universität freiburg

Programmieren in C++ SS 2024

Vorlesung 2: Compiler und Linker, Bibliothken

23. April 2024

Prof. Dr. Hannah Bast
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü1 — Programm + Drumherum
  - Diverse Hinweise — Korrektur, Copyright, …
- Inhalt
  - Compiler und Linker — Was ist das + wofür braucht man das
  - Header Dateien — Trennung in .h und .cpp Dateien
  - Bibliotheken — Statisch, Dynamisch, Erzeugung, Suchpfade, Konfiguration
  - Besseres Makefile — Abhängigkeiten

Ü2: Programm vom Ü1 sauber in .h und .cpp Dateien zerlegen und Makefile geeignet anpassen

Erfahrungen mit dem Ü1

- Auszügen aus den Kommentaren

  Sehr gut erklärt, Ü1 im Grunde einfach, Drumherum viel Arbeit

  "Angenehmer Einstieg in C/C++, Vorlesungsstoff super erklärt"

  "Vorlesung war sehr gut gemacht; was extrem viel Zeit gekostet hat war die Installation (z.B. gtest); insgesamt 5-6 Stunden"

  "Wie angekündigt: Programm war simpel, der Rest hat gedauert"

  "Die Aufzeichnung nochmal anzuschauen hat sehr bei der Einrichtung und dem Lösen der Aufgaben geholfen"

  "Es wäre schön, wenn man eine VM in VMware hätte"

  "Insgesamt trotz Verzweiflungsmomenten eine positive Erfahrung"

  "Werde die VL eher online wahrnehmen, da ich dadurch zeitlich flexibler bin und etwas Angst vor der Professorin habe"

Fragen auf dem Forum

- Wichtige Anmerkungen
  - Wir freuen uns sehr über die vielen Fragen auf dem Forum
  - Fragen Sie bitte weiterhin
  - Aber fragen Sie bitte richtig, siehe Fragen auf dem Forum

    Richtig Fragen ist ein lebenswichtiger Soft Skill

    Lebenswichtige Soft Skills nebenher zu lernen ist ein wesentlicher Teil dieser Lehrveranstaltung

Korrektur Ihrer Abgaben

- Ablauf
  - Ihnen wird heute ein:e Tutor:in zugewiesen … er oder sie wird Ihre Abgabe dann im Laufe dieser Woche korrigieren

    Bis spätestens Freitag, allerspätestens Samstag
  - Sie bekommen dann folgendes Feedback
    - Ggf. Infos zu Punktabzügen oder Lob
    - Ggf. Hinweise, was man besser machen könnte
  - Machen Sie im obersten Verzeichnis Ihrer Arbeitskopie

    svn update
  - Das Feedback finden Sie dann jeweils in einer Datei

    blatt-<xx>/feedback-tutor.md

Copyright

- Hinweise zum "Copyright" Kommentar
  - Ich schreibe in der Vorlesung oben immer

    ```
    // Copyright 2024 University of Freiburg
    // Chair of Algorithms and Data Structures
    // Author: Hannah Bast <bast@cs.uni-freiburg.de>
    ```

  - Die ersten beiden Zeilen sollten Sie nicht schreiben, Ihr Code gehört ja Ihnen und nicht der Uni
  - Wenn Sie Codeschnipsel aus public/code übernehmen (was grundsätzlich erlaubt ist), bitte vermerken, z.B. so:

    ```
    // Author: Nurzum Testen <nurzum@testen.de>
    // Using various code snippets kindly provided via
    // public/code from the SVN of the SS 2024 course
    ```

Musterlösung

- Nur zu Ihrem privaten Gebrauch
  - Nach dem Abgabetermin steht die Musterlösung auf dem Wiki bereit und auch im SVN unter /loesungen
  - Diese Musterlösung ist **ausschließlich** für Ihren persönlichen Gebrauch bestimmt
  - Insbesondere dürfen Sie sie weder jetzt noch später an Dritte weitergeben

    Die Verwendung der Musterlösung aus Vorlesungen von den Vorjahren ist sowieso nicht erlaubt, siehe 10. Gebot auf dem Wiki

Compiler und Linker 1/5

- Compiler
  - Der **Compiler** übersetzt alle Funktionen aus der gegebenen Datei in Maschinencode

    clang++ **-c** <name>.cpp

    Das erzeugt eine Datei <name>.o

    Das ist für sich noch **kein** lauffähiges Programm
  - Mit nm –C <name>.o sieht man:
    - was bereit gestellt wird (T = text = code)
    - was von woanders benötigt wird (U = undefined)
    - Die Option –C wandelt dabei die compiler-internen Namen in die tatsächlichen (C++) Namen um
    - Weitere Infos, siehe man nm

Compiler und Linker 2/5

- Linker
  - Der **Linker** fügt aus vorher kompilierten .o Dateien ein ausführbares Programm zusammen

    clang++ <name1>.o <name2>.o <name3>.o …
  - Dabei muss gewährleistet sein, dass:
    - jede Funktion, die in einer der .o Dateien benötigt wird, wird von **genau einer** anderen bereitgestellt wird, sonst:

      "undefined reference to …" (nirgends bereit gestellt)

      "multiple definition of …" (mehr als einmal bereit gestellt)
    - **genau eine** main Funktion bereitgestellt wird, sonst

      "undefined reference to main" (kein main)

      "multiple definition of main" (mehr als ein main)

Compiler und Linker 3/5

- Compiler + Linker
  - Ruft man clang++ auf einer .cpp Datei (oder mehreren) auf

    clang++ <name1.cpp> <name2.cpp> …
  - Dann werden die eine nach der anderen kompiliert und dann gelinkt

    So hatten wir das ausnahmsweise in Vorlesung 1 gemacht, aber das machen wir ab jetzt anders
  - Im Prinzip könnte man auch .cpp und .o Dateien im Aufruf mischen: es würden dann erst alle .cpp Dateien zu .o Dateien kompiliert, und dann alles gelinkt

    Das ist aber kein guter Stil

Compiler und Linker 4/5

Bei wenig Code natürlich kein Problem

- Warum die Unterscheidung
  - **Grund:** Code ist oft sehr umfangreich und Änderungen daran sind oft inkrementell
    - Dann möchte man nur die Teile neu kompilieren müssen, die sich geändert haben!
    - Insbesondere will man ja nicht jedes Mal die ganzen Standardfunktionen (wie z.B. printf) neu kompilieren
  - In der letzten Vorlesung hatten wir den Code nach jeder Änderung von Grund auf neu kompiliert
  - Wir hatten aber auch da schon "vorkompilierte" Sachen "dazu gelinkt", z.B. das –lgtest bei unserem Unit Test

    Was es damit genau auf sich hat, sehen wir heute

Compiler und Linker 5/5

- Name des ausführbaren Programms
  - Ohne weitere Angaben heißt das Programm einfach

    a.out
  - Mit der –o Option kann man es beliebig nennen

    Konvention: wir nennen es in dieser Vorlesung immer so, wie die .cpp Datei in der die main Funktion steht (die in eine gleichnamige .o Datei kompiliert wird)

    clang++ –o PrimeMain PrimeMain.o Prime.o

    clang++ –o PrimeTest PrimeTest.o Prime.o

Header-Dateien 1/5

- Header-Dateien, Motivation

Nicht: Header Dateien
www.deppenleerzeichen.de

  - Bevor man eine Funktion benutzt, muss man sie entweder

    definieren: `bool checkIfPrime(int n, bool v) { ... }` oder

    deklarieren: `bool checkIfPrime(int n, bool v);`

  - Zum Beispiel brauchen sowohl PrimeMain.cpp als auch PrimeTest.cpp die Funktion checkIfPrime

  - Bisher hatten wir einfach in beiden Dateien ein #include der Definition der Funktion stehen:

    ```
    #include "./Prime.cpp"
    ```

    Dann wird die Funktion aber **zweimal** kompiliert, einmal für das Main Programm und einmal für das Test Programm

  - Eigentlich brauchen wir sie aber nur einmal kompilieren

Header-Dateien 2/5

- Header-Dateien, Implementierung
  - Deswegen **zwei separate** Dateien:

    Prime**.h** nur mit der Deklaration

    Prime**.cpp** mit der Definition (= Implementierung)
  - Die .h Datei mit der Deklaration brauchen wir für unser **Main** und für unser **Test**, dort schreiben wir jeweils:

    ```
    #include "./Prime.h"
    ```

    Zur Kontrolle, ob die Signatur von Deklaration und Definition übereinstimmen, sollte wir das auch in Prime.cpp machen
  - Die .cpp Datei brauchen wir nur einmal und wollen wir auch nur einmal kompilieren … das geht wie gesagt mit

    ```
    clang++ -c Prime.cpp
    ```

Header-Dateien 3/5

- Header-Dateien, Kommentare
  - Kommentare nur an eine Stelle und zwar in der .h Datei

    In der .cpp Datei schreiben wir statt einem Kommentar die folgende, genau 79 Zeichen lange Zeile:

    // _____

    Bei Kommentaren in der .h Datei **und** in der .cpp Datei käme es bei Änderungen unweigerlich zu Inkonsistenzen
  - Außerdem in jeder Datei #include von **genau** dem, was in der Datei auch wirklich gebraucht wird

    Insbesondere: nicht darauf verlassen, dass in einer der #include Dateien etwas "included" wird, das man braucht

Header-Dateien 4/5

- Zyklische Includes, Problem
  - Eine Header-Datei kann eine andere Header-Datei "includen"
  - Bei komplexerem Code ist das sogar die Regel
  - Dabei muss man einen "Zyklus" verhindern, zum Beispiel:
    - Datei xxx.h "included" (unter anderem) Datei yyy.h
    - Datei yyy.h "included" (unter anderem) Datei zzz.h
    - Datei zzz.h "included" (unter anderem) Datei xxx.h

      An dieser Stelle muss man verhindern, dass man xxx.h nochmal liest, sonst geht es immer so weiter

Header-Dateien 5/5

- Zyklische Includes, Lösung
  - In modernen C++ Compilern kann man dazu einfach ganz am Anfang (vor allem Code und auch vor irgendwelchen #include) der Header-Datei schreiben

    #pragma once

  - Kommandos, die mit # beginnen (z.B. #include oder #pragma) sind dabei keine Elemente der Programmiersprache, sondern Anweisungen an den Compiler

    Allerdings ist #include Teil des C/C++ Standards, man kann sich also darauf verlassen, dass es das macht, was es soll

    #pragma once ist **nicht** Teil des C/C++ Standards, aber wir verlassen uns trotzdem darauf; für ein "zu Fuß" Variante, siehe Vorlesung 2, Folien 19+20, Programmieren in C++, SS 2020

Bibliotheken 1/9

- Was ist eine Bibliothek
  - Eine Bibliothek ist vom Prinzip her nichts anderes als eine .o Datei, sie heißt nur anders:

lib<name>.a	a = archive	**statische** Bibliothek
lib<name>.so	so = shared object	**dynamische** Bibliothek

  - Typischerweise enthält eine Bibliothek den Code von sehr **vielen** Funktionen
  - Deswegen enthält die Datei zusätzlich einen **Index**, so dass der Linker den Code von einer bestimmten Funktion schneller findet

    Außerdem spielt bei Bibliotheken die Reihenfolge beim Linken eine Rolle (bei .o Dateien nicht) … siehe Folie 25

Bibliotheken 2/9

- Linken von einer Bibliothek
  - Geht genauso wie bei einer .o Datei, z.B.

    clang++ PrimeTest.o Prime.o libgtest.a

    clang++ PrimeTest.o Prime.o libgtest.so
  - Das setzt, wie bei einer .o Datei, voraus, dass die Datei im aktuellen Verzeichnis steht, sonst absoluten Pfad schreiben

    clang++ PrimeTest.o Prime.o /usr/local/lib/libgtest.a

    clang++ PrimeTest.o Prime.o /usr/local/lib/libgtest.so
  - Wir können uns die (zur Verfügung gestellten und benötigten) Symbole wie einer .o Datei anschauen, siehe Folie 8

    nm –C /usr/local/lib/libgtest.a

Bibliotheken 3/9

- Linken von einer Bibliothek
  - Typischerweise linkt man aber mit der Option –l (ell)

    clang++ PrimeTest.o Prime.o –lgtest
  - Dann entscheidet das System:
    - ob es mit libgtest.a oder libgtest.so linkt (statisch vs. dynamisch)
    - wo und in welcher Reihenfolge es danach sucht
  - Vorteil: die Bibliotheken können auf verschiedenen Systemen an verschiedenen Stellen stehen, trotzdem bleibt der Befehl zum Linken immer gleich
  - Mit der Option –L kann man Verzeichnisse angeben, in denen **zusätzlich** nach der Bibliothek gesucht werden soll

    clang++ … –L/usr/local/lib –lgtest

Bibliotheken 4/9

- Statische Bibliotheken
  - Bei einer **statischen** Bibliothek wird der benötigte Code aus der Bibliothek Teil des ausführbaren Programms

    Vorteil: Man braucht die Bibliothek nur beim Linken aber nicht zum Ausführen des Programmes

    Nachteil: Das ausführbare Programm kann dadurch sehr groß werden
  - Um gezielt eine statische Bibliothek zu linken:

    clang++ **–static** PrimeTest.o Prime.o –lgtest

    Es kann wohlgemerkt auch ohne das **–static** statisch gelinkt werden, nur überlässt man es dann dem System bzw. seiner Grundkonfiguration

Bibliotheken 5/9

- Dynamische Bibliotheken
  - Bei einer **dynamischen** Bibliothek steht im ausführbaren Code nur eine Referenz auf die Stelle in der Bibliothek

    Vorteil: Das ausführbare Programm wird viel kleiner

    Nachteil: Man braucht die Bibliothek zur Laufzeit
  - Achtung: nur weil die Bibliotheken beim Linken gefunden wurden, heißt noch nicht, ob sie auch bei der Ausführung des Programms gefunden werden

    Suchpfade dafür → siehe nächste Folie

    Schauen, welche Bibliotheken (nicht) gefunden werden:

    ldd PrimeMain

Bibliotheken 6/9

- Dynamische Bibliotheken, Suchpfade
  - Zwei Alternativen, um die Suchpfade zu setzen:

    1. Kommandozeile: export LD_LIBRARY_PATH=<path>

    Das setzt den Pfad aber nur temporär für die aktuelle Shell (das Programm, das in dem Konsolenfenster läuft)

    2. Den Pfad zu einer der Dateien in /etc/ld.so.conf.d hinzu fügen (typisch: …/local.conf), danach ldconfig ausführen

    "ld" ist der Name des Programms, das g++ zum Linken benutzt, der sogenannte "Linker"

    Der Ursprung des Namens ist unklar, mögliche Kandidaten sind: "**L**oa**D**", "**L**ink e**D**itor", "**Il**luminati Covi**d**-19", …

Bibliotheken 7/9

- Wie baut man eine Bibliothek
  - Grundlage ist einfach eine Menge von .o Dateien, die den Code von einer Menge von Funktionen enthalten
  - Eine statische Bibliothek baut man dann einfach mit:

    ar cr **lib<name>.a** <name1.o> <name2.o> **...**

    ar = archive ist der Name des Programms, cr = create
  - Eine dynamische Bibliothek baut man einfach mit:

    clang++ –f pic –shared –o **lib<name>.so** <name1.o> …

    shared = baue eine dynamische ("shared") Bibliothek

    pic = position-independent code = Code mit relativer Adressierung = der Code kann ohne Modifikation an einer beliebigen Stelle im Speicher ausgeführt werden

Bibliotheken 8/9

- Reihenfolge beim Linken
  - Bei .o Dateien spielt die Reihenfolge **keine Rolle**

    Es wird einfach für alle .o Dateien zusammen geschaut, welche Symbole gebraucht (U) und welche zur Verfügung (T) gestellt werden
  - Bibliotheken werden von links nach rechts wie folgt bearbeitet

    Für jede Bibliothek wird geschaut, welche Symbole aus den .o Dateien und Bibliotheken davor undefiniert sind und in dieser Bibliothek zur Verfügung gestellt werden

    Symbole der Bibliothek, die nicht gebraucht werden, werden in der Folge **ignoriert** (weil es potenziell viel zu viele sind)

Bibliotheken 9/9

- Welche Bibliothek wurde gelinkt
  - Es ist nicht unüblich, dass auf einem System mehrere Varianten oder Versionen der gleichen Bibliothek installiert sind

    Wie findet man dann darauf, welche tatsächlich gelinkt wurde?
  - Für dynamische Bibliotheken findet man das einfach mittels **ldd** auf dem ausführbaren Programm heraus, siehe Folie 22
  - Für statische Bibliotheken ist das nicht so einfach, ihr Code ist ja Teil des ausführbaren Programms geworden, siehe Folie 21

    Man kann dem Linker aber eine Option mitgeben, mit der er beim Linken genau sagt, was er alles macht, insbesondere welche Bibliotheken wo gesucht und gefunden wurden

    clang++ … -Wl,--verbose …

Anpassungen Makefile 1/8

- Abhängigkeiten, Motivation
  - Nehmen wir an, wir haben unsere drei .cpp kompiliert in:

PrimeMain.o	das Main Programm
PrimeTest.o	das Test Programm
Prime.o	die Funktion checkIfPrime…

  - Nehmen wir an, wir ändern PrimeMain.cpp
  - Dann bräuchte man nur PrimeMain.o neu zu erzeugen und PrimeMain neu zu linken

    Der Rest braucht nicht neu kompiliert / gelinkt zu werden
  - Es wäre schön, wenn das Makefile das erkennen würde

    Das kann es in der Tat, siehe nächste Folien

Anpassungen Makefile 2/8

- Abhängigkeiten, Realisierung
  - Man kann im Makefile **Abhängigkeiten** angeben:

    <target>: <dependency 1> <dependency 2> …
    <command 1>
    <command 2> …

  - Jetzt wird bei make <target> erst folgendes gemacht:

    make <dependency 1>
    make <dependency 2> usw.

  - Wenn es keine targets mit diesem Namen gibt, kommt eine Fehlermeldung von der Art

    "No rule to make target … needed by <target>"

Anpassungen Makefile 3/8

- Abhängigkeiten, Realisierung
  - Man kann im Makefile **Abhängigkeiten** angeben:

    <target>: <dependency 1> <dependency 2> …
        <command 1>
        <command 2> …

  - Wenn man jetzt make <target> macht, passieren zwei Dinge:

    1. Es wird make <dependency i> ausgeführt, für jedes i

    Man beachte, dass jedes dieser <dependency i> wieder ein target im Makefile sein kann, das selbst wiederum Abhängigkeiten haben kann

    Das setzt sich rekursiv fort; die Rekursion endet an targets, die keine weitere Abhängigkeiten haben; bei einem Zyklus bricht das Makefile die Rekursion an der betreffenden Stelle ab

Anpassungen Makefile 4/8

- Abhängigkeiten, Realisierung
  - Man kann im Makefile **Abhängigkeiten** angeben:

    <target>: <dependency 1> <dependency 2> …
        <command 1>
        <command 2> …

  - Wenn man jetzt make <target> macht, passieren zwei Dinge:

    2. Die Kommandos <command1>, <command2>, … werden genau dann ausgeführt, wenn mindestens **eine** der folgenden drei Bedingungen erfüllt ist:

    - Es existiert keine Datei mit Namen <target>
    - Es existiert keine Datei mit Namen <dependency i> für ein i
    - Eine der <dependency i> ist neuer als <target>

Anpassungen Makefile 5/8

- Beispiel: Bauen des Main Programmes

```
DoofMain: DoofMain.o Doof.o
  g++ -o DoofMain DoofMain.o Doof.o          (1)
DoofMain.o: DoofMain.cpp
  g++ -c DoofMain.cpp                        (2)
Doof.o: Doof.cpp
  g++ -c Doof.cpp                            (3)
```

  - Wenn man jetzt etwas an Doof.cpp ändert und dann make DoofMain macht, passiert Folgendes:

    Es wird make DoofMain.o und make Doof.o ausgeführt, dabei:

    (2) wird nicht ausg. (DoofMain.cpp nicht neuer als DoofMain.o)

    (3) wird ausgeführt (DoofMain hängt von Doof.o ab)

    (1) wird ausgeführt (Doof.o jetzt neuer als DoofMain)

Anpassungen Makefile 6/8

- Special targets: .SUFFIXES
  - Make hat jede Menge automatische Regeln

    Zum Beispiel, wie man eine .o Datei aus einer .cpp Datei macht, nämlich mit clang++ -c ...
  - Diese automatischen Regeln wollen wir für diese Vorlesung und das Ü2 **nicht** haben (Sie sollen es selber lernen)
  - Dazu schreiben wir in das Makefile ganz oben einfach:

    .SUFFIXES:

    Nicht vergessen für das Ü2! Wenn die automatischen Regeln fälschlicherweise aktiviert sind, ist es sehr schwer zu verstehen, warum das Makefile macht, was es macht

Anpassungen Makefile 7/8

- Special targets: .PHONY
  - Ein target heißt **phony**, wenn es keine Datei mit diesem Namen gibt und die Kommandos zu dem target auch keine Datei mit diesem Namen erzeugen ... phony = "unecht"

    Diese targets dienen einfach als Abkürzung für eine Abfolge von Kommandos ... was auch oft nützlich ist
  - Deswegen schreiben wir in unser Makefile oben

    .PHONY: compile checkstyle test clean
  - Wenn ein target unter seinen Abhängigkeiten auch nur ein phony target hat, werden die Kommandos immer ausgeführt

    Das ist wohlgemerkt keine neue Regel, sondern folgt aus den Bedingungen unter Punkt 2 von Folie 30

Anpassungen Makefile 8/8

- Special targets: .PRECIOUS
  - make unterscheidet intern zwischen "Endprodukten" und "Zwischenprodukten", zum Beispiel:
    - Bei make PrimeMain ist PrimeMain das **Endprodukt**
    - Die ganzen .o Dateien sind **Zwischenprodukte** ... weil man sie zum Ausführen der PrimeMain nicht braucht
  - Je nach Konfiguration von make, kann es sein, dass am Ende von make compile die .o Dateien gelöscht werden
  - Um das zu verhindern, schreibt man zu Beginn:

    PRECIOUS: %.o

Literatur / Links

- Compiler und Linker
  - Online Manuale zu clang++

    https://clang.llvm.org/

    Liste der wichtigsten Opionen

    clang++ --help
  - Wikipedias Erklärung zu Compiler und Linker

    http://en.wikipedia.org/wiki/Compiler

    http://en.wikipedia.org/wiki/Linker_(computing)
  - Statische und dynamische Bibliotheken

    http://en.wikipedia.org/wiki/Library_(computing)