Universität Hamburg
Fachbereich Informatik
Wissenschaftliches Rechnen
Prof. T. Ludwig, H. Lenhart

Übungsblatt 0 zum Praktikum
Paralleles Programmieren für
Geowissenschaftler im SS 2015
Abgabe: keine

---

1 Grundlegendes

Die Übungsaufgaben dienen der Vertiefung des Stoffes und dem Feedback für die praktische Anwendug des Gelernten.

Es gibt zu jeder Veranstaltung einen Zettel mit Übungsaufgaben und zu jeder Übungsaufgabe wird das Abgabedatum vermerkt; abgegeben werden können sie jeweils bis 23:59 Uhr.

Zusätzlich muss jeder Studierende mindestens zweimal eine Lösung während des Praktikums präsentieren. Unzureichende Präsentation der Lösungen können Punktabzüge zur Folge haben. Der Besuch der Veranstaltung ist Pflicht! Maximal sind 2 Termine ohne vorherige Abmeldung (z.B. Krankheit, Praktikum) erlaubt. Weiteres Fernbleiben von den Veranstaltungen kann zu Punktabzügen führen.

1.1 Mailingliste

Falls Sie Fragen zu den Übungsaufgaben haben, dann schreiben Sie Ihre Fragen bitte direkt an die Mailingliste. Die Erfahrung zeigt, dass viele Gruppen unnötig Zeit verlieren, weil sie an eigentlich einfachen Fragen hängenbleiben.

Die Mailingliste der Vorlesung lautet:

`PPG-15@wr.informatik.uni-hamburg.de`

Uns ist wichtig, dass Sie Ihre Zeit nicht auf zeitraubende Nebensachen verwenden, sondern auf relevante Fragen.

1.2 Gruppenarbeit

Abgaben sollten in Gruppen von zwei bis drei Personen erfolgen. Natürlich können Sie auch in größeren Gruppen zusammenarbeiten, mehrere Gruppen dürfen jedoch nicht identische Lösungen abgeben. Offensichtliche Kopien von Lösungen werden wir nicht akzeptieren und mit 0 Punkten bewerten.

2 Abgabe

Die Vergabe der Punkte richtet sich nach folgende Kriterien.

- Pünktlichkeit der Abgabe
- Vollständigkeit der geforderten Materialien
- Korrektheit der Ergebnisse
- Lauffähigkeit des Binärprogrammes
- Strukturierter, gut dokumentierter Programmcode
  (Wir erwarten, dass Ihr Euch an unseren Styleguide haltet)
- Korrektheit der Datei- und Pfadnamen

3 Styleguide

3.1 Warum ein Styleguide?

Programmcode ist Sprache. Er dient der Kommunikation. Nicht nur der Kommunikation mit dem Computer sondern, und das ist insbesondere in den Geowissenschaften der Fall, auch zur Kommunikation zwischen Menschen. Denn im Programmcode werden Modelle ausgedrückt, die von anderen Menschen gelesen, bewertet und verbessert werden. Will man also viele Menschen mit seinem Code beeinflussen, muss der Code klar formuliert sein. Und dabei möchte dieser Styleguide helfen.

Menschen sind nicht perfekt. Sie machen Fehler. Alle. Das ist oft nicht weiter schlimm. Schlimm ist aber, wenn Fehler nicht korrigiert werden können, weil sich kein Mensch mehr in dem bestehenden Programmcode zurechtfindet. Und die Geschichte der Informatik zeigt leider sehr eindrücklich, dass immer wieder große Teile von Code komplett weggeschmissen werden und komplett neugeschrieben werden mussten, weil kein Mensch mehr in der Lage war, einfache Fehler zu korrigieren, ohne dabei mehr neue Fehler zu produzieren, als er behoben hat. Davor möchte dieser Styleguide schützen.

Schließlich, und das ist eigentlich der nebensächlichste Aspekt, ist dieser Styleguide auch ein Weg der Kommunikation zwischen uns Lehrenden und euch Studenten, um in kurzer, knapper Form zusammenzufassen, was wir von dem von euch produzierten Code erwarten. Nebensächlich bedeutet, dass dieser Styleguide nicht existiert, um zu sagen: "Aber hier hast Du gegen Paragraph 5.7.1 verstoßen, darum gibt es zehn Punkte Abzug", sondern um euch zu helfen, guten Code zu schreiben. Trotzdem werden wir euch selbstverständlich für nicht konformen Code Punkte abziehen.

Teile dieses Styleguides sind reine Geschmacksfragen, wie z. B. die Forderung, Einrückungen nur mit Tabs zu machen. (Trotzdem sind sie meist sehr wohl begründet, im Fall der Tabs dadurch, dass jeder Leser den Programmcode mit der Einrücktiefe sehen können soll, die er bevorzugt, indem er einfach die Tabulatorbreite in seinem Editor ändert. Nach der Änderung sollte der Code genauso perfekt eingerückt sein wie vorher. Aus dieser Überlegung motivieren sich z. B. die Details zum Einrücken von Folgezeilen.) Wie ihr diese Regeln anwendet, wenn ihr nicht für unser Seminar programmiert, ist eure Sache, aber ihr tut gut daran, euch einmal zu überlegen, wie ihr es handhaben wollt, und dabei zu bleiben.

Andere Teile geben einen breiten Konsens innerhalb der Informatik wieder, der Punkt zu den globalen Variablen gehört dazu. Da Informatiker die Menschen sind, die erstens am meisten mit Code zu tun haben und sich zweitens am explizitesten mit Fragen der Codequalität auseinandersetzen, ist es nicht verwunderlich, dass Informatiker an einigen Punkten bereits Antworten auf Fragen gefunden haben, die in anderen Fachbereichen noch gar nicht gestellt wurden. Das führt z. B. dazu, dass ihr, unabhängig davon, dass sich jeder, der mit globalen Variablen arbeitet, ständig selber Beine stellt, auf viele Geowissenschaftler treffen werdet, die sagen: "Aber ohne globale Variablen kann ich nicht arbeiten! Wie soll das denn gehen? " Die Wahrheit ist, dass keiner, der das sagt, es je selbst probiert hat. Sonst wüsste er, dass seine Aussage falsch ist. Und er wüsste auch, dass sein Programmcode ohne globale Variablen viel besser zu warten ist. Es gibt viel schlechten Code in der Welt, doch viel schlechter Code macht euren Code nicht besser. Daher ist unser Rat: Wenn ihr mit solchen Aussagen konfrontiert werdet, ignoriert sie und macht es besser!

3.2 Strukturelemente

3.2.1 Whitespace:

Einrückung mittels Tabs (ein Tab pro Einrückungsebene)

Alles, was in einem Block steht, der mit "end XXX" abgeschlossen wird, wird eingerückt.

Öffnende und schließende Zeilen (die mit if, do, else, end etc.) werden nicht eingerückt.

Präprozessorcode wird genauso gehandhabt wie Fortrancode, allerdings muss für die meisten Compiler das das "#" am Anfang der Zeile stehen. Der Rest sollte der Einrücktiefe des Blocks entsprechen.

```
↦       if(xxx) then
# ↦     ↦       ifdef SWITCH
↦       ↦       ↦       ! optionaler Code
# ↦     ↦       endif
↦       endif
```

Fortsetzungszeilen werden mit Tabs bis zur aktuellen Ebene eingerückt, dort steht das "&". Anschließend werden sie mit mindestens drei weiteren Spaces an der fortzusetzenden Zeile ausgerichtet (gilt nicht, wenn der Umbruch sich in einem String befindet). An dieser Stelle wird die Kombination von Tabs und Spaces verwendet, damit beim Umstellen der Tabbreite, die Ausrichtung an der fortzusetzenden Zeile erhalten bleibt.

Beispiel:

```
↦       result = foo(argument1, argument2, argument3, &
↦       &               argument4, argument5, argument6)
```

Kommentare gehören zum Code und werden entsprechend eingerückt. Das "!" steht an der Stelle, an der sonst der Code beginnen würde.

Zeilenendkommentare werden mit einem Tab vom Code getrennt.

3.2.2 Bezeichner:

Alle Bezeichner werden aus einem oder mehreren vollständigen(!) Worten zusammengesetzt.
Sie beschreiben deutlich, welchen Zweck die Variable/Subroutine erfüllt.
Die Teilwörter werden entweder durch Unterstriche oder CamelCase voneinander abgegrenzt.
Es werden keine Abkürzungen verwendet, außer den folgenden: prev, dest, temp
Einzelne Buchstaben als Bezeichner sind ausschließlich bei reinen Schleifenvariablen statthaft.
Programmparameter beginnen mit einem kleinen "k" (= konstant).

3.2.3 Codegliederung:

Ein Modul = eine Datei.

Globale Variablen sind verboten.

Modulvariablen müssen als private deklariert werden.

Bemerkung:
Dieser Forderung liegt die Erkenntnis zugrunde, dass der Aufwand, die Bedeutung einer Variablen zu ändern, überproportional zunimmt mit der Anzahl der Stellen, die sie referenzieren. Und diese Komplexitätszunahme hat zudem zwei massive Sprünge: Den ersten, wenn die Variable über mehrerer Subroutinen hinweg benutzt wird, und den zweiten, wenn sie in mehreren Modulen benutzt wird. Leider ist es nicht immer praktikabel, Modulvariablen zu vermeiden (man sollte es aber so weit wie möglich tun), aber es ist immer sinnvoll, nicht eine einzige globale Variable zu verwenden.

In der Informatik werden für diese Entwicklungsmethodik die Begriffe der Kapselung und der Entkoppelung verwendet: Alles, was zur Bereitstellung zusammengehöriger Funktionalität gebraucht wird, wird in einem eigenen kleinen Bereich des Codes gekapselt und so weit wie möglich von anderen Codeteilen entkoppelt. Je besser das gelingt, desto leichter ist es die Strukturen in dem gekapselten Bereich zu ändern, ohne an tausend anderen Stellen im Programmcode Fehler zu generieren. Kapselung und Entkoppelung sind auch Grundanliegen der objektorientierten Programmierung (das ist die Methodik die praktisch jeder modernen Softwareentwicklung zugrunde liegt). Aus genau den oben genannten Gründen. Wir wollen hier nicht so weit gehen, von euch objektorientierte Programmierung zu fordern, aber auf ein gewisses Kapselungsgebot bestehen wir.

Kein Stück Code darf länger als 50 Zeilen sein. (Gilt für "program", "subroutine" und "function" Blöcke.)
Das umschließt Zeilen mit "program"/"subroutine"/"function" und "end" und alle Zeilen dazwischen.
Leerzeilen & Kommentare werden auch mitgezählt. Alles, was sonst länger werden würde, ist sinnvoll zu unterteilen.

Bemerkungen:
Die Motivation ist, dass jede Subroutine als kleinste Struktureinheit eures Codes vollständig auf einen (und nicht nur euren) Bildschirm passen sollte, und ohne Mühe als Ganzes erfassbar sein sollte. Das hilft Lesern, also auch euch, den Überblick zu behalten.

Keine Datei darf länger als 500 Zeilen sein. Leerzeilen & Kommentare werden auch mitgezählt. Alles, was sonst länger werden würde, ist sinnvoll zu unterteilen.

Keine Datei außer dem Hauptprogramm darf nur eine Subroutine enthalten.

Die Subroutinen sind zu sinnvollen Modulen zusammenzufassen.

Überflüssiger Code ist nicht zulässig.

Alles, was berechnet wird, muss auch verwendet werden. Nichts, was berechnet wird, darf doppelt berechnet werden.

Das ist ein kleines bischen schwammig, da niemand euch den Kopf abreißt, wenn in einer Schleife zweimal "i+1" steht. Wenn hingegen eine reine Funktion zweimal mit den gleichen Argumenten aufgerufen wird, ist das überflüssiger Code.