Programmieren in C

SS 2019

Vorlesung 2, Dienstag 30. April 2018

(Compiler und Linker, Bibliotheken)

Prof. Dr. Peter Thiemann
Lehrstuhl für Programmiersprachen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Hinweise — Korrektur, Copyright
- Inhalt
  - Testen mit assert
  - Compiler und Linker — was + warum
  - Header Dateien — Trennung in .h und .c Dateien
  - Besseres Makefile — Abhängigkeiten
  - Live Programm — Arrays (Felder)
  - **Ü1:** Programme vom Ü0 sauber in .h und .c Dateien zerlegen, Makefile geeignet anpassen, Arrays

Korrekturen Ihrer Abgaben

UNI FREIBURG

- Ablauf
  - Ihnen wird heute ein Tutor zugewiesen … er/sie wird Ihre Abgabe dann im Laufe dieser Woche korrigieren
  - Sie bekommen 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 jeweils in

    blatt-<xx>/feedback-tutor.txt

Copyright

UNI FREIBURG

- Hinweise zum "Copyright" Kommentar
  - Ab jetzt schreibe ich immer

    // Copyright 2019 University of Freiburg
    // Author: Peter Thiemann <thiemann@informatik.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://proglang.informatik.uni-freiburg.de/

Testen mit assert

UNI FREIBURG

- Assert ist eine Funktion in der C Standardbibliothek
  - Verwendung in approximationOfPiTest.c

    ```
    #include <assert.h>
    ```

  - Dann in der main() Funktion die Testfälle angeben

```
int main (void) {
  assert (countNumberOfPointsInCircle (1) == 5);
  assert (countNumberOfPointsInCircle (2) == 13);
  return 0;
}
```

  - Falls assert fehlschlägt, dann gibt es einen Laufzeitfehler mit Zeilennummer, Dateiname und dem Ausdruck

Compiler und Linker 1/5

UNI FREIBURG

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

    cc **-c** <name>.c

    Das erzeugt eine Objektdatei namens <name>.o

    Das ist für sich noch **kein** lauffähiges Programm
  - Kann mit nm <name>.o inspiziert werden:
    - was wird bereit gestellt (T = text = code)
    - was wird von woanders benötigt (U = undefined)
    - 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

    cc <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 cc auf einer .c Datei (oder mehreren) auf

    cc <name1.c> <name2.c> …
  - 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 .c und .o Dateien im Aufruf mischen: es würden dann erst alle .c 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 sollen nur die Teile neu kompilieren werden, die sich geändert haben!
    - Insbesondere sollen die ganzen Standardfunktionen (z.B. printf) nicht jedes Mal neu kompiliert werden
  - 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. die Standardbibliothek mit der Definition von printf.

    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 .c Datei in der die main Funktion steht

    cc –o ApproximationOfPiMain …

    cc –o ApproximationOfPiTest …

Header Dateien 1/6

UNI FREIBURG

- Header Dateien, Motivation
  - Jede Funktion muss vor der Benutzung deklariert werden

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

    #include "./ApproximationOfPi.c"

    Dann wird die Funktion **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:

    ApproximationOfPi**.h** nur mit der Deklaration

    ApproximationOfPi**.c** mit der Implementierung
  - Die .h Datei mit der Deklaration brauchen wir für **Main** und für **Test**:

    #include "./ApproximationOfPi.h"
  - Die .c Datei brauchen wir nur einmal und wollen wir auch nur einmal kompilieren
  - cc **–c** ApproximationOfPi.c

Header Dateien 3a/6

UNI FREIBURG

- Header Dateien, Details
  - Kommentare zur Verwendung der Funktionen nur an einer Stelle und zwar in der .h Datei

    In der .c Datei schreiben wir statt einem Kommentar:

    // _____

    Bei Kommentar in der .h Datei **und** in der .cpp Datei käme es bei Änderungen unweigerlich zu Inkonsistenzen
  - Implementierungsdetails natürlich in .c kommentieren!

Header Dateien 3b/6

UNI FREIBURG

- Header Dateien, Details
  - In der .c Datei die zugehörige .h Datei includen; dadurch kann der Compiler die Konsistenz zwischen Deklaration (in .h) und Funktionsdefinition (in .c) prüfen
  - In jeder Datei nur **genau** das **direkt** includen, was in der Datei gebraucht wird
  - Insbesondere **keine indirekten includes** (durch includes in einer inkludierten Datei)
  - Achtung: Systemheader wie stdio.h oder stddef.h müssen meist auch in .h Dateien includiert werden

Header Dateien 4/6

UNI FREIBURG

- 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 darf xxx.h nicht nochmal gelesen werden, 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 "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 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=. …
```

Anpassungen Makefile 1/6

UNI FREIBURG

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

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

  - Nehmen wir an, wir ändern ApproximationOfPiMain.c
  - Jetzt muss nur ApproximationOfPiMain.o neu erzeugt werden und damit ApproximationOfPiMain neu gelinkt

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

    Das kann es in der Tat, siehe nächste Folien

Anpassungen Makefile 2/6

UNI FREIBURG

- Abhängigkeiten, Realisierung
  - Im Makefile werden **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
  - Im Makefile werden **Abhängigkeiten** angeben:

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

  - Wenn die Abhängigkeiten alle befriedigt sind, 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

UNI FREIBURG

- Automatische Regeln
  - Make hat jede Menge automatische Regeln

    Zum Beispiel, wie eine .o Datei aus einer .c Datei erzeugt wird, nämlich mit cc -c …
  - Diese automatische Regeln können abgeschaltet werden, indem das Target

    .SUFFIXES:
  - Zu Beginn ins Makefile geschrieben wird

Anpassungen Makefile 5/6

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 = künstlich

    Alle Jenkins targets, die wir in der Übung benutzt haben (compile, checkstyle, test, clean) waren "phony"

    Phony targets dienen als Abkürzung für eine Abfolge von Kommandos
  - Die Kommandos zu einem phony target werden immer ausgeführt

Anpassungen Makefile 6/6

UNI FREIBURG

- Beispiel: Bauen des Main Programmes

```
DoofMain:  DoofMain.o  Doof.o
   cc -o  DoofMain  DoofMain.o  Doof.o        (1)

DoofMain.o:  DoofMain.c
   cc -c  DoofMain.c                          (2)

Doof.o:  Doof.c
   cc  -c  Doof.c                             (3)
```

  - Eine Änderung an Doof.c und nachfolgendes make DoofMain bewirkt Folgendes:

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

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

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

Felder 1/5 (Hauptspeicher, konzeptuell)

UNI FREIBURG

- Der **Hauptspeicher** eines Rechners ist eine Menge von Speicherzellen
- Jede Speicherzelle fasst **1 Byte = 8 Bits**

  Also eine Zahl zwischen 0 und 255 (einschließlich)
- Die Speicherzellen sind fortlaufend nummeriert innerhalb von Bereichen, die von der Architektur und/oder dem Betriebssystem vorgegeben sind
- Die Nummer einer Speicherzelle ist ihre Adresse

Felder 2/5 (Variablen)

UNI FREIBURG

- **Variablen** sind Namen für ein Stück Speicher, z.B.

```
int x = 12;  // One int (typically 4 bytes).
```

- Je nach Typ umfasst die Variable ein oder mehrere Bytes ... diese Anzahl bekommt man mit sizeof:

```
printf("%zu\n", sizeof(x));  // Use %zu for type size_t.
```

- Eine Verwendung der Variablen in einem Ausdruck („**rechts** vom =“) steht für den **Wert** in diesen Speicherzellen, interpretiert gemäß Typ
- Eine Verwendung der Variablen links vom „=„ in einer Zuweisung steht für die **Adresse** selbst

Felder 3/5 (Zugriff)

UNI FREIBURG

- **Felder** sind Folgen von Variablen vom gleichen Typ, auf die mit demselben Namen und einem sogenannten Index zugegriffen werden kann
- Zugriff auf ein Element des Feldes mit dem [] Operator, wobei das erste Element Index 0 hat, das zweite 1, usw.

```
int a[10];                      // Array of 10 integers.
printf("%zu\n", sizeof(a));     // Prints 4 * 10 = 40.
printf("%d\n", a[2]);           // Prints third element.
```

- Die Elemente stehen **hintereinander** im Speicher
- Der Feldname steht immer für die **Adresse** (nicht den Wert) des ersten Elementes

Felder 4/5 (Rechts-, Linkswert, Grenze)

UNI FREIBURG

- **Rechts** von = steht a[i] für den Wert der i-ten Variable im Feld

```
printf("%d\n", a[2]);                    // Prints third element.
```

- **Links** von = steht a[i] für die Adresse der i-ten Variable im Feld

```
a[2] = 42;              // Assigns to third element.
```

- **Achtung „buffer overflow“:**
  - Ein Zugriff über die deklarierte Feldgröße hinaus liefert ein undefiniertes Ergebnis
  - Eine Zuweisung über die deklarierte Feldgröße hinaus kann das Programm zum Absturz bringen!!!

Felder 5/5 (Initialisierung)

UNI FREIBURG

- Feldelemente können schon bei der Deklaration Werte erhalten

```
int a[3] = {1, 2, 3};
int a[3] = {1, 2, 3, 4};  // Too many values -> compiler error.
int a[3] = {1, 2};        // Missing values initialized to zero!
int a[3] = {0};           // Initializes all elements to zero.
```

- **Achtung:** ohne Initialisierung ist der Inhalt der betreffenden Speicherzellen laut C/C++ Standard beliebig
- Daher immer Initialisierung sicherstellen!

Literatur / Links

- Compiler und Linker
  - Online Manuale zum cc

    http://gcc.gnu.org/onlinedocs

    Oder von der Kommandozeile: man cc
  - Wikipedias Erklärung zu Compiler und Linker

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

    http://en.wikipedia.org/wiki/Linker_(computing)
- Felder
  - https://en.wikibooks.org/wiki/C_Programming/Arrays_and_strings
  - https://www.tutorialspoint.com/cprogramming/c_arrays.htm