Programmieren in C++

SS 2012

Vorlesung 2, Mittwoch 8. Mai 2012

(Compiler und Linker, Bibliotheken, Jenkins)

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 1. Übungsblatt / Korrekturen
  - gtest und cpplint.py ... wohin damit
  - Abschreiben ... und die Folgen
- Compiler und Linker
  - Was macht der Compiler und was macht der Linker?
  - Trennung in .h und .cpp Dateien
  - Konsequenzen für das Makefile
  - Übungsblatt: all das für das Programm vom Ü1 machen

    (das wird für die meisten sehr schnell gemacht sein)
- Continuous Build System (Jenkins)
  - Was ist das und wie hilft uns das?

Erfahrungen mit dem Ü1 (erstes Programm)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 2.5. 14:00
  - Bearbeitungszeit 30 Minuten – 2 Tage
  - Wie erwartet ging die meiste Zeit für das Drumherum drauf
    - vor allem gtest, aber auch SVN, Checkstyle, Linux, ...
  - Viel Hilfe zum gtest Problem im Forum ... aber sehr verteilt
  - Problem mit –lpthread gab es letztes Jahr schon
  - Probleme bei ASSERT_EQ(false, ...) mit neuem gtest
  - Checkstyle Skript ist doof / pingelig / absurd / sinnlos / ...
    - "epische Schlacht mit cpplint.py"
  - Art der Vorlesung gut: vormachen, praktisch, low-level, ...
  - Für manche etwas schnell ... und viel auf einmal ... und heiß
  - Aufzeichnung dabei aber (und überhaupt) sehr hilfreich

Erfahrungen mit dem Ü1 (erstes Programm)

UNI FREIBURG

- Fortsetzung …
  - Was ist mit großen Zahlen, die nicht in einen int passen?
  - Hoffe selber nette Tutor wie letztes Semester in AlgoDat
  - Die Klausur in AlgoDat war gemein
  - Viele Befehle eingeführt ohne genauere Erklärung
  - In cpplint.py bitte #!/usr/bin/python statt #!/usr/bin/python2.4
  - Was darf man benutzen und was nicht, z.B. #include <math.h>

Korrekturen

- Das 1. Übungsblatt ist schon korrigiert
  - Sie bekommen Feedback von Ihrem Tutor / Ihrer Tutorin
    - Insbesondere Infos zu Punktabzügen
    - Wobei wir Sie bei vielen Sachen erstmal freundlich hinweisen, wie man es besser machen könnte / sollte
  - Machen Sie dazu einfach in Ihrer Arbeitskopie

    svn update
  - Das Feedback finden Sie dann in

    uebungsblatt-1/feedback-tutor.txt
  - Das gilt entsprechend für alle kommenden Übungsblätter

gtest und cpplint.py ... wohin damit

UNI FREIBURG

- gtest
  - Installieren Sie das bitte bei sich **global**
  - Auf dem Wiki steht eine (aktualisierte) Anleitung dazu
  - Dann funktioniert es auch wie gezeigt mit ... –lgtest
    - wobei manche auch noch ... –lpthread brauchen
- cpplint.py
  - Kopieren Sie die bitte in das oberste Verzeichnis Ihrer Arbeitskopie und committen Sie es (einmalig) in's SVN
  - In den Makefiles für die einzelnen Übungsblätter dann wie ursprünglich gezeigt python ../cpplint.py *.h *.cpp
  - Sie sehen dann auf Jenkins ob alles richtig ist → später

Abschreiben … und die Folgen

- Nochmal ausführlich und klar
  - Sie können gerne über die Aufgaben, Lösungswege, Probleme, usw. miteinander reden
  - Und natürlich Fragen im Forum stellen
  - Aber den Code muss am Ende **jeder selber** schreiben
    - sonst lernt man es nicht
  - Code von anderen abschreiben gilt als Täuschungsversuch und hat das Nicht-Bestehen der Veranstaltung zur Folge
    - das gilt auch wenn man ein paar Benennungen ändert, oder den Code geringfügig umstellt, oder ähnliches …

Compiler und Linker 1/4

UNI FREIBURG

- Warum die Unterscheidung
  - Eigentlich will man ja nur ein lauffähiges Programm, und dafür ist der Compiler da, warum also so kompliziert?
  - Grund: Code ist oft sehr umfangreich und man ändert ihn inkrementell
    - Dann möchte man nur die Teile neu kompilieren müssen, die sich geändert haben!
    - Insbesondere will man ja nicht jedesmal die ganzen Standardfunktionen (wie z.B. printf) neu kompilieren

Compiler und Linker 2/4

- 1. Schritt (Compiler)
  - Wir übersetzen die Funktion, das Main und das Test separat, aber machen noch kein ganzes Programm daraus
    - das geht mit g++ -c <name>.cpp
    - damit bekomme wir zu jeder .cpp Datei eine .o Datei
    - die enthält den Maschinencode für die jeweiligen Funktionen
    - mit nm –C <name>.o sieht man welche Funktionen eine .o Datei bereitstellt (T = text = code) und welche sie von woanders benötigt (U = undefined)
  - Das schauen wir uns am Beispiel unseres Programms aus der 1. Vorlesung (LeapYear) an

Compiler und Linker 3/4

- 2. Schritt (Linker)
  - Wir fügen die .o Dateien zu einem einzelnen ausführbaren Programm zusammen, das nennt man linken
    - LeapYear.o und LeapYearMain.o geben das ausführbare Main Programm
    - LeapYear.o und LeapYearTest.o geben das ausführbare Test Programm
  - Beim Zusammenfügen muss gewährleistet sein
    - dass jede Funktion, die in einer der gelinkten .o Dateien benötigt wird von genau einer anderen bereitgestellt wird
      - sonst "undefined reference" bzw. "multiple definition of"
    - dass genau eine main Funktion bereitgestellt wird
      - sonst "multiple definition of main"

Compiler und Linker 4/4

UNI FREIBURG

- 2. Schritt (Linker) ... Fortsetzung
  - Eine Funktion bereitstellen, die nirgendwo benötigt wird, ist **kein** Problem und gibt auch keine Fehlermeldung
    - insbesondere machen das Bibliotheken in hohem Maße
    - eine Bibliothek ist nichts anderes als eine .o Datei
      - mit einer typischerweise großen Menge an bereit gestellten Funktionen, z.B. printf
      - und eine speziellen Index, so dass der Linker die gewünschte Funktion schnell findet
      - mehr zu Bibliotheken auf den nächsten Folien ...

Bibliotheken 1/3

- Wissenswertes
  - Standardbibliotheken oder solche von anderen Programmen stehen typischerweise im Verzeichnis /usr/lib oder /usr/local/lib
  - es gibt statische und dynamische Bibliotheken → nächste Folie
  - sie heißen lib<name>.a (stat.) bzw. lib<name>.so.<x> (dyn.)
  - Man kann Sie einfach über ihren Dateinamen dazulinken, z.B.

    g++ LeapYearTest.o LeapYear.o /usr/lib/libgest.a
  - Besser ist aber man schreibt

    g++ LeapYearTest.o LeapYear.o –lgtest
    - der Compiler sucht dann in seinen Standardverzeichnissen (siehe oben) nach libgtest.a bzw. libgtest.so.1 etc.
    - mit –L <Ordnername> kann man angeben, wo noch gesucht werden soll

Bibliotheken 2/3

- Statisch vs. Dynamisch
  - Code aus einer statischen Bibliothek wird Teil des ausführbaren Programms
    - Vorteil: man braucht die Bibliothek nur beim Linken aber nicht zum Ausführen des Programmes
    - Nachteil: das ausführbare Programm kann sehr groß werden
  - Bei einer dynamischen Bibliothek steht im ausführbaren Code nur eine Referenz auf die Stelle in der Bibliothek
    - Vorteil: das ausführbare Programm wird kleiner
    - Nachteil: man braucht die Bibliothek zur Laufzeit
    - Mit ldd bekommt man die von einem ausführbaren Programm benötigten dynamischen Bibliotheken

Bibliotheken 3/3

- Man kann sich Bibliotheken auch leicht selber bauen
  - Eine statische Bibliothek baut man einfach (wie eine .o Datei auch) mit g++ -c –o lib<name>.a ...
  - Mit ar c lib<name>.a <.o file 1> <.o file 2> ... kann man sich aus einer beliebigen Menge von .o Dateien eine statische Bibliothek bauen
  - Eine dynamische Bibliothek baut man mit g++ -fpic –shared –o lib<name>.so ...
    - Das ausführbare Programm sucht dann **zur Laufzeit** in den Standardverzeichnssen nach dieser Bibliothek
    - Steht die Bibliothek woanders muss man setzen

      export LD_LIBRARY_PATH=<folder name>

Header bzw. .h Dateien

- Wofür braucht man die?
  - Bevor man eine Funktion benutzt, muss man sie deklarieren (so wie eine Variable)
    - auch wenn die Implementierung in einer anderen Datei steht (und dann am Ende dazugelinkt wird)
  - In einer .cpp Datei (oder eine ganze Bibliothek) sind oft viele Funktionen implementiert
    - jeder, der eine oder mehrere von diesen Funktionen benutzen will muss sie dann erst deklarieren
    - deswegen sammelt man die ganzen Deklarationen in einer sogenannten header Datei, die enden auf .h
    - braucht man eine oder mehrere davon macht man

      #include <xyz.h> sucht in /usr/include etc. oder

      #include "./xyz.h" Pfad explizit angeben

Header guards

- Eine .h Datei kann andere .h Dateien includen
  - Bei komplexerem Code ist das sogar die Regel
  - Dabei muss man einen "include cycle" verhindern
    - 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
  - Dazu gibt es die sogenannten header guards am Anfang und Ende jeder .h Datei

```
#ifndef VORLESUNGEN_VORLESUNG_2_XXX_H_
#define VORLESUNGEN_VORLESUNG_2_XXX_H_
...
#endif  // VORLESUNGEN_VORLESUNG_2_XXX_H_
```

Make

- Abhängigkeiten
  - Man kann make sagen, dass ein bestimmtes target von anderen targets abhängt, letztere heißen dann dependencies

    <target>: <dependency 1> <dependency 2> ...
    <command 1>
    <command 2>
    ...
  - Jetzt wird bei make <target>, erstmal make <dependency 1>, make <dependency 2> usw. ausgeführt und dann erst die Kommandos <command 1>, <command 2> usw.
  - Wenn <target> ein Dateiname ist, werden die Kommandos nur ausgeführt wenn
    - es eine Datei mit dem Namen <target> noch nicht gibt
    - oder es diese Datei gibt, sie aber **älter** ist als eine existierende Datei mit Namen <dependency i>

Continuous Build System (Jenkins)

UNI FREIBURG

- Features
  - Läuft auf einem separaten Rechner
  - "Baut" Ihr Programm periodisch oder wann immer Sie etwas im SVN ändern

    make compile test checkstyle
  - Über ein Web Interface können Sie sich die einzelnen "Builds" bequem anschauen
  - Ab jetzt Pflicht für jedes Übungsblatt auf Jenkins nachzuprüfen, das alles funktioniert
  - Bei Problemen bitte Mail ans Unterforum "Daphne, SVN, Make, Jenkins, etc."

Warnungen vom g++

- ... sollten grundsätzlich ernst genommen werden
  - Auch wenn das Programm trotzdem kompiliert und läuft
  - In den allermeisten Fällen liegt ein Progammierfehler vor
  - Ab jetzt übersetzen wir immer mit der Option –Wall
  - Dann werden alle Warnungen angezeigt
    - die leicht zu vermeiden sind und
    - meistens auf einen Programmierfehler hindeuten

Vim

- Das ist der Editor den ich benutze
  - Sie können aber irgendeinen Editor benutzen
  - Es sollte aber einer sein, der viel kann, insbesondere
    - Syntax Highlighting
    - Autovervollständigung
    - Blick auf mehrere Dateien gleichzeitig
    - Komfortables Wechseln zwischen Dateien
  - Auf dem Wiki finden Sie einen Link zu meiner .vimrc
    - Das ist die / meine Konfigurationsdatei von Vim
    - Die setzt einige nützliche Standardeinstellungen
    - Und definiert jede Menge nützlicher shortcuts

Literatur / Links

- Compiler und Linker
  - Online Manual zum g++ Version 4.7

    http://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc
  - Linker Optionen von eben diesem

    http://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Link-Options.html#Link-Options
  - 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)