Programmieren in C++

SS 2022

Vorlesung 2, Dienstag 3. Mai 2022

(Compiler und Linker, Bibliotheken)

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik, Universität Freiburg

Blick über die Vorlesung heute

UNI FREIBURG

- 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 1/3

UNI FREIBURG

- Auszügen aus den Kommentaren

  Die meiste Arbeit war wie erwartet und geplant das Drumherum

  Für die meisten keine größeren Probleme, für einige aber schon

  Angaben zum Zeitaufwand variieren stark, von 1 – 20 Stunden

  "Keine größeren Schwierigkeiten, ich war positiv überrascht"

  "Hat Spaß gemacht + gute Übung um in C++ reinzukommen"

  "Ich habe nicht vor die Vorlesung zu "konsumieren"

  

  "Mit der Aufzeichnung ging alles, aber mehr nachmachen, als richtig verstehen."

  Keine Sorge, das wird sich ändern

  

  "Prof ist Gandalf, der uns alles wie Zauberin erklärt und Licht ins Dunkel bringt."

  

Erfahrungen mit dem Ü1 2/3

UNI FREIBURG

- Widerstände und Missverständnisse

  Die üblichen Widerstände gegen cpplint.py → ein Stylechecker ist Standard in jedem Softwareprojekt (das 80-Zeichen-Limit auch)

  "The 80 character line limit is incredibly annoying"

  "Das einzige Ärgernis ist, dass man keine Wahlfreiheit bei der Klammerung hat"

  Man braucht nicht **vim** zu benutzen, das war nur ein Vorschlag

  Es lohnt sich allerdings, einen Editor zu beherrschen, den man von der Kommandozeile aus aufrufen kann, wie z.B. **vim**

  

  Linuxkenntnisse keine Voraussetzung, sondern lernen Sie hier mit

  "Die Vertrautheit mit Linux wurde vorausgesetzt und war für mich eine größere Hürde." Voraussetzung nein, Hürde ja

  

Erfahrungen mit dem Ü1 3/3

UNI FREIBURG

- Für die, für die es sehr schwierig war
  - Ich hoffe sehr, Sie waren bei der Fragestunde
  - Auch wer beim Ü1 große Schwierigkeiten hatte und lange gebraucht hat, ist bei uns **herzlich willkommen**
  - Es sind aber ein paar Voraussetzungen notwendig:

    1. Akzeptieren Sie, dass die Veranstaltung wahrscheinlich für Sie mehr Arbeit ist als 6 ECTS

    2. Nicht den Fehler im außen suchen (Veranstaltung doof, Erklärungen doof, alles doof)

    3. Früh anfangen mit dem Blatt und Hilfe suchen: im Forum, bei Fragestunden, oder Treffen mit Tutor:in ausmachen

Korrektur Ihrer Abgaben

UNI FREIBURG

- 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

    blatt-<xx>/feedback-tutor.txt

Copyright

UNI FREIBURG

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

    ```
    // Copyright 2022 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 2022 course
    ```

Musterlösung

UNI FREIBURG

- 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

Anmerkung zu den Folien

UNI FREIBURG

- Interaktive Vorlesung vs. Vollständige Folien
  - In der ersten Ausgabe dieser und anderer Vorlesungen habe ich sehr viel vorgemacht, mit sehr wenigen Folien

    Sehr interaktiv und lebendig, es gab aber immer wieder den Wunsch nach vollständigeren Folien mit mehr Details
  - Folien sind dann nach und nach umfangreicher geworden

    An sich sehr gut, aber wenn ich alles auf den Folien durchgehe, geht das stark auf Kosten der Interaktivität
  - Lösungsversuch: Ich werde bei einigen Folien …

    1. … darauf hinweisen, dass sie als Referenz oder Nacharbeiten gedacht sind und sie **nicht** im Detail durchgehen

    2. … sie kurz stehenlassen, damit man an der Stelle später gut die Aufzeichnung anhalten und sie lesen kann

Compiler und Linker 1/5

UNI FREIBURG

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

    g++ **-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

UNI FREIBURG

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

    g++ <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

UNI FREIBURG

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

    g++ <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

UNI FREIBURG

- 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

    g++ –o PrimeMain …

    g++ –o PrimeTest …

Header-Dateien 1/6

Nicht: Header Dateien
www.deppenleerzeichen.de

- Header-Dateien, Motivation
  - 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 2/6

Mit Option --repository=.
siehe Erklärung auf Folie 17

UNI FREIBURG

- 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** und cpplint.py möchte es auch für Prime.cpp … dort machen wir jeweils:

    ```
    #include "./Prime.h"
    ```
  - Die .cpp Datei brauchen wir nur einmal und wollen wir auch nur einmal kompilieren … das geht wie gesagt mit

    ```
    g++ –c Prime.cpp
    ```

Header-Dateien 3/6

UNI FREIBURG

- Header-Dateien, Details
  - 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 Kommentar 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 4/6

UNI FREIBURG

- Header-Guards, Motivation
  - 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 5/6

UNI FREIBURG

- Header-Guards, Implementierung
  - Dazu schreiben wir um den Inhalt jeder Header Datei etwas von folgender Art herum:

```
#ifndef XXX
#define XXX
...
#endif  // XXX
```

  - Wenn der Compiler die Datei das erste Mal sieht, wird dabei eine interne Variable definiert, das XXX oben

    Diese Variable nennt man auf Englisch "header guard"

  - Wenn der Compiler die Datei noch mal sieht, wird der Inhalt (die "..." oben) einfach übersprungen

Header-Dateien 6/6

UNI FREIBURG

- Header-Guards, Benennung der Variablen
  - Der Name der Header-Guard Variablen sollte möglichst eindeutig gewählt werden
  - Deswegen verlangt cpplint.py Pfad + Dateiname, z.B.

```
#ifndef PUBLIC_CODE_VORLESUNG_02_PRIME_H_
#define PUBLIC_CODE_VORLESUNG_02_PRIME_H_
...
#endif  // PUBLIC_CODE_VORLESUNG_02_PRIME_H_
```

    Falls der Code in einer SVN Arbeitskopie steht, verlangt cpplint.py den Pfad ab dem Oberverzeichnis der Kopie

  - Das lässt sich (und dürfen und **sollen** Sie) umgehen mit

```
python3 cpplint.py --repository=. …
```

Bibliotheken 1/7

UNI FREIBURG

- 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

Bibliotheken 2/7

UNI FREIBURG

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

    g++ PrimeTest.o Prime.o libgtest.a

    g++ PrimeTest.o Prime.o libgtest.so

    Das setzt voraus, dass die Bibliothek im aktuellen Verzeichnis steht, sonst Pfad davor schreiben, z.B.

    g++ PrimeTest.o Prime.o /usr/lib/libgtest.a

    g++ PrimeTest.o Prime.o /usr/lib/libgtest.so
  - Andere Verzeichnisse, in denen Bibliotheken oft stehen:

    /usr/local/lib, /lib/x86_64-linux-gnu, …

Bibliotheken 3/7

UNI FREIBURG

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

    g++ PrimeTest.o Prime.o –lgtest
  - Dann entscheidet das System, gegen welche der (oft vielen verschiedenen) vorhandenen Bibliotheken es linkt
  - 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 auch nach der Bibliothek gesucht werden soll

    g++ –L/usr/local/lib …

Bibliotheken 4/7

UNI FREIBURG

- 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 eine statische Bibliothek zu linken:

    g++ **–static** DoofTest.o Doof.o –lgtest

Bibliotheken 5/7

UNI FREIBURG

- 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 DoofMain

Bibliotheken 6/7

UNI FREIBURG

- 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", "I**l**luminati Covi**d**-19", …

Bibliotheken 7/7

UNI FREIBURG

- 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:

    g++ –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

Anpassungen Makefile 1/7

UNI FREIBURG

- 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 2/7

UNI FREIBURG

- 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 3/7

UNI FREIBURG

- 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 4/7

UNI FREIBURG

- 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 5/7

UNI FREIBURG

- Automatische Regeln
  - Make hat jede Menge automatische Regeln

    Zum Beispiel, wie man eine .o Datei aus einer .cpp Datei macht, nämlich mit g++ -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 6/7

UNI FREIBURG

- Phony targets
  - 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"

    Alle targets die wir in Vorlesung 1 benutzt haben (compile, checkstyle, test, clean) waren "phony" in diesem Sinne

    Phony targets dienen einfach als Abkürzung für eine Abfolge von Kommandos … was auch oft nützlich ist
  - 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 der vorvorherigen Folie 31

Anpassungen Makefile 7/7

- 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)

Literatur / Links

- Compiler und Linker
  - Online Manuale zum g++

    http://gcc.gnu.org/onlinedocs

    Oder von der Kommandozeile: man g++
  - 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)