Einführung in die Informatik
Ausarbeitung Übung 3

Julian Niethammer

20. November 2023

1 Compile und Debug

```
// EuclidAlgo.cpp
#include <stdio.h>

int main()
{
    int a = 50;
    int b = 5;
    int erge = 0;

    if(a == 0)
    {
      erge = b;
    }
    else
    {
        while(b != 0)
        {
            if(a>b)
            {
                a = a-b;
            }
            else
            {
                b = b-a;
            }
        }
        erge = a;
    }
    printf("Wert: %i", erge);
}
```

1.1 gcc, g++, gbd

Die meisten Terminalbefehle werden vom g++ Compiler als auch vom c++ Compiler akzeptiert und ausgeführt. Wenn eine Befehl nur nützlich für c++ ist, wird dies im Handbuch expliziet erwähnt. Wenn eine Beschreibung des Befehls nicht eine spezielle Sprache erwähnt ist er für beide Sprachen bzw. compiler geeignet.
gcc ist der Compiler für c-Programme und g++ ist der Compiler für c++ Programme.
Wichtitg ist, dass es sich bei beiden eigentlich nicht nur um den Compiler handelt. Beide enthalten einen Linker, welcher den compilierten Code zusammensetzt und aus diesem Code(Assemblercode) ausführbaren Maschinencode erzeugt. Des Weiteren besitzen beide einen Präprozesser (Schaut,

welche Bibliotheken genutzt werden, ...).
gdb ist ein Debugger. Er ist nützlich, um zu verstehen, was das Programm macht und wie es sich verhällt. man kann also durch in in ein laufendes Programm hineinsehen.

- Das Programm an einer gewissen Stelle im Code stoppen und danach weiter ausführen
- Den Status des Programmes zum gestopten Zeitpunkt feststellen (Werte von Variablen. . .)
- Dinge im Programm ändern

1.2 Terminalbefehle für Compiler und debugger

Befehl:

- gcc -c, g++ -c
  Der Code wird kompiliert, aber noch nicht verlinkt (Linker wird nicht ausgeführt). Es werden die Dateien, welche kompiliert wurden in Form von Objektdateien ausgegeben (Endung .o)
  Beispiel: g++ EuklidA.cpp -c
- gcc -o „Name“, g++ -o „Name“
  Es wird eine Datei (ausführbare Datei, Objektdatei, ...) mit dem Namen erzeugt
  Beispiel: g++ EuklidA.cpp -o test
- gcc -g, g++ -g
  Fügt dem Maschinencode Debug Informationen hinzu. Diese werden im Format des Betriebssystems des Rechners hinzugeügt. Man kann den kompilierten Code später dann mit Hilfe enies Debuggers (bspw. GDB) debuggen.
- gcc -W, g++ -W Gibt vorhanden Warnungen aus. Falls der Code Fehler enthält
  Beispiel: g++ EuklidA.cpp -W

Befehl zum Aufrufen des Debuggers: gdb „Programmname“
Programm wird gedebugt und man kann verschiedene Befehle mit dem Debugger ausführen (Normalerweise gibt man Debugger noch weitere Argumente mit)

Befehle innerhalb des Debuggers:

- break „Zeilennummer“
  Setzt Breakpoint an einerbestimmten Zeile. Wenn man dann Programm ausführt wird dieses an diser Zeile gestoppt.

- run
  Programm wird ausgeführt
- c
  Programm wird nach Breakpoint weiter ausgeführt
- step
  Nächte Programmzeile wird ausgeführt.
- delete „Zahl“
  Breakpoint bekommt eine Zahl zuegwiesen mit diesem Befehl kann man ihn löschen.
- print „Variablenname“
  Gibt den Wert der Variablen am Breakpoint aus.

1.3 Ablaufbeschreibung einer Debug-Session

gcc -o EuclidProg EuclidAlgo.cpp -g
gdb
file [FileName]
lay next (x2)
break main / break ⟨line⟩
run
C Code line by line: next
Assembly line by line: nexti
print ⟨variable⟩

2 Build

2.1 make

Mithilfe von make kann man automatisch Änderungen eines Programmmes kompilieren bzw. mit wenigen Befehlen das Programm ausführen und debuggen. Es ist aber auch nützlich wenn manche Dateien automatisch geupdatet werden müssen. Um den make Befehl nutzen zu können muss man ein makefile erstellen. Dieses beschreibt die Beziehung zwischen den Dateien die geupdatet werden müssen und die Befehle für die Aktualisierung der einzelnen Dateien. Durch den make Befehl werden dann für alle Dateien/ Programme, die Änderungen enthalten, entsprechende Befehle ausgeführt.
⇒ make ist vor allem bei komplexen Projekten hilfreich, da in diesen viele Dateien komiopiliert werden müssen und dies durch make automatisiert werden kann. Zudem wird überprüft, ob alle oder nur bestimmte Dateien kompiliert werden müssen. Ein makefile kann Variablen enthalten, welche

innerhalb des Makefiles verwendet werden können. Darüber hinaus enthält ein makefile Regeln, die angeben, welche Dateien erstellt werden sollen und welche Abhängigkeiten die Dateien haben. Verwendetes Programm: Heron Verfahren
Vorgehen: Im selben Verzeichnis wie das Programm ein makefile erstellen.

nano makefile
Name „makefile“, weil make sucht expliziet nach dem Dateinamen makefile.

Makefile aufsetzen:

```
#Variablen
Coption = -W -g

#Regeln
all: EuclidAlgo
EuclidAlgo: EuclidAlgo.cpp
	g++ EuclidAlgo.cpp $(Coption) -o EuclidAlgo
	#Obiger Befehl auch ohne Variablen moeglich: g++ HeronV.cpp
	    -W -g>
ex: EuclidAlgo
	./EuclidAlgo
db: EuclidAlgo
	gdb EuclidAlgo
rm: EuclidAlgo
	rm EuclidAlgo
	rm EuclidAlgo.o
```

Man kann in einem makefile Variablen definieren, die dann verwendet werden können ⇒ spart Schreibarbeit
Vor dem Doppelpunkt steht der Name der Regel. Dahinter welche Dateien benötigt werden (hier immer nur eine). Darunter steht (mit Einschub!) der Terminalbefehl, der dann ausgeführt wird
all: wird verwendet, wenn man nur make aufruft (Defaultaufruf)
Die einzelnen Befehle kann man mit make „Befehl“ aufrufen
Beispiel: make db

2.2 cmake

2.2.1 Unterschied zwischen make und cmake

cmake ist ein System, welches ähnlich wie makefiles funktioniert, jedoch platformunabhängig ist.
Alles beginnt mit einer CMakeLists.txt Datei. In dieser kann man Anweisungen und Regeln schreiben, wie ein Projekt geupdatet, kompiliert und/oder ausgeführt werden soll. Diese Datei kann dann in verschiedene Build-Systeme umgewandelt (bspw. makefiles). Diese können dann auf den jewei-

ligen Plattformen ausgeführt werden und die Regeln und Anweisungen der CMakeLists.txt umsetzen.

- Cmake
  Plattformunabhängig auf einfache/schnelle Weise Code, updaten, ausführbar machen, debuggen, ... (buildsystems führ verschiedene Plattformen)

- make
  Code auf Linux auf einfache/schnelle Weise, updaten, ausführbar machen, debuggen, ...(buildsystem von Linux)

CMakeList.txt erstellen: nano CMakeLists.txt
Code:

```
cmake_minimum_required(VERSION 3.22.1)
project(EuclidAlgo)
add_executable(EuclidAlgoExe EuclidAlgo.cpp)
```

Weitere Terminalbefehle: mkdir build
Um ausführbaren Code nicht mit Quellcode zu verwechseln Vorallem hilfreich bei komplexen Programmen
cd build
cmake ../
Die CmakeLists.txt aus dem übergeordnetem Verzeichnis wird aufgerufen. Und es wird in dem erstellten build Ortner eine Makefile Datei und weitere wichtige Dateien für Cmake erstellt

2.2.2 Was macht PHONY-target clean

.phony sorgt dafür, dass der Befehl clean ausgeführt wird und make nicht nach einer Datei namens clean sucht und diese versucht ausführbar zu machen. Das Phony target clean sollte die Makefile2-Datei im CMakeFiles Odrner bereinigen.

2.2.3 Prüfen der Funktionserfüllung

Die Funktionserfülung wurde überprüft, indem die Makefile Datei mit dem Befehl make ausgeführt wurde.
Es entsteht eine ausführbares Programm, welches man mit dem Terminalbefehl ./„Dateiname“ (hier: ./EuclidAlgo) ausführen kann.
Durch einsetzen von Testwerten kann man erkennen, dass das Prgramm das macht, was es soll.