Programmieren in C

SS 2021

Vorlesung 2, Dienstag 27. April 2021

(Compiler und Linker, Bibliotheken)

Prof. Dr. Peter Thiemann
Lehrstuhl für Programmiersprachen
Institut für Informatik
Universität Freiburg
(nach Folienvorlagen von Prof. Dr. Hannah Bast)

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Hinweise — Korrektur, Copyright
- Inhalt
  - Testen mit unity
  - Compiler und Linker — was + warum
  - Header Dateien — Trennung in .h und .c Dateien
  - Besseres Makefile — Abhängigkeiten
  - Live Programm — Arrays (Felder)
  - **Ü2:** Programme vom Ü1 sauber in .h und .c Dateien zerlegen, Makefile geeignet anpassen, Arrays

Erfahrungen mit dem Ü1

UNI FREIBURG

- 264 Abgaben, 244 Erfahrungen, 235 mit lesbarer Zeit
- Zeitstatistik (in Stunden)

Min	Q1	Med	Q3	Max	Avg	MD	Var
0.33	2.0	3.75	5.0	25.25	4.03	1.89	8.69

- Regexp-Stats

#	Schlüsselworte
37	schwer|schwierig|anspruch|aufwendig|fordernd|hart|viel zeit|lange gedauert
4	nicht schwer|nicht schwierig|nicht anspruch|nicht aufwendig|unaufwendig|nicht fordernd|nicht hart|nicht viel zeit|nicht lange gedauert
111	cool|nett|spaß|gut|schön|toll|super
2	nicht cool|uncool|nicht nett|keinen spaß|nicht gut|nicht schön|unschön|nicht toll|nicht super
6	unklar|verwirrend

Erfahrungen mit dem Ü1

UNI FREIBURG

- „Das ganze Linux/git Zeug zum ersten Mal machen ist ziemlich verwirrend."
- "es war etwas schwer, Ubuntu zum laufen zu bringen."
- Aber "Das Übungsblatt hat es aber sehr ausführlich und gut erklärt und somit gab es kaum Probleme."
- "…Windows Laptop. Einrichtung von Linux-Subsystem hat lange gedauert, nachdem es mit einem Linux-Image über VirtualBox nicht geklappt hat."
- Aber "Erklärungsvideos zu Linux/Shell und GIT sind aber super hilfreich gewesen!
- "Ziemlich schwer am Anfang. Braucht Zeit zum Angewöhnen."
- Aber "Zeitaufwand 4:00"

Korrekturen Ihrer Abgaben

UNI FREIBURG

- Ablauf
  - Ihnen wurde bereits ein Tutor zugewiesen, der/die Ihre Abgabe im Laufe der Woche korrigiert
  - Sie bekommen folgendes Feedback
    - Ggf. Infos zu Punktabzügen
    - Ggf. Hinweise, was man besser machen könnte
  - Machen Sie in Ihrer Arbeitskopie (egal wo)

    `git pull`
  - Das Feedback finden Sie jeweils in

    `blatt-<xx>/README.md`
  - Oder verwenden Sie das Web-Frontend

Copyright

UNI FREIBURG

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

```
/* Copyright 2021 University of Freiburg
 * Author: Peter Thiemann thiemann@informatik.uni-freiburg.de
 */
```

  - Nicht nachmachen, Ihr Code gehört Ihnen!
  - 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 unity

UNI FREIBURG

- Unity ist ein Framework für Unittests
  - Verwendung in approximationOfPiTest.c

    ```
    #include <../unity/unity.h>
    ```

  - Dann in der main() Funktion die Testfälle starten

    ```
    int main (void) {
      UnityBegin();
      RUN_TEST (test_count_points_in_circle);
      return UnityEnd();
    }
    ```

  - Am Ende gibt es einen Fehlerbericht mit Zeilennummer, Dateiname und den verglichenen Werten

Compiler und Linker 1/5

UNI FREIBURG

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

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

    Ergebnis: eine Objektdatei namens <name>.o

    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 vorher kompilierte .o Dateien zu einem ausführbaren 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, 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" (fehlendes 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 wird eine nach der anderen kompiliert, die resultierenden .o Dateien gelinkt und dann gelöscht

    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

    --> kein guter Stil

Compiler und Linker 4/5

Bei wenig Code
natürlich kein Problem

- Warum die Unterscheidung
  - **Grund:** Code kann sehr umfangreich sein, das Compilieren kann dauern, aber Änderungen sind oft inkrementell
    - Es sollen nur die Teile neu kompiliert werden, die sich geändert haben!
    - Insbesondere sollen die ganzen Standardfunktionen (z.B. printf) nicht jedes Mal neu kompiliert werden
  - Bisher hatten wir den Code nach jeder Änderung von Grund auf neu kompiliert
  - Auch da schon "vorkompilierte" Sachen "dazu gelinkt", z.B. die Standardbibliothek mit der Definition von printf.

    Heute mehr dazu!

Compiler und Linker 5/5

UNI FREIBURG

- Name des ausführbaren Programms
  - Ohne weitere Angaben heißt das Programm

    a.out (kurz für „assembler output“)
  - Mit der –o Option kann es benannt werden

    Konvention: das Programm heiße 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 grid_points_inside()…
  - 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
  - Das ist redundant!

Header Dateien 2/6

UNI FREIBURG

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

    ApproximationOfPi**.h** nur mit der Deklaration

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

    #include "./ApproximationOfPi.h"
  - Die zugehörige .c Datei soll nur einmal kompiliert werden
  - 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 .c Datei käme es bei Änderungen unweigerlich zu Inkonsistenzen
  - Implementierungsdetails natürlich in .c kommentieren!

Header Dateien 3b/6

UNI FREIBURG

- Header Dateien, Details
  - Die .c Datei inkludiert die zugehörige .h Datei; dadurch kann der Compiler die Konsistenz zwischen Deklaration (in .h) und Funktionsdefinition (in .c) prüfen
  - **Genau** das und nur das **direkt** inkludieren, 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 inkludiert werden

Header Dateien 3/6

UNI FREIBURG

- Header Dateien, Implementierung
  - Vor dem Compiler läuft der C-Präprozessor
  - Der Compiler selbst läuft auf der Ausgabe des Präprozessors
  - Der Präprozessor kopiert alle Zeilen der Eingabe, die **nicht** mit # beginnen.
  - Zeilen, die mit # beginnen, werden interpretiert:
  - `#include "./ApproximationOfPi.h"`
  - Der Präprozessor verarbeitet nun ./ApproximationOfPi.h und kehrt dann zur Folgezeile zurück
  - Das Inkludieren von Dateien kann geschachtelt sein

Header Dateien 4/6

UNI FREIBURG

- Header Guards, Motivation
  - Eine Header Datei kann eine andere inkludieren
  - Bei komplexerem Code ist das sogar die Regel
  - Zyklische Abhängigkeiten verhindern! Beispiel:
    - Datei xxx.h inkludiert Datei yyy.h
    - Datei yyy.h inkludiert Datei zzz.h
    - Datei zzz.h inkludiert Datei xxx.h

    An dieser Stelle darf xxx.h nicht nochmal gelesen werden, sonst terminiert der Präprozessor nicht

Header Dateien 5/6

UNI FREIBURG

- Header Guards, Implementierung
  - Der Präprozessor kann selbst Variable definieren und einfache logische Ausdrücke auswerten:

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

  - Wenn der Präprozessor die Datei das erste Mal sieht, wird dabei die Variable XXX definiert

    Diese Variable nennt man "Header Guard"

  - Wenn der Präprozessor die Datei ein weiteres Mal liest, wird der Inhalt übersprungen

Header Dateien 6/6

UNI FREIBURG

- Header Guards, Benennung der Variablen
  - Der Name der Header Guard Variablen sollte möglichst eindeutig gewählt werden
  - Konvention: Pfad + Dateiname, z.B.

```
#ifndef APPROXIMATIONOFPI_H_
#define APPROXIMATIONOFPI_H_
...
#endif  // APPROXIMATIONOFPI_H_
```

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 grid_points_inside…

  - Wenn sich ApproximationOfPiMain.c ändert, …
  - muss nur ApproximationOfPiMain.o neu erzeugt werden und danach nur ApproximationOfPiMain neu gelinkt werden

    Der Rest braucht nicht neu kompiliert / gelinkt zu werden
  - Wir können make beibringen, das zu automatisieren!

Anpassungen Makefile 2/6

UNI FREIBURG

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

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

  - Bei make <target> geschieht Folgendes:

    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 alle Abhängigkeiten befriedigt sind, werden folgende Bedingungen geprüft:
    - Es existiert bereits eine Datei mit Namen <target>
    - Es existieren Dateien <dependency 1>, <dependency 2>, …
    - <target> ist neuer als alle <dependency i>
  - **Nur wenn eine der Bedingungen nicht erfüllt ist**, werden die Kommandos <command1>, <command2>, … ausgeführt

Anpassungen Makefile 4/6

UNI FREIBURG

- Impliziete Regeln
  - Make hat jede Menge implizite Regeln

    Zum Beispiel, wie eine .o Datei aus einer .c Datei erzeugt wird, nämlich mit

    ```
    $(CC) $(CPPFLAGS) $(CFLAGS) –c …
    ```

  - Diese automatische Regeln können abgeschaltet werden, indem zu Beginn des Makefile das Target

    ```
    .SUFFIXES:
    ```

  - Oder indem Make mit der Option –r gestartet wird

    ```
    make –r
    ```

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

    Phony targets dienen als Abkürzung für eine Abfolge von Kommandos

    .PHONY: compile checkstyle test clean

  - 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
   gcc -o  DoofMain  DoofMain.o  Doof.o        (1)

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

Doof.o:  Doof.c
   gcc  -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 liefert die Funktion 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** (Arrays) 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 ints.
printf("%zu\n", sizeof(a));     // Prints 4 * 10 = 40.
printf("%d\n", a[2]);           // Prints third element.
```

- Die Elemente stehen **direkt hintereinander** im Speicher
- Der Feldname (hier a) steht immer für die **Adresse** (nicht den Wert) des ersten Elementes
- Die Größe des Feldes (hier 10) ist **fest**! Sie ändert sich während der Ausführung nicht.

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 initialisiert werten (d.h. 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