Beispiel `egyptian_multiplication`

Dr. Günter Kolousek

28. Juli 2016

1 Allgemeines

- Drucke dieses Dokument **nicht** aus!
- Halte **unbedingt** die Coding Conventions ein! Zu finden am edvossh!
- Lege für dieses Beispiel wieder ein entsprechendes Unterverzeichnis mit Beispielnummer und Beispielname an!
- Verwende den angegebenen Texteditor!

2 Aufgabenstellung

Schreibe ein `C++` Programm `egyptmul`, das die ägyptische Multiplikation zweier Zahlen implementiert und das Laufzeitverhalten der verschiedenen Varianten (siehe Anleitung) misst.

3 Anleitung

1. Natürlich kann man zum Multiplizieren von Zahlen einfach den Operator `*` verwenden, aber was wäre wenn es nur den Operator `+` geben würde? Schauen wir uns diesen Fall einmal an.

   Beginnen wir wiederum mit einer Funktion `unsigned int mul0(unsigned int n, unsigned int a)`, die $n \cdot a$ berechnet, indem Zahl `a` entsprechend `n` Mal addiert wird und das Ergebnis zurückliefert. `mul0` nennen wir die Funktion nur, weil wir weitere Varianten der Multiplikation in Kürze implementieren werden.

   Überlege dir wie du das realisieren kannst und implementiere diese Funktion in einem Modul `mathutils`, das du in einem gepflegten `main.cpp` an Hand einiger Testfälle testest.

Hier noch einige Tipps:

- Der Einfachheit gehen wir davon aus, dass es sich bei diesen Zahlen um natürliche Zahlen handelt (also ganze Zahlen ≥ 0). Deshalb auch `unsigned` im Prototypen der Funktion `mul0`.
- Nicht vergessen, dass ein Modul aus einer `.cpp` und einer `.h` Datei besteht und in einer Headerdatei auch ein Guard vorhanden sein muss. Beginne mit der Headerdatei, dann kannst du den Prototypen in die `.cpp` Datei kopieren.
- Ach ja, die `.h` Datei gehört natürlich auch in der `.cpp` Datei inkludiert, aber das weißt du ja schon. Und vergiss nicht, dass wir die Headerdateien der Standardbibliothek **vor** unseren eigenen Headerdateien inkludieren.
- Außerdem werden allfällige `using` Direktiven **nach** den Inklude-Direktiven platziert.
- Teste auf jeden Fall *auch* die Faktoren 0 und 1.
- Teste auch, ob deine Funktion die Kommutivität der Multiplikation sicher beherrscht. Natürlich wirst du sehen, dass es so ist, aber es gehört getestet. Das ist immens wichtig, da im Laufe der Zeit Sourcecode erweitert und modifiziert wird und sich leicht Fehler einschleichen können. Wie man richtig testet werden wir noch später lernen.

Ok, fertig? Ja? Dann weiter zum nächsten Punkt. War ja nicht so schwierig.

2. Schaut deine Ausgabe des Testteils in etwa folgendermaßen aus?

```
mul0:
  0 * 0 = 0
  5 * 0 = 0
  0 * 5 = 0
  1 * 1 = 1
  5 * 1 = 5
  1 * 5 = 5
  4 * 5 = 20
  5 * 4 = 20
  4 * 4 = 16
  5 * 5 = 25
```

   Ja, dann ist ja alles gut. Wenn nicht, dann solltest du vielleicht noch den einen oder anderen Testfall hinzufügen. Ob jetzt wirklich alle der angeführten Testausgaben notwendig sind oder nicht ist schwierig zu bestimmen, aber diese schaden auf keinem Fall.

   Schau dir an:

   - alle "Sonderfälle" abgehandelt: Multiplikation mit 0 und mit 1
   - ungerade multipliziert mit gerade
   - gerade multipliziert mit gerade
   - ungerade multipliziert mit ungerade

3. Natürlich kann man die Multiplikation natürlicher Zahlen auf iterative Art und Weise implementieren, aber definiert ist die Multiplikation in der Mathematik ja per Induktion, also rekursiv.

   Die Mathematiker sagen ja, dass 0 eine natürliche Zahl ist und wenn man zu einer natürlichen Zahl die Zahl 1 hinzuzählt man wieder eine natürliche Zahl erhält. Fein, also das sieht aus:

   a) 0 ist eine natürliche Zahl
   b) $0 + 1 = 1$, also ist 1 auch eine natürliche Zahl
   c) $1 + 1 = 2$, also ist auch 2 eine natürliche Zahl
   d) usw.

   Das bedeutet, dass man ausgehend von einer natürlichen Zahl sofort wieder eine natürliche Zahl erhalten kann.

   Ok, aber was hat das mit der Multiplikation zu tun? Hier funktioniert der Mechanismus ähnlich:

   a) $1a = a$, also 1 der einer Zahl a multipliziert gibt a

b) $(n+1)a = na+1$, also führt man durch Auflösen der Klammern der Klammern eine Multiplikation einer größeren Zahl $(n + 1)$ auf die Multiplikation einer kleineren (n) und eine Addition zurück.

Kopiere nun die bestehende Funktion `mul0` auf eine neue Funktion `mul1`, die die gleiche Signatur (also die Reihenfolge und Typen der formalen Parameter) und den gleichen Typ des Rückgabewertes aufweist und implementiere den obigen Algorithmus *rekursiv* (also mittels rekursiver Funktionsaufrufe).

Also: Abbruchbedingung bei dem Wert 1 und ansonsten den Wert berechnen als die Addition des rekursiven Aufrufes (mit $n - 1$) und dem Wert a.

4. Funktioniert es? Ja, so soll es sein! Überlegen wir uns im nächsten Schritt einmal welche Unterschiede es bezüglich dieser beiden Implementierungen gibt. Beide funktionieren offensichtlich, warum also die eine oder die andere vorziehen?

   - `mul1` wird eine kürzere Implementierung aufweisen (2 Zeilen).
   - Beide Funktionen sind gleich verständlich zum Lesen, `mul0` ist etwas geradliniger, aber die rekursive Variante ist ebenfalls leicht zu lesen. Generell werden wir später noch feststellen, dass die rekursiven Varianten *meistens* viel einfacher zum Lesen und zum Formulieren sind als die iterativen Gegenstücke.
   - Wie sieht es aber mit dem zeitlichen Verhalten aus? Dazu müssen wir messen… Und die alte Messtechnikerregel trotzdem nicht vergessen: "Wer misst, misst Mist"

     Füge folgende Headerzeile zu deinem Programm hinzu `#include <chrono>` (gleich nach `#include <iostream>`, du weißt ja mittlerweile wie wir unsere Inklude-Direktiven anordnen).

     Das eigentliche Messen hängen wir gleich nach unseren Testcode hinten an. Dazu benötigen wir einmalig die folgende Definitionen:

     ```
     chrono::time_point<chrono::system_clock> start;
     chrono::time_point<chrono::system_clock> end;
     ```

     Eine Definition legt den Typ eines Speicherobjektes fest und reserviert auch den Speicher dafür. Eine Deklaration legt nur den Typ fest. In einer Headerdateien stehen überwiegend Deklarationen, wie unsere Funktionsdeklarationen es sind.

     Was diese Definitionen wirklich bedeuten und wie die nachfolgende Verwendung dieser beiden Variablen funktioniert werden wir später noch lernen. Jetzt geht es nur um den Zeitbedarf unserer beiden Funktionen.

     Das eigentliche Messen funktioniert folgendermaßen:

     ```
     start = chrono::system_clock::now();
     for (int i{0}; i < 300000; ++i)
         mul0(12345, 67890);
     end = chrono::system_clock::now();
     ```

```
cout << "mul0::elapsed milliseconds: " <<
  chrono::duration_cast<chrono::milliseconds>(end -
    start).count() << endl;
```

Dupliziere dieses Programmausschnitt (engl. snippet) und adaptiere diesen für die andere Funktion. Dann übersetzen und ausführen. Was stellst du fest? Je nach verwendeten Computer und auch je nach Compiler werden die Zeiten unterschiedlich ausfallen. Sind die Zeiten sehr klein, dann hast du einen schnelleren Computer und du erhöhst einfach die Anzahl der Schleifendurchgänge.

5. Ok, jetzt aber zur Interpretation… Aha, die rekursive Variante ist bedeutend langsamer im Vergleich zur iterativen Variante. Woran das liegt haben wir im Theorieunterricht schon besprochen und sollte deshalb zu keiner großen Überraschung gesführt haben.

   Nach dieser absolut überwältigenden Erkenntnis stellt sich natürlich sofort die Frage wie man hier Verbesserungen vornehmen kann. Hier kommt die sogenannte ägyptische Multiplikation (so ca. 3600 Jahre alt) ins Spiel, die ein Schreiber mit dem Namen Ahmes in Ägypten niedergeschrieben hat. Nebenbei gesagt, wird diese ägyptische Multiplikation auch russische Bauernmultiplikation genannt, da laut einer Sage die russischen Bauern auf diese Art und Weise multiplizierten. Schön, nach so viel geballter Information und Unterhaltung jetzt zum eigentlichen Verfahren.

   Die grundlegende Idee liegt darin, dass man die Assozivität der Multiplikation ausnützt, nämlich, dass $4a = ((a + a) + a) + a)$ ist und dies sich wiederum in $(a+a)+(a+a)$ umformen lässt. Hier siehst du, dass man sich auf diese einfache Art und Weise eine Multiplikation erspart, denn man benötigt nur mehr 3 Additionen anstatt 4 Additionen.

   Auch wenn dies auf den ersten Blick nicht sonderlich aufregend ist, können wir uns dies praktisch an Hand eines Beispiels ansehen. Nehmen wir an, wir wollen 41 mit 59 multiplizieren. Dazu erstellen wir zuerst folgende Tabelle:

1	✓	59
2		118
4		236
8	✓	472
16		944
32	✓	1888

   In der ersten Spalte erkennst du mit geschultem Auge sofort unsere alt bekannten 2er-Potenzen, die dritte Spalte beinhaltet in der ersten Zeile die Zahl, die zu multiplizieren ist und in jeder weiteren Zeile jeweils den doppelten Wert der vorhergehenden Zeile. Jetzt zur zweiten Spalte: Hier findest du für jedes 1-Bit der Darstellung der Zahl 41 im 2er-System ein Hakerl.

Das Prinzip ist jetzt folgendes: Alle Zahlen der dritten Spalte addieren, bei denen in der jeweiligen 2. Spalte ein Hakerl ist. Fertig.

Ok, das bedeutet:

$$41 \cdot 59 = 1 \cdot 59 + 8 \cdot 59 + 32 \cdot 59 = 2419$$

Die einzige Multiplikation, die benötigt wird ist die Verdopplung, d.h. ausgehend von der Zahl 59 werden fortlaufend Verdopplungen vorgenommen. Diese Verdopplungen selber können aber durch eine einfache Addition des jeweiligen Wertes mit sich selbst realisiert werden. Hat man die Einzelterme ermittelt, dann müssen die nur noch mehr addiert werden.

Gut, aber wie sieht das praktisch aus? Einfach bei jeder Zweierpotenz, die im Faktor enthalten ist den entsprechenden verdoppelten Wert von der zu multiplizierenden Zahl addieren und bei ungeraden Zahlen einmal die zu multiplizierende Zahl zusätzlich addieren (für 2^0).

Also hier in einem Pseudocode:

- $1a = a$ wie gehabt $\rightarrow$ Abbruch, Rückgabe von a
- anderenfalls
  - $r \leftarrow mul2(n/2, a + a)$
  - Wenn n ungerade, dann $r \leftarrow r + a$
  - Rückgabe von r.

Der Pfeil nach links ($\leftarrow$) wird in Pseudocodedarstellungen gerne verwendet, um eine Zuweisung auszudrücken.

Los geht's! Implementieren und testen mit Testcode wie gehabt. Beachte nur, dass wir bei unseren Testfällen auch die Multiplikation mit 0 explizit mit aufgenommen haben. Daher diesen Fall explizit behandeln.

6. So, jetzt auch noch schnell die Zeitmessung für `mul2` hinzufügen!

   Fertig? Beeindruckt?!

7. Geht es noch schneller? Bevor wir uns an die Verbesserung des Algorithmus wagen, wollen wir uns zuerst die Implementierung ansehen. Wahrscheinlich hast du die folgende Berechnungen vorgenommen:

   - Die Hälfte von n hast du mit `n / 2` berechnet
   - Die Summe von $a + a$ hast du mittels `a + a` berechnet (eh klar)
   - Ob n ungerade ist hast du mittels `if (n % 2)` festgestellt

   Alles richtig, keine Frage, aber explizit kann man einige Optimierungen vornehmen, die ein Compiler *unter Umständen* selber vornimmt. Schreibe eine Funktion `mul3`, die folgende Optimierungen vornimmt:

- Die Hälfte von n wird durch `n >> 1` realisiert. Du weißt schon warum die so ist, nicht wahr?
- Die Summe von $a+a$ wird auf $2a$ zurückgeführt und dies wird ebenfalls mittels Verschieben vorgenommen. Auch das solltest du mittlerweile schon wissen.
- Ob n ungerade ist, kann man am Bit 0 erkennen (auch schon gelernt) und dies geht wiederum ganz einfach: `if (n & 0x1)`.

Wenn es Unterschiede bezüglich der Laufzeit gibt, dann werden diese gering ausfallen, aber es geht ja um das Prinzip… und um das Wissen!

8. Man kann schon noch weitere Verbesserungen vornehmen. Schau dir einmal die Abfrage an, ob der Faktor gleich 0 ist. Dieser ist eigentlich genau ein Mal von Relevanz, sonst aber wird die Abfrage bei jedem Aufruf der Funktion vorgenommen, kommt aber sonst überhaupt nicht zum Tragen, da die Rekursion ja mit der Gleichheit von `n` mit 1 abbricht.

   In diesem Fall kann man eine substantielle Verbesserung vornehmen, indem man diese Abfrage aus der Funktion herausnimmt:

   ```
   unsigned int mul4_rec(unsigned int n, unsigned int a);
   unsigned int mul4(unsigned n, unsigned a) {
      if (n == 0) return 0;
      return mul4_rec(n, a);
   }
   ```

   Beachte, dass `mul4_rec` **nicht** in die Headerdatei gehört, diese Funktion ist nicht öffentlich, d.h. diese soll nicht von einem normalen "Benutzer" aufgerufen werden, sondern nur von der Funktion `mul4`.

   Natürlich wird diese Verbesserung in unserem Fall keine nennenswerte Senkung der Laufzeit nach sich ziehen. Deshalb machen wir gleich im nächsten Punkt weiter.

9. Das eigentliche Problem an Funktionen ist, dass ein Funktionsaufruf relativ lange dauert (siehe Theorieunterricht) und im Falle eines rekursiven Algorithmus kann das durchaus zu drastischen Performanceproblemen führen.

   Um diese rekursiven Funktionsaufrufe zu eliminieren gehen wir so vor, dass wir eigentlich (im ersten Schritt) mehr berechnen! Wir werden nämlich jetzt unsere Funktion so umstellen, dass diese nicht nur multipliziert sondern zusätzlich auch noch *addiert*. D.h. wir schreiben eine Funktion, die $r + na$ berechnet, wobei es sich bei r um das Zwischenergebnis handelt, das alle Teilprodukte na aufsummiert.

   D.h. wir benennen `mul4_rec` in `mul4_acc0` ("acc" für accumulate und "0" für die Version 0, es werden weitere folgen) um und erweitern die Parameterliste sodass folgender Prototyp entsteht:

   ```
   unsigned int mul4_acc0(unsigned int r, unsigned int n, unsigned int a);
   ```

Bevor wir uns mit der Implementierung dieser Funktion befassen, werden wir allerdings den Aufruf innerhalb von `mul4` entsprechend anpassen müssen. Das ist allerdings einfach, da lediglich 0 als erster Parameter übergeben werden muss, der Rest bleibt gleich. 0 weil es sich ja bei `r` um das Zwischenergebnis handelt (quasi ein Akkumulator, deshalb auch der Name der Funktion).

Jetzt noch zur Implementierung der Funktion `mul4_acc0` selber. Hier einmal wie das aussehen sollte:

```
unsigned int mul4_acc0(unsigned int r, unsigned int n, unsigned int a) {
    if (n == 1) return r + a;
    if (n & 0x1)
        return mul4_acc0(r + a, n >> 1, a << 1);
    else
        return mul4_acc0(r, n >> 1, a << 1);
}
```

Bei genauerer Betrachtung erkennen wir, dass sich auch hier nichts bei der eigentlichen Funktionalität geändert hat. Lediglich die explizite Verwendung einer lokalen Variable hat sich zur Verwendung eines Parameters hin geändert.

Ich möchte klar darauf hinweisen, dass ich eigentlich **nicht** dazu rate die `if` Anweisung *ohne* geschwungenen Klammern in den beiden Zweigen zu schreiben, da weitere hinzugefügte Anweisungen (hier im `else` Zweig) nicht mehr richtig vom Compiler so erkannt werden wie man sich das vorstellt (`C++` ist nicht Python). Ok, hier ist es so, weil ich weiß, dass hier in dieser Form keine Erweiterungen mehr vorgenommen werden.

So, jetzt ist es wieder Zeit zu testen. Es sollte wie gehabt funktionieren. Lediglich bei der Performance wird sich nichts getan haben, da wir ja keine substantiellen Verbesserungen vorgenommen haben.

10. Na dann begeben wir uns langsam in Richtung einer Verbesserung der Performance. Aber vielleicht doch nur sehr langsam und nehmen die nächste strukturelle Änderung vor. Irgendwie ist es nicht schön, dass wir jetzt in `mul4_acc0` zwei rekursive Aufrufe haben. Schöner ist es, wenn nur ein rekursiver Aufruf enthalten ist.

    Kopiere daher `mul4_acc0` auf `mul4_acc1` und ändere die Funktion wie folgt ab:

    ```
    if (n == 1) return r + a;
    if (n & 0x1) r += a;
    return mul4_acc0(r, n >> 1, a << 1);
    ```

    Nur mehr ein rekursiver Aufruf, Ziel erreicht. Aber ein ganz spezieller rekursiver Aufruf, der am Ende der rekursiven Funktion ganz alleine steht. So einen rekursiven Aufruf nennt man *tail-recursive*!

11. Aber auch da ist nicht wirklich viel passiert. Erinnere dich, dass ein Verändern eines Programmteiles mit dem Ziel eine Verbesserung der Struktur ohne Änderung des funktionalen Aspektes "Refactoring" genannt wird.

    Schauen wir uns jetzt unsere `mul4_acc1` Funktion an, dann erkennen wir zwei Punkte:

    - `n` ist sehr selten 1, aber die Überprüfung wird jedes Mal vorgenommen.
    - Wenn `n` gerade ist, dann müsste `n` auch nicht auf 1 überprüft werden.

    D.h. zuerst auf ungerade abfragen und dann auf `r` aufakkumulieren und erst dann wenn `n` gleich 1 ist, `r` zurückzuliefern. Anderenfalls den Rückgabewert des rekursiven Aufrufes zurückliefern. Schaffst du das? Baue es in eine Funktion `mul4_acc2` ein.

    Probiere es!

    Nein, dann hast du hier eine Lösung mit ein paar Lückentexten (wie bei einem Rätsel):

    ```
    if (n & 0x1) {
        r __ a;
        if (n == 1) return ____;
    }
    return mul4_acc0(r, n >> 1, a << 1);
    ```

    Beachte, dass es noch immer nicht zu einer wirklichen Performanceverbesserung gekommen ist. Alles was mit dem Messen der Performance zu tun hat ist äußerst kompliziert. Es spielen einfach zu viele Faktoren eine Rolle: Compiler, Compiler-Optionen, Betriebssystem, Speicher, Cache (!), Auslastung des Systems,...

12. Jetzt noch zu einem weiteren Refactoring-Schritt. Ich gebe den Code wieder einmal zur Gänze vor (nenne die Funktion `mul_acc3`):

    ```
    if (n & 0x1) {
        r += a;
        if (n == 1) return r;
    }
    n >>= 1;
    a <<= 1;
    return mul4_acc0(r, n, a);
    ```

    Klar, man sieht auf den ersten Blick die Änderungen und natürlich sieht man auch, dass es sich um ein Refactoring handelt, da keine Änderung der Funktion erreicht wurde. So eine tail-recursive Funktion nennt man *strictly tail-recursive*, da im rekursiven Aufruf dieser Funktion genau/nur die formalen Parameter als Argumente übergeben werden.

13. Jetzt der finale Schritt: Eine strictly tail-recursive Funktion kann man in eine iterative Funktion umwandeln, indem man den Funktionsaufruf in eine `while (true)` Schleife umwandelt (Funktion `mul4_acc4`)

```
while (true) {
    if (n & 0x1) {
        r += a;
        if (n == 1) return r;
    }
    n >>= 1;
    a <<= 1;
}
```

Implementiere und betrachte jetzt auch die Performance! Nicht schlecht, oder?

14. Ok, das ist ja alles schön und gut und wir haben auch eine Menge dabei gelernt, aber schaue dir einmal die vorhergehende Tabelle an und versuche diese direkt zu implementieren! Das geht direkt ohne Rekursion und man kommt zu gleichen Ergebnissen wie bei der letzten optimierten Variante!

Nenne die Funktion `mul5` weil die ja doch ein anderer Zugang zu dem Problem darstellt.

15. So, jetzt lernen wir noch eine neue Technologie kennen (zwecks der Abwechslung…). Das händische Übersetzen haben wir jetzt kennengelernt, ist aber auf die Dauer doch mühsam. Daher werden wir uns jetzt eine Software anschauen, die diese Tätigkeit automatisiert.

Erstelle eine Textdatei mit dem Namen `CMakeLists.txt` in dem Verzeichnis in dem sich auch deine Source- und Includedateien befinden. Befülle diese Datei dem folgenden Inhalt:

```
cmake_minimum_required(VERSION 3.5)
project(egyptian_multiplication)
add_executable(egyptmul main.cpp mathutils.cpp)
```

Dann gib in diesem Verzeichnis in der Shell den folgenden Befehl ein:

```
cmake .
```

Daraufhin wird das Programm `cmake` sich deine Datei `CMakeLists.txt` ansehen (im aktuellen Verzeichnis, siehe Argument `.` des Befehls!) und nach einem geeigneten Compiler auf deinem System suchen und dies auch in entsprechenden Meldungen auf der Ausgabe dokumentieren.

Wenn alles funktioniert hat, dann erscheint als letzte Meldung:

```
-- Build files have been written to:...
```

Ein beherztes `ls` in deinem Arbeitsverzeichnis wird auch zeigen, dass `cmake` so einiges an Dateien erstellt hat. Die wichtigste Datei ist die Datei `Makefile`. Darauf werden wir später noch zu sprechen kommen. Jetzt aber folgt nur mehr die Eingabe eines kurzen Kommandos `make` in der Shell und das Programm wird übersetzt.

So, jetzt brauchst du nach jeder Änderung in deinem Programm nur mehr `make` eingeben und dein Programm wird korrekt übersetzt und kann danach mit dem Executable `egyptmul` zur Ausführung gebracht werden.

16. Weitere Übungsbeispiele zum Üben der Rekursion (irgendwo, irgendwann, irgendwie), damit du für die praktische Arbeit gut vorbereitet bist:

    - Schreibe eine rekursive Funktion `unsigned int fact(unsigned int)`, die die Faktorielle gerechnet (und zurückliefert).
    - Schreibe eine rekursive Funktion `string reverse(string)`, die den übergebenen String "umgedreht" zurückliefert.
    - Schreibe eine Funktion `double pow(int a, unsigned int b)`, die die Potenz der ganzen Zahlen `a` mit der natürlichen Zahl `b` rekursiv berechnet und zurückliefert.
    - Schreibe eine Funktion `unsigned int sum_digits(unsigned int a)`, die die Ziffernsumme der natürlichen Zahl `a` rekursiv berechnet und zurückliefert.
    - Die Fibonacci-Folge ist eine unendliche Folge von Zahlen. Jede Zahl dieser Folge berechnet sich aus der Summe der beiden Vorgänger. Die erste Zahl ist 0 und die zweite Zahl ist 1. Die Folge lautet also: $0, 1, 1, 2, 3, 5, 8, 13, 21,...$

      Schreibe eine rekursive Funktion `unsigned int fibonacci(unsigned int n)`, die die n.te Zahl der Fibonacci-Folge berechnet und zurückliefert. Also `fibonacci(=6)` liefert 8.
    - Schreibe eine rekursive Funktion `boolean strcmp(string a, string b)`, die überprüft, ob die gegebenen Strings `a` und `b` gleich (liefert `true`) sind oder nicht (liefert `false`).
    - Schreibe eine rekursive Funktion `double add(vector<double> numlist)`, die alle Werte des Arguments `numlist` aufaddiert, indem jeweils nur eine Addition verwendet wird.
    - Schreibe eine rekursive Funktion `boolean is_palindrom(string)`, die feststellt, ob der übergebene String ein Palindrom ist. Ein Palindrom ist von vorne nach hinten gelesen gleich ist wie von hinten nach vorne gelesen. Beispiele "otto", "lagerregal",...
    - Schreibe eine rekursive Funktion `unsigned int horner(string num, unsigned int b)`, die das Horner-Schema implementiert.

4 Übungszweck dieses Beispiels

- Testen
- rekursive Funktionen
- Deklaration vs. Definition
- Algorithmus analysieren und schrittweise verbessern
- Zeitverhalten von rekursiven Funktionen
- Parameter
- Refactoring kennenlernen
- Umwandlung von Rekursion in Iteration
- CMake (ohne Unterverzeichnisse)