IFM 3.3 Systemprogrammierung, W19

Regeln zur Bearbeitung der Praktikumsaufgaben

Carsten Gips, Birgit C. George, FH Bielefeld

Die folgenden Regeln gelten ab Blatt 02 für *alle* Praktikumsaufgaben.

Diese Regeln stellen typische Mindestanforderungen für Programmierer im Arbeitsalltag in der Wirtschaft oder auch in Open-Source-Projekten dar und sollen Sie im Praktikum ermuntern, auf saubere Programmierung zu achten. Zum anderen sollen die Regeln aber auch die Abnahme und die Beurteilung Ihrer Lösungen im Praktikum erleichtern.

Nichteinhalten der formalen Kriterien führt zu Punktabzug.

1 Vorgaben

Die Vorgaben zu den Aufgabenblättern finden Sie unter

```
git@git01-ifm-min.ad.fh-bielefeld.de:cagix/sp-vorgaben.git
```

Forken Sie dieses Repository und nutzen Sie es als Projekt für *alle* Aufgaben. Aktualisieren Sie Ihren Fork regelmäßig, da die Vorgaben schrittweise im Laufe des Semesters erarbeitet/verbessert und bereitgestellt werden.

Fehler, Fragen und Hinweise können Sie im Issue-Tracker des Vorgabe-Repos melden, Verbesserungsvorschläge können gern als Merge-Request gegen das Vorgabe-Repo gestellt werden.

2 Ordnerstrukturen

Ihr Projekt enthält die Ordner `src/`, `include/`, `lib/`, `tests/`, `bin/`, `res/` und `doc/`. Für jedes Teilprojekt, beispielsweise die Ansteuerung der LED-Segment-Anzeige, gibt es je einen entsprechenden Unterordner unterhalb von `src/`, `include/` und `tests/`. Die Namen der Teilprojekt-Unterordner werden auf den Aufgabenblättern spezifiziert.

Die einzelnen Dateien werden nach folgenden Regeln auf diese Ordner verteilt:

- Alle Header-Dateien (`.h`-Dateien) liegen im passenden Teilprojekt-Unterordner unter `include/`.
- Alle Sourcen (`.c`- und `.cpp`-Dateien) liegen im jeweiligen Teilprojekt-Unterordner unter `src/`.
- Kompilierte Programme werden nach `bin/` „installiert“ (kopiert).
- Erzeugte Bibliotheken werden nach `lib` „installiert“ (kopiert).
- Doxygen-Dokumentation wird nach `doc/` generiert.
- Sonstige Ressourcen (beispielsweise Datenbanken oder HTML-Seiten für Webserver) liegen unter `res/`.

Hinweis: Alle Test-Header und -Sourcen liegen zusammen mit einem eigenen Test-Makefile in den jeweiligen Teilprojekt-Unterordnern im `tests/`-Ordner (s.u.).

3 Trennung `.h` und `.c`

Die Trennung von Header und Implementierung ist obligatorisch:

- Deklarationen von Funktionen und Klassen sowie Definitionen von Datentypen gehören in die `.h`-Dateien.
- Sämtliche Implementierungen kommen in die `.c`- bzw. `.cpp`-Dateien.
- Die Namen der `.h`-, `.c`- bzw. `.cpp`-Dateien für die Teilprojekte sowie die Aufteilung der Funktionalitäten werden in der Aufgabenstellung vorgegeben.

Beachten Sie zusätzlich die folgenden Regeln:

- Hilfsfunktionen dürfen nicht von außen sichtbar sein:
  1. Signaturen nicht in der Headerdatei wiederholen, und
  2. Sichtbarkeit auf die jeweilige Implementierungsdatei (`.c`/`.cpp`) beschränken.

- Header-Dateien sollen ausschließlich Schnittstellen-relevante Deklarationen bzw. Definitionen enthalten. Es dürfen zusätzlich nur die in der Header-Datei selbst benötigten `#include` enthalten sein.
  [In der Regel gehören die in der Aufgabenstellung genannten Funktionen und Typen zur Schnittstelle.].hinweis
- Alle Header-Dateien müssen sich von C++ aus nutzen lassen (vgl. VL03)!
- `main()`-Funktion: In den Implementierungsdateien (`.c`/`.cpp`) darf es wegen der Einbindung in die Testsuiten keine `main()`-Funktion geben! Definieren Sie für Ihre Projekte separate Demo-Programme und benennen Sie diese nach dem Schema „`demo_xyz.c`“ für Projekt (-teil) „xyz“. In diesen Dateien darf es neben der `main()`-Funktion keine weiteren Definitionen/Deklarationen geben.

4 Makefile mit Standardtargets

Für den typischen „Unix-Dreisprung“ orientieren wir uns an traditionell vorhandenen Targets. Das Haupt-Makefile im Projektordner soll deshalb folgende Standardtargets aufweisen:

- `all`: Kompiliert alle geforderten Programme und Bibliotheken in den Teilprojekt-Unterordnern unter `src/`
- `install`: Kopiert die durch `all` erzeugten Programme nach `bin/` und die Bibliotheken nach `lib/`
- `test`: Ruft die `test*`-Targets in den `tests/`-Makefiles (s.u.) auf
- `run`: Lässt die in `bin/` installierten Programme laufen
- `doc`: Generiert mit Doxygen die Dokumentation zu Ihrem Programm in den Unterordner `doc/`
- `clean`: Entfernt alle temporären (generierten) Dateien in den Teilprojekt-Unterordnern unter `src/`
- `distclean`: Wie `clean`, entfernt zusätzlich die `bin/-`, `lib/-` und `doc/`-Ordner

Achten Sie auf die korrekte Angabe der Abhängigkeiten, d.h. es soll nur dann kompiliert werden, wenn die Zieldatei fehlt oder veraltet ist (d.h. sich eine Abhängigkeit geändert hat). Die Targets müssen eigenständig aufgerufen werden können.

Legen Sie sich in den Teilprojekt-Unterordnern unter `src/` je ein Hilfs-Makefile an, welches durch das Haupt-Makefile aufgerufen wird und welches die Kompilier- und Aufräum-Vorgänge im Teilprojekt-Unterordner unter `src/` erledigt.

5 Dokumentation der Lösung mit Doxygen

Der Ihnen erstellte Quellcode muss mit **Doxygen**[1] dokumentiert sein.

Alle globalen Variablen/Konstanten und alle Makros, Typen und Funktionen sowie in C++ alle Klassen und Methoden (inkl. Konstruktoren, Copy-Konstruktoren, Destruktoren) müssen in der Dokumentation so erklärt werden, dass man versteht, was diese tun und wie sie genutzt werden, d.h. welche Parameter ggf. erwartet werden und welche nicht zulässig sind und welche Werte ggf. zurückgeliefert werden. Bei globalen Variablen/Konstanten muss klar werden, wo sie genau verwendet werden und wofür.

Im Kommentar muss erklärt werden, **was** eine Funktion/Methode/Klasse macht. „Lesen“ Sie dabei nicht den Code vor, d.h. schreiben Sie *nicht*, *wie* diese Funktionalität erreicht wird.

Nutzen Sie interne Verlinkung, um auf die Nutzung weiterer Funktionen hinzuweisen.

6 Debugger, keine Hilfsausgaben auf der Konsole

Nutzen Sie den GDB zum Debuggen Ihrer Programme! Verzichten Sie nach Möglichkeit auf Hilfsausgaben auf der Konsole.

Ab Blatt 03 dürfen bei Nutzung des Schalters `-DNDEBUG` beim Kompilieren **keinerlei** Hilfsausgaben mehr auf der Konsole o.ä. gemacht werden! Definieren Sie sich dazu ein entsprechendes Präprozessormakro und nutzen Sie es (wenn Sie überhaupt noch Debug-Ausgaben einsetzen).

[1] doxygen.org

7 Sonstiges: Kompilierbarkeit, Tests, Speicherlöcher, ...

7.1 Einheitliche Formatierung

Verwenden Sie eine einheitliche Formatierung Ihres gesamten Codes (zB. K&R).

Auch wenn Sie mit Geany oder einem einfachen Texteditor arbeiten, sollten Sie auf eine vernünftige Formatierung achten! Mit Eclipse CDT oder JetBrains Clion können Sie spätestens zur Abgabe den Code leicht in eine einheitliche Formatierung bringen.

7.2 Abgegebenes Projekt muss kompilieren

Ihr abgegebenes Projekt muss kompilieren!

- Dabei dürfen keine Fehler oder Warnungen (`-Wall -pedantic`) auftreten
- Bei Aktivierung des jeweils vorgegebenen Sprachstandards (C11 o.ä.) dürfen keine Warnungen oder Fehler auftreten
- Die Projektwurzel muss frei verschiebbar sein, es darf keine Abhängigkeit zu absoluten Pfaden geben!

7.3 Tests

Über die Makefiles in den Teilprojekt-Unterordnern unter `tests/` stehen Ihnen Test-Targets (siehe Aufgabenblätter) für die Teilprojekte zur Verfügung. Dabei werden jeweils die auf dem Aufgabenblatt spezifizierten Dateien mit Ihrer Lösung kompiliert und zusammen mit den Vorgabetests durchgeführt.

Zusätzlich gibt es jeweils die Targets `abgabe` (wie die Test-Targets, plus Kompilierung mit `-DABGABE` und Ausführung mit der LeakCheck-Bibliothek) und `clean` (löscht alle in den Teilprojekt-Unterordnern unter `tests/` generierten temporären Dateien).

- Die Vorgabe-Tests sollten fehlerfrei durchlaufen! Betrachten Sie die Testfälle als „ausführbare Spezifikation“, die ggf. die Aufgabenstellung ergänzt und präzisiert.
- Die Test-Targets sollten wegen der eventuellen Nutzung der wiringPi-Bibliothek immer mit `sudo` aufgerufen werden, also beispielsweise `sudo make abgabe`.
- Das Abgabe-Target aktiviert ggf. zusätzliche Tests. Einige davon sind nur eine Erinnerung, in der Abnahme bestimmte Dinge manuell zu prüfen, und schlagen deshalb *immer* fehl.
- Beim Wechsel zwischen Test- und Abgabe-Target muss jeweils einmal `clean` ausgeführt werden.

7.4 Sonstiges

- **Alle** von Ihnen belegten Ressourcen auf dem Heap-Speicher sind wieder **freizugeben**, sobald diese nicht mehr benötigt werden! `make abgabe` darf keine Speicherlöcher finden.
- Achten Sie auf die Prüfung der Parameter: Ihre Funktionen und Methoden müssen fehlertolerant arbeiten und falsche Eingabewerte zurückweisen. Insbesondere muss *sinnvoll* auf Null-Pointer reagiert werden.
- Verwenden Sie **keine globalen Variablen** (außer es wird in der Aufgabenstellung explizit verlangt)!

8 Abzüge bei Nichteinhaltung der Regeln

- Kein funktionsfähiges Makefile mit Standard-Targets: bis zu 4P
- Code lässt sich nicht kompilieren (`make all`): bis zu 6P
- falls Code kompilierbar:
  - Tests kompilieren nicht (`make test`, `make abgabe`): bis zu 6P
  - Warnungen bei `-std=XXX -Wall -pedantic` (C/C++-Standard gemäß Blatt): bis zu 2P
- Keine einheitliche Formatierung des gesamten Codes (zB. K&R): bis zu 4P
- Keine ordentliche Doxygen-Dokumentation: bis zu 2P
- Nichtbeachtung der Ordnerstrukturen und/oder Trennung von `.h` und `.c`: bis zu 2P

Cut-Off: Insgesamt werden pro Blatt maximal 10 Punkte für die Nichteinhaltung der Regeln abgezogen.