Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

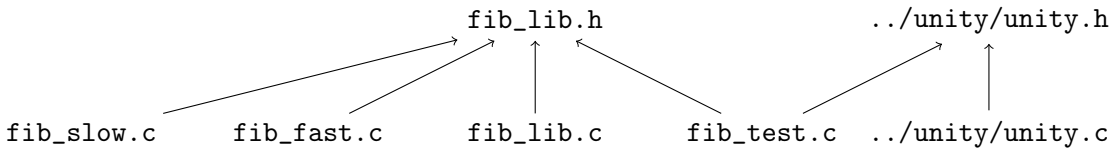
Übungsblatt 2

Abgabe: Montag, 03.05.2021, 9:00 Uhr morgens

In diesem Übungsblatt sollen Sie üben

- einfache Funktionen mit Zahlen und Arrays zu schreiben;
- Funktionen aus der `stdio.h` zu verwenden, um Benutzereingaben zu fordern und Ausgaben zu erzeugen;
- Unittests mit dem *Unity*-Framework zu schreiben;
- Code in `.h`- und `.c`-Dateien aufzuteilen, um den Code in potentiell mehreren `.c`-Dateien wiederzuverwenden; und
- selbständig Regeln in die `Makefile` einzutragen, die Ihren Code kompilieren und linken.

Hierzu sollen Sie zwei Algorithmen zum Berechnen der Fibonacci-Folge[1] implementieren und deren Laufzeit vergleichen. Die Dateistruktur soll dabei, aus Perspektive des `blatt02`-Verzeichnisses, wie folgt aussehen:



Ein Pfeil von a nach b bedeutet in dieser Abbildung, dass die Datei a durch ein `#include` die Datei b verwendet. Die Bedeutung der Dateien ist wie folgt:

- In der Datei `fib_lib.h` und `fib_lib.c` sollen die beiden Algorithmen zum Berechnen der Fibonacci-Zahlen implementiert werden.
- In den Dateien `fib_slow.c` und `fib_fast.c` soll jeweils eine `main`-Funktion stehen, die dazu auffordert eine Zahl n einzugeben, mit einem der beiden Algorithmen $\mathrm{fib}(n)$ berechnet und das Ergebnis ausgibt.
- In der Datei `fib_test.c` sollen Unittests und eine zugehörige `main`-Funktion stehen, die überprüfen ob die beiden Algorithmen korrekt implementiert wurden.

[1] Ich weiß, ich weiß: nicht schon wieder Fibonacci... aber bisher haben wir halt nur Zahlen und Arrays `:3`

– Die `unity.c` und `unity.h` sind bereits Teil Ihres Repositories und im Verzeichnis `xy123/unity/` zu finden – also *nicht* in `xy123/blatt02/unity/`. Wenn sich Ihr Terminal bzw. Ihre `Makefile` in `xy123/blatt02/` befindet, können Sie die Dateien über die relativen Pfade `../unity/unity.c` und `../unity/unity.h` beschreiben.

Hinweis. Die Regeln für `make format` und `make checkstyle` waren im vorherigen Übungsblatt wie folgt:

```
checkstyle:
  clang-tidy --quiet *.c --
format:
  clang-format -i *.c
```

Da wir ab diesem Übungsblatt auch `.h`-Dateien verwenden und diese ebenfalls formatiert und überprüft werden sollen, müssen die Regeln wie folgt angepasst werden:

```
checkstyle:
  clang-tidy --quiet *.c *.h --
format:
  clang-format -i *.c *.h
```

Aufgabe 2.1 (Fibonacci nach Rezept; 3 Punkte)
Die Fibonacci-Folge ist definiert durch

$$\text{fib}(n) = \begin{cases} 0 & \text{wenn } n = 0 \\ 1 & \text{wenn } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{wenn } n \geq 2 \end{cases}$$

Übertragen Sie diese mathematische Definition in die Definition einer rekursiven C-Funktion `int fib_slow(int n)` in der Datei `fib_lib.c` und deklarieren Sie die Funktion entsprechend in der Datei `fib_lib.h`. Sie können dabei annehmen, dass `n` immer positiv ist.

Aufgabe 2.2 (Unittests I; 3 Punkte)
In der Vorlesung wurde gezeigt wie Sie mit dem *Unity*-Framework Unittests schreiben können.

Schreiben Sie drei Unittests für `fib_slow`, die sowohl die beiden Basisfälle abdecken, als auch einen nicht-trivalen Fall für ein höheres `n`, z.B. fib(10) = 55.

Die Unittests inklusive der `main`-Funktion, die die Tests aufruft, sollen in der Datei `fib_test.c` platziert werden.

Achten Sie darauf, dass die Dateien `unity.c` und `unity.h` im Vergleich zur Vorlesung in einem anderen Verzeichnis liegen (wie in der Einleitung beschrieben). Die relativen Pfade `../unity/unity.c`, `../unity/unity.h` und `../unity/unity.o` funktionieren dabei auch in der `Makefile`.

Denken Sie daran, Ihrer `Makefile` eine Regel hinzuzufügen, um die `../unity/unity.o` zu erzeugen, sonst können Sie diese auch nicht beim Linken von `fib_test` verwenden.

Die Regel für `make test` soll dafür sorgen, dass `fib_test` kompiliert und gelinkt wird und anschließend `fib_test` ausführen. Die Ausgabe von `make test` könnte dann z.B. wie folgt aussehen:

```
$ make test
gcc -c fib_test.c -o fib_test.o
gcc -c fib_lib.c -o fib_lib.o
gcc -c ../unity/unity.c -o ../unity/unity.o
gcc fib_test.o fib_lib.o ../unity/unity.o -o fib_test
./fib_test
fib_test.c:42:test_fib_slow_0:PASS
fib_test.c:43:test_fib_slow_1:PASS
fib_test.c:44:test_fib_slow_10:PASS

-----------------------
3 Tests 0 Failures 0 Ignored
```

Aufgabe 2.3 (Benutzereingaben für Fibonacci; 3 Punkte)

Schreiben Sie in der Datei `fib_slow.c` eine `main`-Funktion, die zur Eingabe einer Zahl `n` auffordert, `fib_slow(n)` aufruft, und das Ergebnis ausgibt.

Nach Ausführen des Programms, soll bei einer Eingabe von `10` *exakt* der folgende Text im Terminal zu lesen sein:

```
$ ./fib_slow
Input: 10
fib(10) = 55
```

Hinweis. In C fügt `printf("foo")` nicht automatisch einen Zeilenumbruch am Ende des Strings hinzu. Wenn dieser gewünscht ist muss `printf("foo\n")` geschrieben werden.

Hinweis. Um zur Eingabe einer Zahl aufzufordern, können Sie die folgende Funktion verwenden:

```
int read_int() {
  int n;
  scanf("%d", &n);
  return n;
}
```

Das `&n` müssen Sie zu diesem Zeitpunkt noch nicht verstehen. Es erfüllt den Zweck, dass beim Aufruf von `scanf`, die Zahl `n` nicht kopiert wird, sondern die Speicheradresse von `n` an `scanf` übergeben wird. Dadurch kann `scanf` den Wert von `n` so verändern, dass diese Änderung auch außerhalb der Funktion sichtbar ist.

Hinweis. `printf` und `scanf` sind beide in der `stdio.h` definiert.

Aufgabe 2.4 (Fibonacci on Steroids; 5 Punkte)

Die Implementierung von `fib_slow` ist zwar kurz und schick, da sie genau der mathematischen Definition entspricht, aber macht leider ihrem Namen alle Ehre: ein Aufruf von `fib_slow(100)` kann je nach Rechner viele Stunden dauern.

Dies liegt daran, dass für `n >= 2` jeder Aufruf von `fib_slow(n)` zwei weitere Aufrufe von `fib_slow` verursacht, die wieder zwei Aufrufe verursachen und so weiter. Die Laufzeit von `fib_slow(n)` hängt also exponentiell Argument n ab, da für `fib_slow(n)` insgesamt 2^n rekursive Aufrufe benötigt werden.

Man kann Fibonacci aber auch deutlich schneller implementieren indem man die sogenannte *Dynamic Programming*-Technik verwendet. Die Idee ist hier, dass man um $\text{fib}(n)$ zu berechnen, ein Array `cache` der Größe $n + 1$ anlegt in welchem man die Fibonacci-Zahlen zwischenspeichert, sodass an `cache[i]` die Fibonacci-Zahl $\text{fib}(i)$ steht. Zu Beginn initialisiert man das Array mit den beiden Basisfällen, `cache[0] = 0` und `cache[1] = 1`, und dann berechnet man in einer `for`-Schleife alle Fibonacci-Zahlen für $2 \leq i \leq n$ und trägt diese an der Stelle `cache[i]` ein. Da in jedem Schleifendurchlauf nur die Fibonacci-Zahlen kleiner i benötigt werden und diese bereits in den vorherigen Durchläufen berechnet wurden, kann man die Ergebnisse für $\text{fib}(i-1)$ und $\text{fib}(i-2)$ direkt im Cache nachschlagen. Mit diesem Ansatz hängt die Laufzeit eines Aufrufs nur noch linear vom Argument n ab: um $\text{fib}(n)$ zu berechnen, muss der Schleifenkörper lediglich n mal durchlaufen werden. Selbst für eine Zahl wie $n = 4000$ terminiert der Algorithmus in einem Bruchteil einer Sekunde.

Schreiben Sie eine Funktion `fib_fast` in `fib_lib.c`, die diesen Ansatz umsetzt und fügen Sie eine Deklaration von `fib_fast` in `fib_lib.h` hinzu.

Da wir nach aktuellem Stand der Vorlesung nur Arrays anlegen können, deren Größe bereits vor der Ausführung des Programms bekannt ist, soll das `cache`-Array einfach fest die Länge 5000 bekommen und für `n > 4999` soll das Ergebnis von `fib_slow(n)` zurückgegeben werden.

Erweitern Sie `fib_test.c` um drei weitere Unittests für `fib_fast`.

Schreiben Sie analog zu `fib_slow.c` eine `fib_fast.c`, welche die `fib_fast`-Funktion verwendet, um die Fibonacci-Zahl zu berechnen.

Aufgabe 2.5 (Benchmarking; 0 Punkte)

Sie können die Laufzeit der beiden Algorithmen wie folgt vergleichen:

```
$ time ./fib_slow <<< 45
Input: fib(45) = 1134903170

real 0m4.927s
user 0m4.926s
sys 0m0.001s

$ time ./fib_fast <<< 45
```

```
Input: fib(45) = 1134903170

real 0m0.002s
user 0m0.002s
sys 0m0.000s
```

Das Program `time` ist auf allen Unix-Systemen bereits vorinstalliert. Ein Aufruf von `time cmd arg1 ... argn` verhält sich so als würde man `cmd arg1 ... argn` schreiben, aber misst zusätzlich die Zeit, die der Befehl zur Ausführung benötigt.

Da wir nur die Berechnungszeit messen wollen und nicht zusätzlich wie lange es dauert bis die Benutzereingabe erfolgt, sagen wir dem Programm, dass es die Eingabe nicht aus der Datei `stdin` lesen soll, die mit den Eingaben des Terminals verbunden ist, sondern es die Eingaben aus einer anderen Datei lesen soll, deren Inhalt wir vorab bestimmen. Um dafür nicht extra eine Datei anlegen zu müssen, z.B. mit

```
$ echo 45 > my_input.txt
$ ./fib_fast < my_input.txt
Input: fib(45) = 1134903170
```

verwenden wir die Kurzform `<<<`, die sich genauso verhält und zusätzlich die temporäre Datei nach der Ausführung wieder löscht:

```
$ ./fib_fast <<< 45
Input: fib(45) = 1134903170
```

In dem Aufruf `time ./fib_fast <<< 45` tauschen wir eigentlich das `stdin` des `time`-Programms aus und nicht von `fib_fast`. Da `time` aber die Ein- und Ausgabe-Dateien an den Subprozess `fib_fast` weiterleitet, entsteht dennoch der gewünschte Effekt.

Aufgabe 2.6 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
  Zeitaufwand: 3:30

  <...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.

Vergessen Sie nicht die Datei `erfahrungen.txt` mit einem Commit zu ihrem Repository hinzuzufügen und den Commit mit einem Push auf unseren git-Server zu laden.

```
$ git add erfahrungen.txt
$ git commit -a -m "Added erfahrungen.txt."
$ git push
```