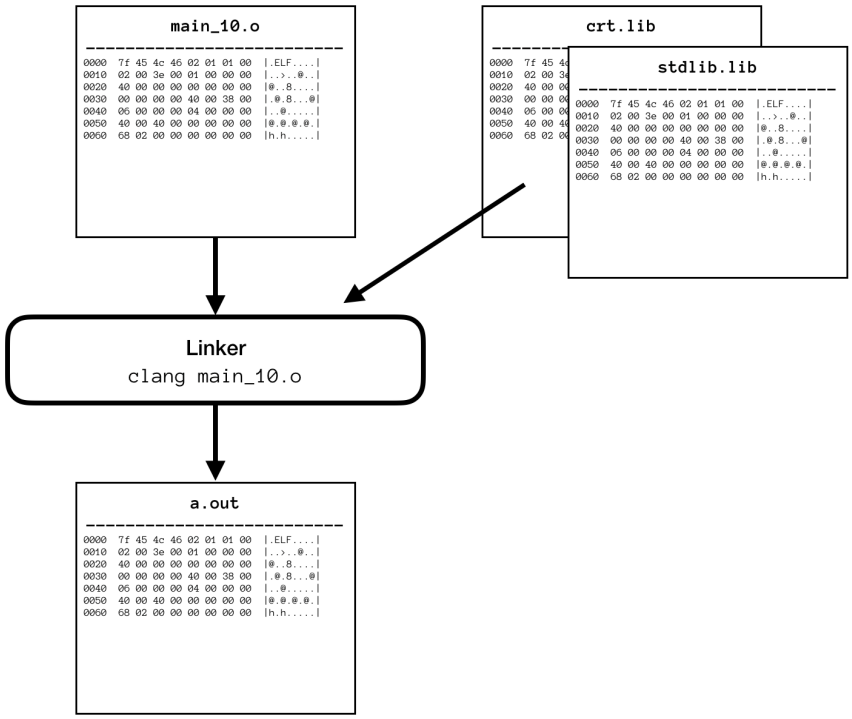
- benötigte Bibliotheksdateien: `-l library`
- Suchpfade für Bibliotheksdateien: `-L path1/:path2/`

Häufige Fehler der Linkerstufe:

- `undefined reference`, `unresolved external`, `unresolved reference` oder `undefined symbol` (synonym): eine `.o`-Datei oder `.a`-Datei wurde nicht angegeben
- die übergebenen `.o`-Datei passen nicht zusammen, da nicht alle von Änderungen betroffenen Quelldateien neu übersezt wurden
- `multiple definitions` oder `duplicate symbol`: eine übersetzte Funktion wurde in mehreren Objektdateien gefunden
- es wird versucht, verschiedene Versionsstände zu linken (z.B. eine von mehreren `.cpp`-Dateien wurde nicht neu übersetzt)

[5]Die genauen Verzeichnispfade, Dateinamen und -endungen von Systemobjektdateien und Systembibliotheksdateien sind abhängig von Compiler (clang, gcc, msvc, …), Betriebssystemart (Windows, UNIX, …), Variante (Debian Linux, Fedora Linux, FreeBSD, OpenBSD, …) und -version.

In der nachfolgenden Abbildung ist Linkerstufe zu sehen:



2.9 Toolchain-Komponente Archiver (Exkurs)

Es gibt Situationen, in denen es sinnvoll ist, Softwarebibliotheken ohne Quellcode auszuliefern. Dies kann umgesetzt werden, indem eine Library-Datei (`.a` oder `.lib`, erstellt mit `ar`) ausgeliefert wird, die alle zur Bibliothek gehörenden Objektdateien enthält. Zusätzlich werden alle Header-Dateien, die die in der Library-Datei enthaltenen `.obj`-Dateien beschreiben, ausgeliefert.

Die von jedem Programm verwendeten System- und Standardbilbiotheken werden auf oben beschriebene Weise vertrieben. So genügt in einem einfachen C-Programm `#include <stdio.h>`, um `printf()` nutzen zu können; das Compilieren der C-Datei, die die `printf()`-Implementierung enthält, entfällt. Weitere Maßnahmen sind nicht erforderlich, da der Compiler Driver dem Linker die Library-Datei zur Standardbibliothek implizit übergibt. Dies kann mit der Option `-v` sichtbar gemacht werden (z.B. `clang -v main.o`).

2.10 Toolchain-Komponente Loader

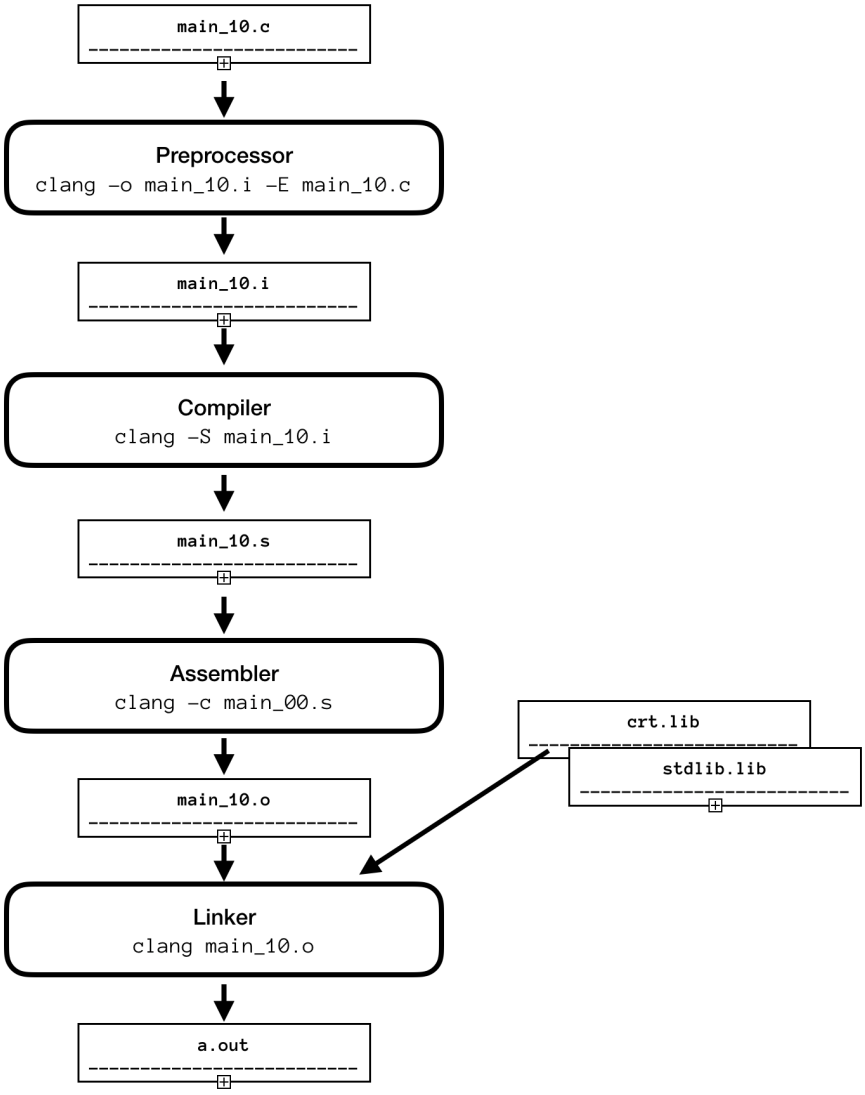
Der Loader ist der Teil des Betriebssystems, welcher Programme in den Arbeitsspeicher lädt und ausführen lässt. Im Zusammenspiel mit Standardbibliotheken wird dann die Funktion `main()` aufgerufen.

Häufige Fehler der Loaderstufe:

- keine, da der Loader meist korrekt arbeitet und mögliche Fehler bereits von vorherigen Stufen abgefangen werden
- (Level C) dynamische Bibliothekn können nicht gefunden werden

2.11 Toolchain-Komponenten für eine Übersetzungseinheit

Die folgende Abbildung zeigt alle Komponenten der Toolchain im Zusammenspiel, um aus der Quelldatei `main_10.c` die ausführbare Datei `a.out` zu machen:



2.12 Toolchain-Komponenten für mehrere Übersetzungseinheiten

Ein C-Programm besteht im wesentlichen aus Deklarationen und Definitionen (für Variablen, Funktionen und ggf. Typen), welche vom Programmierer auf Quelltext-Dateien (`.c`) und Include-Dateien (`.h`) verteilt werden. Meist werden noch Datenstrukturen und Funktionen des Betriebssystems und von Drittanbietern verwendet, welche sich in Include-Dateien (`.h`) und Bibliotheksdateien (Libraries, `.a`-Dateien) befinden.

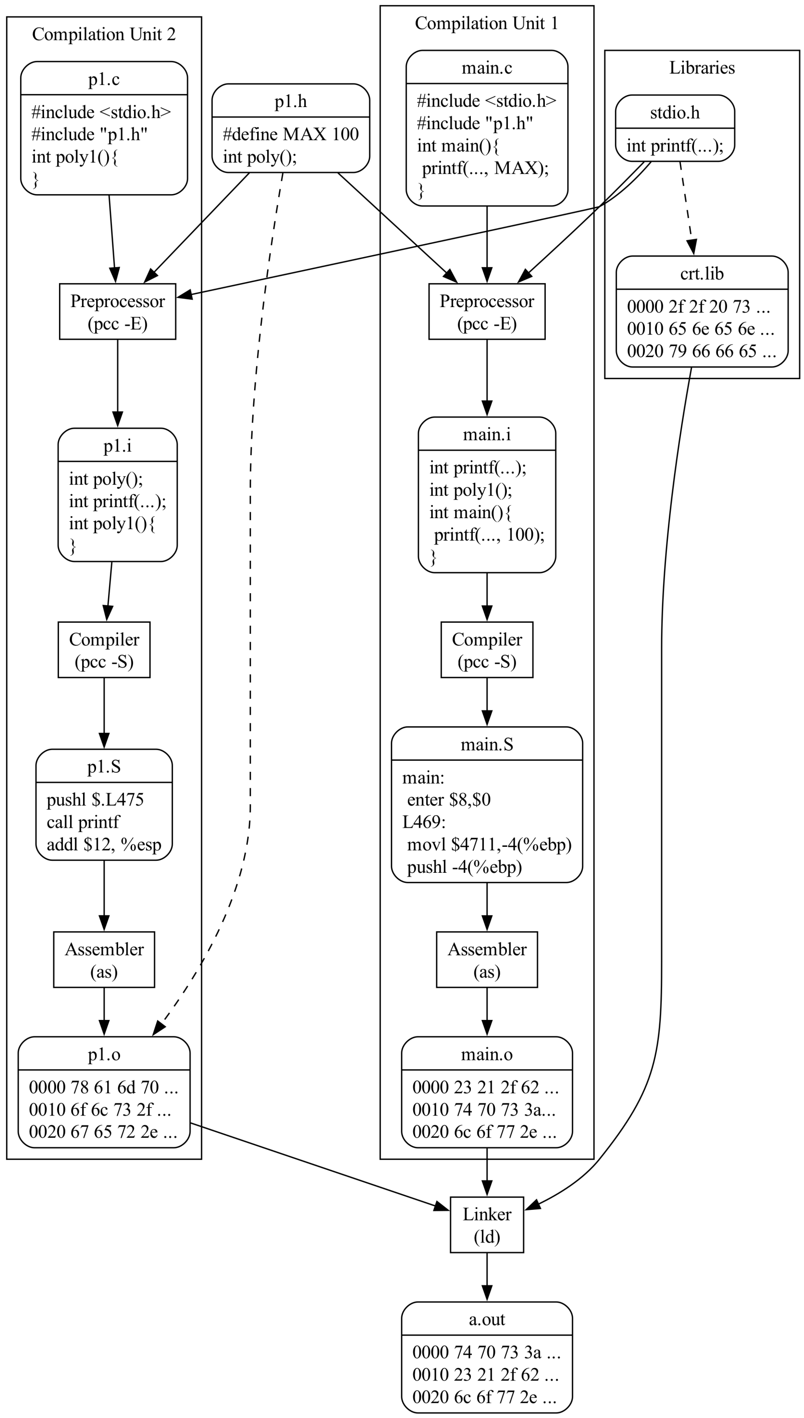


Abbildung 3: Die wichtigsten Werkzeuge der Werkzeugkette. Im Pseudocode ist zu sehen, welche Art von Transformation die einzelnen Werkzeuge vornehmen. Durchgezogene Pfeillinien stehen für Lese- und Schreibvorgänge; gestrichelte bedeuten, dass die `.h`-Datei eine Schnittstellenbeschreibung zur dazugehörigen `.o`-Datei ist.

In Abbildung 3 ist zu sehen, wie die einzelnen Werkzeuge aus Eingabedateien Ausgabedateien erstellen[6]:

- Der Editor erlaubt dem Programmierer das verfassen von Quelltext-Dateien (`.c`) und Include-Dateien (`.h`)
- Der Präprozessor behandelt Präprozessordirektiven. So wird beispielsweise die Stelle der Direktive `#include filename` durch den Inhalt der Datei `filename` ersetzt. Die Direktive `#define BUFSIZE 10` sorgt dafür, dass im darauffolgenden Quelltext die Zeichenfolge `BUFSIZE` durch `10` ersetzt wird. Die erstellte Datei hat die Endung `.i`
- Der Compiler übersetzt den vom Präprozessor behandelten Quelltext in Assemblercode oder Maschinencode (Dateiendungen `.s` oder `.o`)
- Der Assembler übersetzt Assemblercode (Assembly, Endung `.s`) in Maschinencode (Dateiendung `.o`)
- Der Linker setzt mehrere Dateien mit Maschinencode (`*.o`) zu einer ausführbaren Datei zusammen (`a.out`)
- Schließlich lädt das Betriebssystem (loader) das Programm in den Arbeitsspeicher, versorgt es mit Speicherbereichen und lässt die CPU in die Funktion `main` springen

[6] In der Abbildung wurde der Portable C Compiler pcc verwendet.

2.13 Toolchain-Komponenten in den zukünftigen Materialpaketen

In diesem Materialpaket werden die wichtigsten Werkzeuge genau betrachtet, damit in der Zukunft Fehlermeldungen besser einzuordnen sind und eine Vorstellung davon existiert, wie der Quellcode vom Editor bis zur Ausführung kommt.

In den zukünftigen Materialpaketen brauchen die Werkzeuge nicht einzeln aufgerufen werden. Es sollen C++-Quelldateien mit der Option `-c` zu Objektdateien übersetzt werden. Anschließend werden alle Objektdateien des Programms zusammengelinkt (über `clang++` als driver):

```
bash$ clang++ -c main.cpp
bash$ clang++ -c one.cpp
bash$ clang++ -c two.cpp
bash$ clang++ main.o one.o two.o
bash$ ./a.out
C++ rocks!
```

Hierbei ist zu beachten:

- die Quelldateien sollen ohne Warnings übersetzen
- bei Fehlermeldungen sollte die oberste Fehlermeldung zuerst bearbeitet werden (da Fehler oft künstliche Folgefehler nach sich ziehen)

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Erstellen Sie ein Programm, das aus den Dateien `main.c`, `polynom1.h`, `polynom1.c` besteht. In `polynom1.c` wird eine Funktion definiert, welche in `polynom1.h` folgende Deklaration hat:

```
1 int polynom1(int x); // returns y = f(x) for value x
2                      // f(x) is a polynominal function
3                      // e.g. y = (x-1)*(x-2)*(x-3)
```

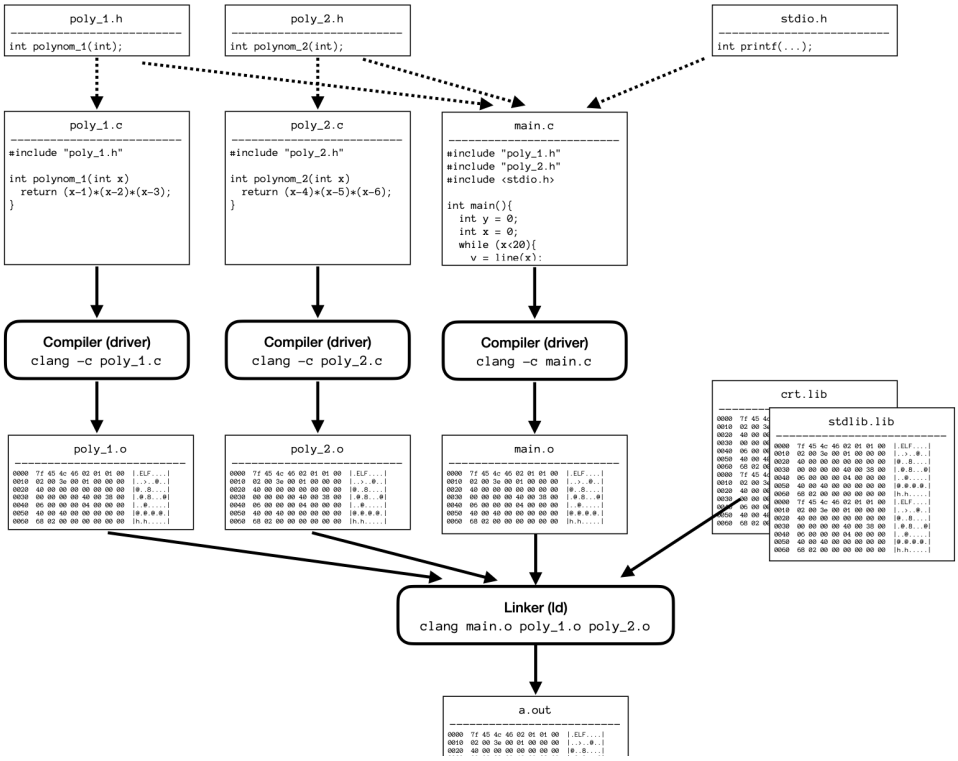
In der `main()`-Funktion wird die Funktion `polynom1()` für die Werte `0..20` aufgerufen und dort die Werte mit `printf("x=%d y=%d\n", x, y);` ausgegeben.

Lassen Sie die Quelldateien von `clang` getrennt übersetzen (siehe Compileroption `-c`) und lassen das ausführbare Programm aus den Objektdateien zusam-

menbinden (`clang polynom1.o main.o` ist einfacher, als selbst den Linker `ld` aufzurufen; siehe `clang -v *.o`). Hierzu können Sie ein `bash`-script verwenden[7], um sich Tipparbeit zu ersparen. Beachten Sie die Hinweise in `0README.txt` in ihrem Quellvorlagenverzeichnis.

Hinweis: Wir erstellen in diesem Materialpaket `C`-Programme, übersetzen diese mit `clang` und machen Ausgaben mit `printf()` aus `stdio.h` (`#include <stdio.h>`; siehe `main_printf.c`). In allen anderen Materialpaket erstellen wir `C++`-Programme, übersetzen diese mit `clang++` und machen Ausgaben mit `println()` aus `println.hpp` (`#include "println.hpp"`).

In der nachfolgenden Abbildung sind die Datein des Grundgerüsts und der Modifikationen zu sehen, die von der Toolchain verarbeitet werden:



3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende

[7] Mit `chmod 755 <file name>` können Sie das Script ausführbar machen

nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Hinweis: Im Testat sollen Sie Kommandozeile (`bash`-Shell im Terminal) und `pluma` verwenden.

Modifikationen:

1. Bauen Sie eine zweite C-Funktion `polynom2()` mit einer anderen Polynomfunktion ein und verwenden diese in `main()`, wobei sie individuelle `.cpp` / `.c` und `.h`-Dateien verwenden
2. Erstellen Sie eine Funktion `poly_all()` (in separaten Dateien), welche das Produkt der bisherigen Funktionen zurückliefert
3. Lassen Sie für `main.c` den Präprozessor, den Compiler und den Assembler einzeln ablaufen und zeigen die jeweiligen Ergebnisse. Sie dürfen einen Spickzettel für die `clang`-Optionen verwenden, da Sie diese nicht auswendig lernen sollen.
4. Provozieren Sie den Fehler `undeclared identifier` bzgl. einer Variable. Hinweis: verwenden Sie eine neue `.cpp`-Datei und compilieren diese mit `clang++`
5. Provozieren Sie den Fehler `undefined reference` bzgl. einer Funktion. Hinweis: verwenden Sie eine neue `.cpp`-Datei und compilieren diese mit `clang++`
6. Provozieren Sie den Fehler `multiple definitions` bzgl. einer Funktion
7. (Level C) erstellen Sie `preprocess.sh`, `compile.sh`, `assemble.sh`, `link.sh` und füllen diese mit den jeweiligen Tool-Aufrufen. Kopieren Sie jeweils `tool_c_to_out.sh` als Vorlage.

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels “Konzepte”, der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Schreiben Sie kurze Glossar-Einträge für alle an der Übersetzung/Erstellung eines Programms beteiligten Komponenten bzw. Phasen.
2. Ist es möglich, von einer Quelldatei aus eine Funktion aufzurufen, welche in einer anderen Quelldatei definiert ist? Begründung!
3. Welche Aufgaben übernimmt der Compiler?

4. Was ist der Unterschied zwischen einer ausführbaren Datei (z.B. `a.out`) und einer Objektdatei (`.obj`)?
5. Was ist der Unterschied zwischen einer Bibliotheksdatei (`.a`) und einer ausführbaren Datei (z.B. `a.out`)?
6. Welche Art von Fehler von welchem Werkzeug ergibt sich, wenn `#include "polynom1.h"` in `main.c` vorhanden ist und `int polynom1(int)` in `polynom1.h` deklariert ist, aber `polynom1.c` keine Definition zu dieser Deklaration enthält?
7. Nehmen Sie an, ein Programm `a.out` besteht aus den Dateien `main.c`, `f.h` und `f.c`. Geben Sie an, welche Werkzeuge vom Compiler Driver aufgerufen werden, wenn Sie die folgenden Optionen angeben:
   - `clang main.c f.c`
   - `clang main.o f.o`
8. Welche Teile der Toolchain sind beteiligt bei `clang -c main.c` und welche Datei entsteht?
9. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.
10. (Level C) Welche Folgen hat es, wenn in `main.c` die Datei `polynom1.c` per `#include` eingebunden wird, statt `polynom1.h`?
11. (Level C) Betrachten Sie die Dateien im Verzeichnis `01_TOOL/implDecl` und experimentieren Sie damit.
    - welches Fehlverhalten zeigt sich?
    - wie lässt es sich konkret korrigieren?
    - wie lassen sich derartige Fehler im Allgemeinen verhindern?
    - zeigen Sie die Unterschiede zwischen der fehlerhaften und der korrekten Variante von mittels `objdump` auf

4 Nützliche Links

- C-HowTo, E. Fischer, http://www.c-howto.de/
- Sektion 3 der Linux Manual Pages (z. B. `man 3 printf`)
- Clang Man Page online[8] oder per `man clang`
- StackOverflow, *What is compiler, linker, loader?*, https://stackoverflow.com/questions/3996651/what-is-compiler-linker-loader
- TENOUK'S BUFFER OVERFLOW 3 An Assembly Language[9]
- Tenouk's C & C++ Site[10]
- Wolfgang Schröder, C++-Tutor[11]

[8] http://clang.llvm.org/docs/CommandGuide/clang.html
[9] http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1b.html
[10] http://www.tenouk.com/Sitemap.html
[11] https://www.cpp-tutor.de/

- Herbert Schildt, C The Complete Reference online[12] (Vorsicht: enthält Fehler)
- Bjarne Stroustrup's FAQ http://www.stroustrup.com/bs_faq.html

5 Literatur

- [TENCAL] Tenouk: COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY[13]
- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition

[12] https://docs.google.com/file/d/0B3OzFFMgEP0tU3RVcmh2Wm5ZUWs/edit?pref=2&pli=1

[13] http://www.tenouk.com/ModuleW.html