Programmieren in C++

SS 2018

Vorlesung 2, Dienstag 24. April 2018

(Compiler und Linker, Bibliotheken)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü1 — Programm + Drumherum
  - Diverse Hinweise — Korrektur, Copyright, Feiertag
- Inhalt
  - Compiler und Linker — was + warum
  - Header Dateien — Trennung in .h und .cpp Dateien
  - Bibliotheken — statisch, dynamisch, Erzeugung
  - Besseres Makefile — Abhängigkeiten
  - **Ü2:** Programm vom Ü1 sauber in .h und .cpp Dateien zerlegen und Makefile geeignet anpassen

Erfahrungen mit dem Ü1

Polyvalenter 2-Hauptfächer
Bachelorstudiengang
mit Hauptfach Informatik

- Zusammenfassung / Auszüge
  - Wie angekündigt, war das Aufwändigste das Drumherum
  - Approximation über Integral war tricky (siehe Musterlösung)
  - Wunsch der Polyvalenten: Linux-Crashkurs (keine Systeme 1)
  - ASSERT_FLOAT_EQ für Unit Tests mit Fließkommazahlen

    Ungefähre Gleichheit, die eine Abweichungen von der Größe der inhärenten Ungenauigkeit der Darstellung akzeptiert
  - Bitte die Vim und Linux Tricks erklären? Fragen Sie gerne!
  - Sehr viele Fragen (und schnelle Antworten) auf dem Forum
  - Etwas längere Leine für erfahrene Programmierer?

Korrekturen Ihrer Abgaben

UNI FREIBURG

- Ablauf
  - Ihnen wird heute ein Tutor zugewiesen ... er/sie wird Ihre Abgabe dann im Laufe dieser Woche korrigieren

    Bis spätestens Freitag, allerspätestens Samstag
  - Sie bekommen dann folgendes Feedback
    - Ggf. Infos zu Punktabzügen
    - 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

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

    ```
    // Copyright 2018 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 von uns übernehmen, können Sie das ja in einem Folgekommentar vermerken, z.B.

    ```
    // Author: Nurzum Testen <nurzum@testen.de>
    // Using various code snippets kindly provided by
    // http://ad-wiki.informatik.uni-freiburg.de/teaching
    ```

Feiertag

- Nächste Woche (1. Mai) keine Vorlesung
  - Es ist Tag der "Arbeit"
  - Das ist eine Art Oxymoron, so wie in

    Offenes Geheimnis

    Eile mit Weile

    Viva la muerte!

Compiler und Linker 1/5

- 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 internen Namen (C Style) in die tatsächlichen (C++ Style) um
    - Weitere Infos, siehe man nm

Compiler und Linker 2/5

- 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

- 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

- 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 ApproximationOfPiMain ...

    g++ –o ApproximationOfPiTest ...

Header Dateien 1/6

- Header Dateien, Motivation
  - Bevor man eine Funktion benutzt, muss man sie deklarieren

    Das gilt insbesondere, wenn die Implementierung in einer anderen Datei steht (und am Ende erst dazu gelinkt wird)
  - Z.B. brauchen sowohl ApproximationOfPiMain.cpp als auch ApproximationOfPiTest.cpp die Funktion approximatePiUsing...
  - Bisher hatten wir einfach in beiden Dateien stehen:

    #include "./ApproximationOfPi.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

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

    ApproximationOfPi**.h** nur mit der Deklaration

    ApproximationOfPi**.cpp** mit der Implementierung
  - Die .h Datei mit der Deklaration brauchen wir für unser **Main** und für unser **Test** und cpplint.py möchte es auch für ApproximationOfPi.cpp ... dort machen wir jeweils:

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

    g++ **–c** ApproximationOfPi.cpp

Header Dateien 3/6

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

    In der .cpp Datei schreiben wir statt einem Kommentar:

    // _____

    Bei Kommentar in der .h Datei **und** in der .cpp Datei käme es bei Änderungen unweigerlich zu Inkonsistenzen
  - Außerdem in jeder Datei nur **genau** das includen, was in der Datei auch wirklich gebraucht wird
  - Insbesondere **keine impliziten includes** (durch includes in einer inkludierten Datei)

Header Dateien 4/6

- Header Guards, Motivation
  - Eine Header Datei kann eine andere "includen"
  - Bei komplexerem Code ist das sogar die Regel
  - Dabei muss man einen "Zyklus" verhindern, z.B.
    - 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

- 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 "Header Guard"

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

Header Dateien 6/6

- 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 APPROXIMATIONOFPI_H_
#define APPROXIMATIONOFPI_H_
...
#endif  // APPROXIMATIONOFPI_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

```
python cpplint.py --repository=. ...
```

Bibliotheken 1/7

- 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

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

    g++ DoofTest.o Doof.o libgtest.a

    g++ DoofTest.o Doof.o libgtest.so

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

    g++ DoofTest.o Doof.o /usr/local/lib/libgtest.a

    g++ DoofTest.o Doof.o /usr/local/lib/libgtest.so

Bibliotheken 3/7

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

    g++ DoofTest.o Doof.o –lgtest
  - Dann entscheidet das System, gegen welche der 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 zusätzliche Verzeichnisse angeben, in denen nach der Biblio. gesucht werden soll

    g++ –L/usr/local/lib ...

Bibliotheken 4/7

- 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

- 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
  - Vor der Ausführung schauen, ob alle benötigten dynamischen Bibliotheken gefunden werden und wo:

    ldd DoofMain

Bibliotheken 6/7

- 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
  - Zwei Alternativen, um die Suchpfade dafür zu setzen:

    1. 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

    2. Kommandozeile: export LD_LIBRARY_PATH=...

    das setzt den Pfad, aber nur temporär, für das aktuelle Fenster

Bibliotheken 7/7

- 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++ –fpic –shared –o **lib<name>.so** <name1.o> ...

    shared = dynamisch, fpic = siehe g++ Dokumentation

Anpassungen Makefile 1/6

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

    ApproximationOfPiMain.o das Main Programm
    ApproximationOfPiTest.o das Test Programm
    ApproximationOfPi.o die Funktion approximatePiUsing...

  - Nehmen wir an, wir ändern ApproximationOfPiMain.cpp
  - Dann bräuchte man nur ApproximationOfPiMain.o neu zu erzeugen und ApproximationOfPiMain 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 2/6

- 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 3/6

UNI FREIBURG

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

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

  - Nach make <dependency 1>, usw. werden die Kommandos <command1>, <command2>, ... ausgeführt, außer wenn jede der drei folgenden Bedingungen erfüllt ist:
    - Es existiert bereits eine Datei mit Namen <target>
    - Es existieren Dateien <dependency 1>, <dependency 2>, ...
    - <target> ist neuer als alle <dependency i>

Anpassungen Makefile 4/6

- 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 automatische 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:

Anpassungen Makefile 5/6

- 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 = künstlich

    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 folgt aus der Regel von der vorvorherigen Folie

Anpassungen Makefile 6/6

- 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.o                                (3)
```

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

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

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

    (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)