10_calc: Entwicklung eines einfachen Taschenrechners

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

2 Aufgabenstellung

In diesem Beispiel geht es darum, den Prototypen für einen einfachen RPN (Reverse Polish Notation, auch UPN) in einen "echten" Rechner `calc` umzuschreiben. Also auch nichts besonderes.

Die Benutzerschnittstelle soll auf einer REPL (Read Evaluate Print Loop) Interpreter basieren:

```
>>>
>>> 1
>>> 2
>>> 3
>>> print
3
2
1
>>> swap
>>> print
2
3
1
>>> rot
>>> print
1
2
3
>>> +
3
>>> 5
>>> mul
15
>>> print
15
3
>>> 90
```

```
>>> sin
1.5708
>>> asin
90
>>> rad
1.5708
>>> deg
90
```

- An Operatoren sollen zur Verfügung stehen: `+`, `-`, `*`, `/`, `**` (Potenzierungsoperator)
- Die Operanden werden als Gleitkommazahlen angenommen.
- Als Befehle/Funktionen benötigen wir:

  `add`, `sub`, `mul`, `div`, `pow` wie `+`, `-`, `*`, `/`, `**`

  `chs` unärer Operator zum Wechseln des Vorzeichens

  `print` zeigt Stack

  `clst` löscht Stack

  `sqrt` Quadratwurzel vom Top

  `swap` Vertaucht die beiden obersten Elemente

  `rot` rotiert den Stack um ein Element nach unten, d.h. unterstes Element wird oberstes Element

  `log` dekadischer Logarithmus

  `ln` natürlicher Logorithmus

  `ld` dualer Logarithmus

  `exp` Exponentialfunktion

  `sin`, `cos`, `tan`, `asin`, `acos`, `atan` Winkelfunktionen: Wir rechnen in Grad!!!

  `deg`, `rad` Umrechnen in Grad bzw. Radiant

  `help` Anzeigen eines Hilfetextes

  `quit` Beenden des REPL (auch mit `CTRL-D`)

3 Anleitung

1. Für dieses Beispiel macht es Sinn sich die relevanten Teile, also das Modul `repl` und das Meson-Projekt in das neue Beispiel zu kopieren. Das Modul `stack` benötigen wir nicht mehr!
2. Eigentlich ist es ziemlich sinnlos für solch einen Anwendungsfall einen eigenen Stack zu implementieren. Verwende statt dessen den Container-Adaptor `stack` aus der Standardbibliothek und kümmere dich darum, dass dein Prototyp (ist ja nicht mehr im Moment) genauso funktioniert wie zuvor.
3. Wir wollen für die eigentliche Implementierung von `eval` wieder einmal eine `switch`-Anweisung verwenden. Erstens zur Übung und zweitens um dies auch mit einer `enum class` zu verwenden.

   Unser Rechner soll viele Kommandos beherrschen und damit wird unsere `switch`-Anweisung natürlich auch recht lang. Falls du es noch nicht auf Basis einer `switch`-Anweisung realisiert hast, stelle vorerst deinen Rechner auf solch eine um.

   Es soll jetzt wieder alles wie gewohnt funktionieren.

4. Wenn du dir das ansiehst, dann bemerkst du, dass in jeder `case`-Klause das selbe Abrufen der Argumente vom Stack nötig ist und das ist eigentlich ziemlich unnötig. Wir müssten ja nur wissen, ob ein Kommando eine einwertige (unary) oder zweiwertige (binary) Operation darstellt, dann könnte man vorweg die richtige Anzahl an `pop`-Aufrufen durchführen.

   Schreibe daher eine Hilfsfunktion `pair<double, double> get_arguments(Command)`, die in Abhängigkeit der Wertigkeit der Operation, die entsprechende Anzahl an Zahlen vom Stack holt und diese zurückliefert. Bei einwertigen Operatoren soll der zweite Wert einfach 0 sein und bei nullwertigen Operatoren kann man einfach alle beiden Werte des Paares auf 0 setzen.

   Beim Aufruf kannst du ein neues Feature von `C++17`, nämlich structured bindings, verwenden:

   ```
   auto [arg1, arg2]{get_arguments(cmd)};
   ```

5. Ok, spätestens jetzt können wir grundsätzlich `add`, `sub`, `mul` und `div` in `eval` fertig implementieren.

6. Was passiert, wenn du jetzt zuwenig Elemente am Stack hast? Probiere es aus?

   Ok, das ist nicht gut. Ich denke in `get_arguments()` wäre eine entsprechende Abfrage am besten aufgehoben, die in weiterer Folge eine entsprechende Exception wirft. Jetzt fehlt vermutlich noch das Abfangen der Exception in `main()`...

   Beachte, dass wir die Exceptions nicht in der Schnittstelle unseres Moduls haben, aber diese sind trotzdem alle von `std::exception` abgeleitet. Ob, das gut ist oder nicht...

7. Versuche jetzt `print` zu implementieren.

   Ok, das geht nicht. Was also ist zu tun? Verwenden wir einfach `std::deque` anstatt `std::stack`. Recactoring!

   Über eine `deque` kann man iterieren. Das ist praktisch. Hast du eh die richtige Seite gewählt? `;-)`

8. Wahrscheinlich hast du jetzt innerhalb von `eval()` die Ausgabe durchgeführt. Aber ist es sinnvoll eine Funktion `eval()` zu schreiben, die als Nebeneffekt eine Ausgabe durchführt. Sicher nicht!!!

   Also was ist zu tun? Es wäre doch gut, wenn wir beliebig viele Ergebniswerte von `eval()` zurückliefern könnten. Dann brauchen wir auch kein `optional` mehr. Was wir brauchen ist, einen beliebig großen Bereich an Werten von Typ `double` zurückzuliefern.

   Natürlich könnte man z.B. einen `vector` zurückliefern, aber

   - es muss einer angelegt werden.
   - die Werte aus dem Stack müssen hineinkopiert werden.
   - der Vektor muss zurückgeliefert werden. Das kann wieder zu einer Kopieraktion führen.

   Eigentlich wollen wir lediglich auf die schon im Stack vorhandenen Werte verweisen. In `C++` Dafür gibt es in `C++` das Iteratorkonzept, das sowohl in der Standardbibliothek verwendet wird, als auch direkt mit Arrays verwendet werden kann.

   - Wir ändern den Prototypen der Funktion `eval()` so um, dass der Rückgabetyp jetzt `pair<deque<double>::const_iterator, deque<double>::const_iterator>` ist. D.h. wir geben wieder ein `pair` zurück, das zwei Iteratoren zurückliefert, deren referenzierende Werte (auf die verwiesen wird) nicht geändert werden können. Den konkreten Iteratorentyp bekommen wir direkt von der `deque` eben mittels `const_iterator`.

     Die Angabe ist etwas sperrig, aber wenn wir einen Typalias für solch einen Iterator verwenden...

     Der Sinn von diesem `pair` soll sein, dass es einen Bereich von Werten zurückliefert, der durch die beiden Iteratoren gekennzeichnet ist.

- Als nächstes muss noch der Rumpf von `eval()` entsprechend geändert werden:
  - Soll nur ein Wert zurückgeliefert werden, dann wird dieser durch das rechts offene "Intervall" `pair{stack.cbegin(), stack.cbegin() + 1}` angegeben.
  - Will man alle Werte spezifizieren, dann nimmt man als obere Grenze `stack.cend()`.
  - Kein Wert wird angegeben, wenn die untere Grenze gleich der oberen Grenze ist.
- Als letztes muss noch der Inhalt von `main()` noch anpassen. Man muss die zurückgelieferten Werte iterieren.

  Als kleine Hilfestellung, gebe ich hier ein Snippet an, das genau das erfüllt:

  ```
  for (auto it{res.first}; it != res.second; ++it)
  cout << *it << endl;
  ```

  Du siehst, das funktioniert wie bei Zeigern und Zeigerarithmetik!

9. Jetzt bist du in der Lage `print` und alle anderen Kommandos zu implementieren. Happy hacking!

4 Übungszweck dieses Beispiels

- `std::stack` verwenden
- `std::pair` verwenden
- structured bindings verwenden
- `deque` verwenden
- Iteratoren verwenden und verstehen
- Mathematische Funktionen der Standardbibliothek kennenlernen
- Grad und Radiant, Umrechnung