Bonusaufgaben zum C/C++-Praktikum

Fortgeschrittene Themen in C++

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 4

Hinweise zur Abgabe:
verwendet als Grundlage für die Bearbeitung die im Moodle bereitgestellte Vorlage. Beachtet auch die Hinweise auf dem ersten Bonuszettel.

Aufgabe 4.1: `[F]` Strings (6 Punkte)

In dieser Aufgabe sollen Symbole und Buchstaben in `std::string` gezählt werden.

Bei allen Teilaufgaben ist es sinnvoll Lambda Ausdrücke zu verwenden und die Ergebnisse aller Teilaufgaben sollen im Anschluss auf der Konsole ausgegeben werden. Hilfreiche Funktionen der STL sind im Anhang aufgelistet, ihr dürft aber auch andere verwenden. Eine Erklärung zu Lambdas ist ebenfalls angehängt.

4.1a) Alle Buchstaben zählen (1 Punkt)

In dieser Funktion sollen alle Buchstaben (aA bis zZ) im string gezählt werden, dabei ist die Groß oder Kleinschreibung irrelevant. Umlaute und ’ß’ werden hier nicht betrachtet.

```
size_t countAbc(const std::string& input);
```

4.1b) Einzelne Buchstaben zählen (1 Punkt)

Dieses Mal sollen die Buchstaben getrennt gezählt und in einer Map aufgeschlüsselt werden. Speichere für jeden Buchstaben, der mindestens einmal vorkommt, in der Map, wie häufig dieser auftaucht. Dabei ist wieder die Groß- und Kleinschreibung zu ignorieren, heißt A und a zählen als der selbe Buchstabe.

```
std::map<char, size_t> countIndividual(const std::string& input);
```

4.1c) Symbole zählen (2 Punkte)

Erstelle nun eine Klasse `SymbolCounter` die im Konstruktor eine Liste von Symbolen übergeben bekommt. Die Klasse soll den `(...)`-Operator überladen um damit die Summe der übergebenen Symbole im String zu zählen. Hier muss die Groß-/Kleinschreibung beachtet werden!

```
// Verwendung der Klasse SymbolCounter
SymbolCounter sc({'a', 'A', '$'});
size_t anzahl = std::count_if(str.begin(), str.end(), sc);
```

4.1d) Vielfalt der Symbole (2 Punkte)

Nun soll bestimmt werden welche verschiedene Symbole im String vorkommen. Dabei sind Groß- und Kleinschreibung wieder zu ignorieren, „a“ und „A“ zählen als das selbe Symbol. Am Ende soll eine `std::list<char>` mit den Symbolen zurückgegeben werden.

Ein möglicher Ansatz wäre es den String zu sortieren und `std::unique` zu benutzen. Ihr könnt aber auch anders vorgehen.

```
std::list<char> usedSymbols(const std::string& input);
```

Aufgabe 4.2: [F] Numerik (4 Punkte)

4.2a) Newton Verfahren (2 Punkte)

Das Newton Verfahren ist eine Methode um numerisch die Gleichung $f(x) = 0$ zu lösen. In Gleichung 1 ist das Verfahren selbst dargestellt. Die Anzahl der Iterationen die durchgeführt werden sollen, sowie der Startpunkt x_0 werden dabei vorgegeben.

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \tag{1}$$

Gegeben ist die Funktion $f(x) = x^3 + 4x^2 - 7x + 12$ und ihre Ableitung $f'(x) = 3x^2 + 8x - 7$. Als Startpunkt soll hier $x_0 = 0$ verwendet werden und $n = 1000$ Iterationen durchgeführt werden. Vervollständige die Signatur der angedeuteten Funktion `newton(...)` und implementiere diese.

```
double newton(... fx, ... fderiv, double x0, size_t n);

// mögliche Verwendung der Funktion newton(...)
int main(...) {
   auto fx = [](double x){ return std::pow(x, 3) + 4 * std::pow(x, 2) - 7 * x +
    12; };
   auto fderiv = [](float x){ return 3 * std::pow(x, 2) + 8 * x - 7; };

   double solution = newton(fx, fderiv, 0, 1000);
}
```

4.2b) Templates (2 Punkte)

Erstelle von den Funktionen `newton`, `fx` und `fderiv` nun Template Versionen, die den Datentypen als Template Parameter erhalten. Nennt diese `newtonTemp`, `fxTemp` und `fderivTemp`.

Führe die Berechnung aus der vorherigen Ausgabe nun mit den Datentypen `float`, `double`, `unsigned int` und `int` durch. Gebe die Ergebnisse der unterschiedlichen Varianten auf der Konsole aus.

Lambda Funktionen

Lambda Ausdrücke oder auch Lambda Funktionen sind Funktionen, die nicht an einen Bezeichner gebunden sind. Das heißt sie können bspw. an STL Funktionen übergeben werden, die als Parameter eine Funktion erwarten, ohne sie global zu deklarieren.

In C++ haben Lambdas die Form:

```
[]( /* Parameter */ ){ /* Funktionskörper */ }
```

Die runden Klammern enthalten die Übergabeparameter, wie bei einer normalen Funktion auch und in den geschweiften Klammern ist der Funktionskörper enthalten. Ein Beispiel anhand der Funktion `std::count_if`, die ihr auch weiter hinten erklärt findet.

```
std::vector<int> zahlen{1, 2, 3, 4, 5};
auto nbr = std::count_if(zahlen.begin(), zahlen.end(), [](int zahl){
                         return zahl > 2; });
```

Hierbei werden alle Zahlen größer zwei gezählt ohne eine eigenständige Funktion deklarieren zu müssen. Für die Parameter des Lambda kann auch das `auto`-Keyword verwendet werden.

Falls das Lambda mehrmals benötigt wird kann es auch einer Variable zugewiesen und anschließen wie eine Funktion verwendet werden.

```
auto greaterTwo = [](int zahl){ return zahl > 2; };

// Bsp Verwendung
if(greaterTwo(7)) { // ...

// oder
std::(count_if(zahlen.begin(), zahlen.end(), greaterTwo);
```

Zusätzlich kann über die eckigen Klammern noch bestimmt werden ob das Lambda auf die umliegenden Objekte zugreifen darf. Mit `&` werden die umliegenden Objekte als Referenz übergeben und mit = by Value.

```
int x = 2;

// Compiliert nicht, da auf x nicht zugegriffen werden darf
auto greaterX = [](int zahl){ return zahl > x; };

// Compiliert, da x als Referenz übergeben wird
auto greaterX = [&](int zahl){ return zahl > x; };
```

Referenz

Hier sind einige nützliche Funktionen aus der STL, ihr seid aber nicht auf diese beschränkt.

Groß- & Kleinschreibung

Mit den Funktionen `std::tolower(...)` und `std::toupper(...)` können `char` zu großen bzw. kleinen Buchstaben umgewandelt werden.

Zählen

Mit der Funktion `std::count` könnt ihr einzelne Werte zählen.

```
auto nbr = std::count(iterator.begin(), iterator.end(), var);
```

Um zu zählen wie viele Elemente eine Bedingung erfüllen könnt ihr die Funktion `std::count_if` verwenden. Hier wird als dritter Parameter nicht der zu zählende Wert angegeben, sondern ein Prädikat.

```
auto nbr = std::count(iterator.begin(), iterator.end(), [](auto item){
           return true; });
```

Sortieren

Für das Sortieren eines Containers gibt es die Funktion `std::sort(...)`. Der Datentyp der Elemente des Containers muss dafür den <-Operator implementiert haben. Ansonsten kann als dritter Parameter auch eine eigene Vergleichsfunktion übergeben werden.

```
std::sort(iterator.begin(), iterator.end(), [](auto a, auto b){
           return a < b; });
```

Unique

Die Funktion `std::unique(...)` entfernt alle bis auf ein Element von Gruppen des gleichen Elements, innerhalb eines Containers. Aus 'aaabba' würde also 'aba' werden. Der Rückgabewert der Funktion ist ein Iterator zum neuen letzten Element. Da die Funktion selbst nichts löscht sondern nur verschiebt muss das auch noch geschehen.

```
auto last = std::unique(container.begin(), container.end());
container.erase(last, container.end());
```

Transform

Solltet ihr einen Container transformieren sollen, also auf jedes Element eine Funktion anwenden und das Ergebnis in einem neuen Container speichern, könnt ihr `std::transform(...)` verwenden. Die Funktion erwartet zunächst ganz normal die Iteratoren für den Eingabecontainer. Als Dritter Parameter wird ein Iterator zum Einfügen in den neuen Container benötigt und schlußendlich die Funktion die angewandt werden soll.

```
std::transform(in.begin(), in.end(), std::back_inserter(out), [](auto var){
   return var+1;
});
```