Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 1

Abgabe: Montag, 26.04.2021, 9:00 Uhr morgens

In diesem Übungsblatt sollen Sie sich mit den für die Vorlesung benötigten Tools und unserer Lehrplatform vertraut machen. Ein Großteil dieser Tools beschränkt sich nicht nur auf die Programmiersprache C, sondern gehört zum Allgemeinwissen und Alltag in der Informatik und wird sich auch für andere Programmiersprachen als nützlich erweisen. Die C Programmierung besteht in diesem Übungsblatt lediglich aus einem *Hello World*-Programm. Für die **korrekte** Abgabe dieses Programms, gibt es 14 Punkte und für Ihre Erfahrungen zwei weitere Punkte.

Wir verwenden in dieser Vorlesung die folgenden Tools:

- **Shell.** Der Interpreter, der bei Linux (`bash`) und MacOS (`zsh`) standardmäßig installiert ist und automatisch in jedem Terminal läuft. Dieses Tool benötigen Sie um die anderen Tools sinnvoll ausführen zu können.
- **gcc.** Der C-Compiler der GNU Compiler Collection. Dieses Tool benötigen Sie um Ihren C-Quelltext in ausführbare Programme zu übersetzen.
- **make.** Ein C-unabhängiges Buildsystem. Wenn Sie ein Projekt mit mehreren Quelltext-Dateien haben, dann müssen Sie diese mit mehreren `gcc` Aufrufen zu Objekt-Dateien übersetzen und diese dann zu einem ausführbaren Programm linken. `make` erlaubt es Ihnen diese `gcc`-Aufrufe in ein sogenanntes `Makefile` zu schreiben und ihr gesamtes Programm durch einen einzelnen Aufruf von `make my_program` zu übersetzen und zu linken. Dabei werden nur diejenigen Quelltext-Dateien neu übersetzt, die sich seid dem letzten Übersetzen verändert haben, um den Kompiliervorgang zu beschleunigen.
- **git.** Das weitverbreiteste Versionskontrollsystem. Dieses Tool benötigen Sie um Ihren Code abzugeben.
- **clang-format.** Ein Code-Formatter, der automatisch C-Quelltext so umformatiert, dass er bestimmten stilistischen Kriterien entspricht.
- **clang-tidy.** Ein Code-Linter, der automatisch überprüft ob Sie die richtigen Schreibweisen für Bezeichner verwenden, z.B. `myAwesomeFunction` statt `my_awesome_function`.
- **Texteditor.** Wenn Sie etwas Zeit übrig haben, um in eine weitere neue Welt einzutauchen, können Sie sich gerne an `vim` oder `emacs` versuchen, ansonsten ist aber auch jeder andere Texteditor geeignet, welcher Syntaxhighlighting für C unterstützt, wie z.B. `vscode` oder `gedit`.

Aufgabe 1.1 (Tutorial-Videos)

Zur Unterstützung der Vorlesung habe ich zwei Tutorial-Videos aufgenommen:

(a) eine Einführung in Linux, Terminals und Shells:

`https://youtu.be/9t0Qwc-hFHU`

(b) eine Einführung in `git` und unsere Lehrplatform:

`https://youtu.be/sOjv2I98uOI`

Schauen Sie sich diese Videos an. Im zweiten Video wird auch gezeigt wie man zu unserem Matrix Chat gelangt, in dem Sie gerne jederzeit Fragen zu Aufgaben, Tools und der Vorlesung stellen können.

Aufgabe 1.2 (Unix)

In dieser Vorlesung arbeiten wir in einer Unix-Umgebung, wie sie in Linux und MacOS gegeben ist. Es steht Ihnen frei davon abzuweichen, aber Ihre Abgaben müssen auf unserem Buildserver fehlerfrei durchlaufen, welcher eine `Makefile` erwartet die versucht den Quelltext mit `gcc` zu kompilieren.

Sofern Sie nicht bereits Linux/MacOS verwenden, ist es vermutlich am einfachsten Linux als Zweitsystem zu installieren. Hierzu empfehlen wir Ubuntu 20.04 oder neuer, da dort die benötigte Software in den korrekten Versionen angeboten wird.

Für Windows 10 besteht die Möglichkeit, das *Windows-Subsystem für Linux* zu verwenden. Dieses stellt ein vollwertiges Linux-Betriebssystem auf Windows bereit, benötigt aber etwas Konfigurationsaufwand. Ein Tutorial dazu gibt es unter:

`https://www.youtube.com/watch?v=Cvrqmq9A3tA`

Aufgabe 1.3 (Installation)

Die Installation der benötigten Software unterscheidet sich je nach Betriebssystem:

- Auf Ubuntu und dem Windows-Subsystem für Linux kann die benötigte Software in einem Terminal mit dem `apt`-Paketmanager wie folgt installiert werden:

  ```
  $ sudo apt-get install gcc make git clang-format-11 clang-tidy-11
  ```

- Auf MacOS kann die benötigte Software in einem Terminal mit dem `homebrew`-Paketmanager installiert werden. Die Installation von `homebrew` wird auf `brew.sh` beschrieben. Die Installation der Tools geht dann mit:

  ```
  $ brew install gcc make git clang-format
  ```

  Die Installation von `clang-tidy` ist auf MacOS leider etwas komplizierter und wird hier beschrieben:

  https://stackoverflow.com/questions/53111082/how-to-install-clang-tidy-on-macos

  Sie können stattdessen aber auch einfach das `clang-tidy` unseres Buildservers verwenden. Dies wird automatisch ausgeführt, wenn Sie mit `git push` Ihr Repository auf unserem `git`-Server updaten.

Überprüfen Sie ob die Programme korrekt installiert wurden, indem Sie diese mit dem Kommandozeilenargument `--version` aufrufen. Die Programme sollten daraufhin in etwa folgende Ausgabe erzeugen:

```
$ gcc --version
gcc (GCC) 9.3.0
[...]

$ make --version
GNU Make 4.3
[...]

$ git --version
git version 2.29.2

$ clang-format --version
clang-format version 11.1.0

$ clang-tidy --version
LLVM (http://llvm.org/):
  LLVM version 11.1.0
[...]
```

Die genauen Versionen sind nur bei `clang-tidy` wichtig. Wenn `clang-tidy` in Version kleiner 11 benutzt wird, werden Ihnen besimmte Funktionsnamen als falsch angezeigt, die eigentlich richtig sind. Sollte es nicht möglich sein `clang-tidy` in Version 11 zu installieren, können Sie sich stattdessen auch auf die Ausgabe unseres Buildservers verlassen, auf dem `clang-tidy` in der korrekten Version installiert ist.

Aufgabe 1.4 (Hello World)

Klonen Sie sich, wie im Tutorialvideo beschrieben, Ihr persönliches git-Repository von unserem git-Server:

```
$ git clone URL c-exercises
```

Die Abgabe des ersten Übungsblattes erfolgt im Unterverzeichnis `blatt01`, welches bereits in dem geklonten Repository zu finden ist. Achten Sie darauf, dass die Shell sich für die folgenden Befehle in diesem Verzeichnis befindet. Das aktuelle Verzeichnis der Shell kann mit dem Befehl `cd` ("change directory") gewechselt werden.

Erstellen Sie dort eine Datei `hello.c` mit folgendem Inhalt:

```
#include <stdio.h>

int main(int argc, char** argv) {
  printf("Hello, world!\n");
}
```

Kompilieren Sie `hello.c` zu einer Objekt-Datei `hello.o` und linken Sie diese dann zu einem ausführbaren Programm `hello`:

```
$ gcc -c hello.c -o hello.o
$ gcc hello.o -o hello
```

Führen Sie das Programm aus und überprüfen Sie, dass es die erwartete Ausgabe erzeugt:

```
$ ./hello
Hello, World!
```

Erstellen Sie einen neuen git-Commit, welcher die Datei `hello.c` dem Repository hinzufügt:

```
$ git add hello.c
$ git commit -m "Added hello.c to my repository."
```

Laden Sie die Commits Ihrer bisherigen Abgabe auf unseren git-Server:

```
$ git push
```

Auf unserer Lehrplatform sollten Sie nun sehen, wie sich der Buildserver über Ihre Abgabe beschwert, da wir noch keine `Makefile` hinzugefügt haben. Dies beheben wir im nächsten Schritt.

Aufgabe 1.5 (Makefile)

Im Vergleich zu interpretierten Sprachen, wie `python`, wird bei kompilierten Sprachen, wie `C`, nicht direkt der Quelltext ausgeführt, sondern erst der Quelltext in Maschinensprache übersetzt und danach ausgeführt.

Da die Übersetzung bei größeren Projekten mit vielen `.c`-Dateien mehrere Stunden dauern kann, hat man diesen Vorgang in zwei Schritte unterteilt:

(a) Kompilieren: Zunächst wird der Quelltext der einzelnen `.c`-Dateien unabhängig voneinander zu Maschinensprache in `.o`-Dateien übersetzt.

(b) Linken: Anschließend müssen die `.o`-Dateien nur noch aneinandergeklebt werden, um ein ausführbares Programm zu bilden.

Beim ersten mal Kompilieren und Linken, muss dann zwar immer noch alles übersetzt werden, aber danach müssen nur noch diejenigen `.c`-Dateien übersetzt werden, die man verändert hat – für die anderen `.c`-Dateien sind die zugehörigen `.o`-Dateien noch vom ersten mal Kompilieren vorhanden.

Diesen Vorgang bei jeder Änderung von Hand zu machen ist mühselig und fehleranfällig: jede `.c`-Datei muss mit einem `gcc`-Befehl übersetzt werden und wenn man dies vergisst wird einfach stummschweigend die alte `.o`-Datei weiterverwendet, was zu großen Verwirrungen führen kann.

Ein Buildsystem, wie z.B. `make`, kann hier Abhilfe schaffen: bei `make` schreibt man die Befehle zum Kompilieren und Linken in eine sogenannte `Makefile` und lässt diese dann von dem `make`-Programm ausführen. Dies hat folgende Vorteile:

(a) man muss die Befehle zum Kompilieren der einzelnen Dateien nicht nach jeder Änderung erneut eingeben, und

(b) `make` findet automatisch heraus, welche Dateien sich seid dem letzten Aufruf von `make` verändert haben und kompiliert nur diese erneut.

Im einfachsten Fall besteht eine `Makefile` aus einer Liste von Regeln. Eine Regel hat dabei immer die Form

```
output: input1 input2 ... inputN
  command
```

und bedeutet, dass um die Datei `output` zu erstellen, `make` den Shell-Befehl `command` ausführen soll. Dies soll geschehen wenn entweder `output` noch nicht existiert oder wenn sich eine der `input`-Dateien verändert hat. Wenn ein `input` nicht existiert, so sucht `make` zunächst nach einer weiteren Regel, die angibt wie `input` erstellt werden soll und führt diese auf die gleiche Weise rekursiv aus.

Ein wichtiges Detail ist hier, dass die Einrückung vor `command` *keine* Leerzeichen sind, sondern ein einzelnes TAB-Symbol. Wenn Sie versehentlich Leerzeichen verwenden bekommen Sie folgende Fehlermeldung:

```
Makefile:32: *** missing separator.  Stop.
```

Für die beiden `gcc`-Aufrufe, zum Kompilieren und Linken unseres *Hello World*-Programms, schreiben wir also folgende Regeln in unsere `Makefile`:

```
hello.o: hello.c
  gcc -c hello.c -o hello.o

hello: hello.o
  gcc hello.o -o hello
```

Wenn wir nun `make hello` ausführen, dann sucht `make` nach einer Datei `Makefile` im aktuellen Verzeichnis, sucht dort nach einer Regel, die `hello` erstellen kann, und führt diese wie oben beschrieben aus:

```
$ make hello
gcc -c hello.c -o hello.o
gcc hello.o -o hello
```

Wenn wir `make hello` danach erneut ausführen, dann erkennt `make`, dass die Dateien noch aktuell sind und führt keinen der Befehle aus:

```
$ make hello
make: 'hello' is up to date.
```

Schreiben Sie als nächstes eine `bye.c`, die sich analog zu unserem *Hello World*-Programm von der Welt verabschiedet, und fügen Sie die zugehörigen Regeln Ihrer bisherigen `Makefile` hinzu:

```
$ make bye
gcc -c bye.c -o bye.o
gcc bye.o -o bye
$ ./bye
Goodbye, world!
```

Da wir nun zwei Programme in unserem Projekt haben, wäre es praktisch wenn wir einfach nur `make compile` schreiben könnten um beide Programme zu kompilieren und zu linken, anstatt `make hello` und `make bye` einzeln aufrufen zu müssen. Hierzu kann man sogenannte *`.PHONY`-Regeln* zur `Makefile` hinzufügen:

```
.PHONY: compile

compile: hello bye
```

Das `.PHONY` hat für `make` eine spezielle Bedeutung: die Namen die danach gelistet werden, in diesem Fall `compile`, werden von `make` nicht als echte Dateien behandelt, sondern gelten stets als nicht-existierend. Wir möchten ja mit `make compile` unsere Programme kompilieren und nicht wirklich eine Datei mit Namen `compile` erzeugen. Wenn wir `compile` unter `.PHONY` eintragen, verhindert dies Probleme, wenn es wirklich eine Datei mit Namen `compile` gibt.

Das `compile: hello bye` ist eine normale Regel, für die einfach kein Shellbefehl

ausgeführt werden soll, da wir lediglich die Regeln für `hello` und für `bye` zusammenfassen wollen.

Wenn wir nun die alten `.o`-Dateien und Programme mit `rm` löschen und `make compile` aufrufen, sehen wir, dass beide Programme gebaut werden:

```
$ rm hello.o hello bye.o bye
$ make compile
gcc -c hello.c -o hello.o
gcc hello.o -o hello
gcc -c bye.c -o bye.o
gcc bye.o -o bye
```

Unser Buildserver erwartet, dass Ihre `Makefile` die folgenden Aufrufe unterstützt:

- `make compile`, das Ihre Programme kompiliert und linkt.
- `make test`, das Ihre Unittests kompiliert und ausführt.
- `make checkstyle`, das Ihre Quelltexte mit `clang-tidy` nach unseren Vorgaben für Variablennamen überprüft.
- `make format`, das Ihre Quelltexte mit `clang-format` nach unseren Vorgaben formatiert.
- `make clean`, das die Dateien wieder löscht, die Sie durch `make` aus ihrem Quelltext erzeugt haben – also alle `o`-Dateien und Programme.

Für jedes Übungsblatt müssen diese Befehle implementiert werden und auf unserem Buildserver erfolgreich durchlaufen, was Sie auf der Webseite unseres Buildservers überprüfen können.

Hierzu erweitern wir zunächst unsere `.PHONY`-Regel:

```
.PHONY: compile test checkstyle format clean
```

Die noch fehlenden `make`-Aufrufe implementieren wir dann wie folgt:

- Da wir in diesem Übungsblatt noch keine Unittests verwenden, soll `make test` lediglich eine Nachricht drucken:

  ```
  test:
    echo "Everything is ok. There are no tests, yet."
  ```

- Für `make checkstyle` sollen alle unserer `.c`-Dateien mit `clang-tidy` überprüft werden:

  ```
  checkstyle:
    clang-tidy --quiet *.c --
  ```

  Das `--quiet` verhindert unnötige Informationen in der Ausgabe von `clang-tidy` und das `*.c` wird von der Shell ersetzt durch alle Dateienamen aus dem aktuellen Verzeichnis die mit `.c` enden.

- Für `make format` wollen wir alle unsere `.c`-Dateien mit `clang-format` umformatieren:

  ```
  format:
    clang-format -i *.c
  ```

  Ein Aufruf von `make format` überschreibt Ihre `.c`-Dateien, wodurch diese in Ihrem Texteditor unter Umständen aktualisiert werden müssen.

- Für `make clean`, sollen Dateien gelöscht werden, die wir mit `make` aus unserem Quelltext erzeugen. Hierzu verwenden wir einfach den `rm` Befehl, den wir zuvor von Hand benutzt haben:

  ```
  clean:
    rm -f hello.o hello bye.o bye
  ```

  Das `-f` sorgt dafür, dass `rm` sich nicht beschwert, wenn eine der zu löschenden Dateien nicht existiert.

Sie sollten nun beim Ausführen der Befehle folgende Ausgaben sehen:

```
$ make format
clang-format -i *.c

$ make checkstyle
clang-tidy --quiet *.c --

$ make clean
rm -f *.o hello bye

$ make compile
gcc -c hello.c -o hello.o
gcc hello.o -o hello
gcc -c bye.c -o bye.o
gcc bye.o -o bye

$ make test
echo "Everything is ok. There are no tests, yet."
Everything is ok. There are no tests, yet.
```

Fügen Sie die `Makefile` und `bye.c` mit einem Commit ihrem Repository hinzu und pushen Sie den Commit auf unseren `git`-Server:

```
$ git add Makefile bye.c
$ git commit -m "Finished Makefile exercise."
$ git push
```

Stellen Sie sicher, dass auf der Webseite des Build Servers alles korrekt durchläuft.

Aufgabe 1.6 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
Zeitaufwand: 3:30

<...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.

Vergessen Sie nicht die Datei `erfahrungen.txt` mit einem Commit zu ihrem Repository hinzuzufügen und den Commit mit einem Push auf unseren git-Server zu laden.

```
$ git add erfahrungen.txt
$ git commit -a -m "Added erfahrungen.txt."
$ git push
```