Programmieren in C++

SS 2016

Vorlesung 2, Dienstag 26. April 2016

(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
  - Hinweise zur Korrektur — feedback.txt
- Inhalt
  - Compiler und Linker — was + warum
  - Header Dateien — Trennung in .h und .cpp Dateien
  - Bibliotheken — statisch, dynamisch, Erzeugung
  - Besseres Makefile — Abhängigkeiten
  - **Übungsblatt 2:** Programm vom Ü1 sauber in .h und .cpp Dateien zerlegen und Makefile geeignet anpassen

Erfahrungen mit dem Ü1 1/3

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Wie erwartet, war das Aufwändigste das Drumherum
  - Sonst war das Blatt aber für die allermeisten gut machbar
  - Aufzeichnung + Vorlesungscode dabei sehr hilfreich
  - Live-Programmieren stellenweise etwas zu schnell für manche
  - Probleme konnten schnell auf dem Forum geklärt werden
  - Insbesondere einige Probleme mit dem **Linken** der Bibliotheken von gtest: –lgtest usw.

    Genau das Thema der heutigen Vorlesung !

Erfahrungen mit dem Ü1 2/3

UNI FREIBURG

- Was durfte / darf man benutzen?
  - Bitte bis auf Weiteres immer nur so viel benutzen, wie schon in der Vorlesung drankam

    Insbesondere brauchte man für das Ü1 kein char* oder std::string, man konnte einfach einen int zurückgeben ... was zudem hier sowieso die elegantere Lösung ist

    Absolut verboten: externe Bibliotheken (z.b. boost) in unser SVN hochladen
  - Zu C++11 sage ich etwas, wenn wir so weit sind

    In den ersten Vorlesungen machen wir sowieso noch kaum C++ sondern vor allem C (was ja ein Teil von C++ ist)

Erfahrungen mit dem Ü1 3/3

- Pfingsten
  - Lässt sich kommerziell nicht ausschlachten
  - Schokolade würde im Sommer zu schnell schmelzen
  - Pfingstochse wäre kein Kassenschlager als Schokofigur
  - An Pfingsten sind alle im Urlaub, selbst die Kirchgänger
  - Als ich schließlich den blauen Punkt bei Jenkins sah, war das wie Weihnachten, Ostern und Pfingstmontag zusammen
  - Die Freude, als der rote Kreis in Jenkins blau wurde ist unbeschreiblich ... nicht mal bei meiner Geburt bin ich so glücklich gewesen

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 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

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

- 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

    Bei so einem kleinen Programm natürlich kein Problem
  - 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 BirthdayParadoxMain ...

    g++ –o BirthdayParadoxTest ...

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 BirthdayParadoxMain.cpp als auch BirthdayParadoxTest.cpp die Funktion probabilitySameBirthday
  - Bisher hatten wir einfach in beiden Dateien stehen:

    #include "./BirthdayParadox.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

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

    BirthdayParadox**.h** ... nur mit der Deklaration

    Die brauchen wir für unser Main und für unseren Test

    Dort machen wir jeweils #include "./BirthdayParadox.h"

    BirthdayParadox**.cpp** ... mit der Implementierung

    Die brauchen wir nur einmal und wollen wir auch möglichst nur einmal kompilieren

    Wie bereits erklärt mit g++ **–c** BirthdayParadox.cpp

Header Dateien 3/6

- Header Dateien, Details
  - Kommentare nur an eine Stelle und zwar in der .h Datei
  - In der .cpp Datei dann zum Beispiel einfach:

    // _____
  - Außerdem in jeder Datei **genau** das includen, was in der Datei gebraucht wird

    Insbesondere: keine impliziten includes (durch includes in einer inkludierten Datei), das gibt checkstyle Mecker

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, hier XXX

    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 TEST_BLATT_02_XXX_H_
#define TEST_BLATT_02_XXX_H_
...
#endif  // TEST_BLATT_02_XXX_H_
```

  - Falls der Code in einer SVN Arbeitskopie steht, was er sollte, reicht der Pfad ab dem Oberverzeichnis der Kopie

    Sonst verlangt cpplint.py den absoluten Pfad

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

    Alternativ: Suchpfad mit der Option –L angeben:

    g++ –L/usr/local/lib ...

    funktioniert nur beim Linken mit –l... siehe nächste Folie

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

    Besonders wichtig für Code, der auf vielen verschiedenen Plattformen kompiliert werden soll

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 1/5

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

    BirthdayParadoxMain.o das Main Programm
    BirthdayParadoxTest.o das Test Programm
    BirthdayParadox.o die Funktion probabilitySameBirthday

  - Nehmen wir an, wir ändern etwas in BirthdayParadoxMain.cpp
  - Dann bräuchte man nur BirthdayParadoxMain.o neu zu erzeugen und BirthdayParadoxMain 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 2/5

- 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 3/5

- 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:
    - Es existiert eine Datei mit Namen <target>
    - Es existieren Dateien <dependency 1>, <dependency 2>, ...
    - Keine der Dateien <dependency i> ist neuer als <target>

Anpassungen Makefile 4/5

- 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

    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 vorherigen Folie

Anpassungen Makefile 5/5

- 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 Manual zum g++ Version 4.7, 4.8, 4.9

    http://gcc.gnu.org/onlinedocs/gcc-4.<x>.0/gcc

    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)