09_protocalc: 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, einen Prototypen für einen einfachen RPN (Reverse Polish Notation, auch UPN) Taschenrechner `protocalc` zu entwickeln. Also nichts besonderes.

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

```
>>>
>>> 1
>>> 2
>>> +
3.0
>>> 3
>>> mul
9.0
```

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

  `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

deg, rad Umrechnen in Grad bzw. Radiant

help Anzeigen eines Hilfetextes

quit Beenden des REPL (auch mit CTRL-D)

3 Anleitung

1. Auch dieses Beispiel soll einen Stack beinhalten, der weitgehend die gleiche Schnittstelle hat wie im vorhergehenden Beispiel, aber das jetzt mittels Smart-Pointer funktioniert! Außerdem soll als Typ jetzt nicht `int` verwendet werden, sondern wir stellen unseren Stack auf `double` um.

   Dazu kannst du eigentlich den Ordner `stack` des vorhergehenden Beipiels in das neue Beispiel kopieren und entsprechend dieser Anleitung anpassen.

   Um die Sache übersichtlich zu gestalten, lassen wir das Kopieren und Zuweisen eines Stacks weg: Wir benötigen es eigentlich nicht und es verkompliziert die Angelegenheit nur unnötig.

   Was ist zu tun?

   - Die Struktur `Node` besitzt jetzt keinen rohen Zeiger sondern einen `unique_ptr` als `next`.
   - Jetzt gibt es prinzpiell drei Möglichkeiten:
     - Es werden die `unique_ptr`-Instanzen in der folgenden Art angelegt: `unique_ptr<Node> x{new Node};` und dann in weiterer Folge mittels dem Operator `->` die Instanzvariablen `value` und `next` zu setzen. Dies ist bzgl. Speicher nicht die optimale Lösung, denn `make_unique()` legt den Speicher für das Objekt und den Smart-Pointer gemeinsam (d.h. im Speicher hintereinander) an.
     - Will man `make_unique()` verwenden, dann gibt es eine "einfache" Möglichkeit, nämlich mittels `make_unique<Node>()` einen entsprechenden neuen Smart-Pointer anzulegen und danach die Instanzvariablen `value` und `next` wieder im Nachhinein zu setzen. Achtung: `unique_ptr` können nicht kopiert werden, also `move()` verwenden.
     - Die "beste" Möglichkeit ist, `make_unique()` in der folgenden Art `make_unique<Node>(value, move(top_))` aufzurufen und schon ein initialisiertes Objekt zu erzeugen. Da ein Smart-Pointer nicht kopiert werden kann, ist allerdings `move()` zu verwenden.

       Leider ist es damit nicht getan, denn dafür benötigen wir jetzt einen Konstruktor für die Klasse `Node`:

       ```
       Node(double value, std::unique_ptr<Node>&& next) : value{value}, next{move(next)} 
       ```

       Hier die Erkärung:

       * Der Konstruktor ist notwendig, da `unique_ptr` nicht kopiert werden können. Das wissen wir schon.
       * Du siehst hier, dass die Initialisierliste verwendet wird. Ohne diese geht es in diesem Fall so nicht. Weiters siehst du hier, dass der Name der Instanzvariable und der Name des Parameters hier sogar gleich sein können. Ob du dies übersichtlich findest oder nicht bleibt dir überlassen.
       * Weiters siehst du, dass der Typ mit `unique_ptr` als rvalue-Referenz (und nicht als lvalue-Referenz) übergeben worden ist, denn sonst könnte man diesen nicht auch mit `nullptr` aufrufen können.

    * Trotzdem muss man auch hier nochmals `move(next)` in der Initialisierung vornehmen, denn `next` ist ein lvalue!!!

- Du kannst den Copy-Konstruktor, den Copy-Assignment-Operator und die Move-Pendants löschen. Weiters wird der Destruktor und auch `clear()` nicht mehr benötigt.
- Im Großen und Ganzen sind die restlichen Methoden nur mehr auf `unique_ptr` anzupassen. Zu beachten ist lediglich:
  - Ein `unique_ptr` kann nicht kopiert werden, aber mittels `move()` in eine rvalue-Referenz gebracht und verschoben werden.
  - Ein `unique_ptr` wird am Besten mittels `make_unique` angelegt. Dafür wird dann der Compiler den schon implementierten Konstruktor von `Node` verwenden.
  - Für den `operator«` gehst du den Stack am Besten mittels *roher* Zeiger durch. Den rohen Zeiger erhältst du von einem `unique_ptr` mittels der Elementfunktion `get()`. Es ist auch absolut kein Problem hier rohe Zeiger zu verwenden, da diese genau in dieser einen Funktion nur lesend verwendet werden.
- Die unötigen Tests, d.h. die, die nicht benötigt werden `;-)` müssen ebenfalls entfernt werden.

2. Bis jetzt sieht unsere Struktur in `stack.h` so aus:

```
struct Node {
};

struct Stack {
};
```

   Was bedeutet das? Die Struktur `Node` ist in unserem API (Application Programming Interface) enthalten. Schauen wir uns allerdings die eigentliche Schnittstelle von `Stack` an, dann sehen wir, dass `Node` sinnvollerweise nicht vorkommt. Warum ist `Node` dann in in unserem API? D.h. weg damit!

   Ganz weg geht ja nicht, da die Klasse `Stack` diese direkt für die Implementierung verwendet, aber wir verschieben die Definition von `Node` einmal in den `private:` Teil von `Stack`. Go!

3. Bevor wir uns jetzt an die Implementierung des eigentlichen Rechners wagen, werden wir uns ein Modul `pystring` anlegen, das aus der zur Verfügung gestellten Bibliothek `pystring` eine statische Bibliothek erzeugt, die wir zu unserem Projekt linken können. Also so etwas wie unser `stack` nur mit fremden Source-Code.

4. Als nächstes gehen wir den eigentlichen REPL an.

   a) Dazu legen wir ein Modul `repl` an, das (übungshalber) eine dynamische Bibliothek beinhalten soll. Auch hier werden wir natürlich dies in einem eigenem Verzeichnis realisieren und dieses mit `subdir()` in `meson` einbinden. Wie das geht siehst du wieder im `meson_tutorial`.

      Dieses Modul soll vorerst eine Funktion folgender prinzipieller Gestalt `string input(string prompt=">> ")` enthält, die folgendermaßen funktionieren soll:

      Vor jeder Eingabe soll der Prompt ausgegeben werden.

      Es soll solange von `cin` gelesen werden, solange der Eingabestrom in Ordnung ist, außer es wird etwas eingegeben, das nicht ausschließlich aus Whitespace-Zeichen besteht. Solch eine Eingabe soll ohne Whitespace-Zeichen links und rechts zurückgeliefert werden. Ist der Eingabestrom nicht in Ordnung, z.B. weil das Ende erreicht wurde, dann soll ein Leerstring zurückgeliefert werden. Whitespace-Zeichen am Anfang und Ende sollen ignoriert werden.

      Baue diese Funktion in dein Programm ein und teste durch Ausführen.

b) Hmm, natürlich funktioniert dies in unserem konkreten Fall, aber prinzipiell kann ein Leerstring als Rückgabewert nicht von einer Eingabe, die nur einen Leerstring enthält unterschieden werden. Für uns ist das zwar prinzipiell kein Problem, da wir ja explizit eine Eingabe eines Strings nur aus Whitespace-Zeichen explizit nicht als gültige Eingabe interpretieren, sondern diese Eingabe ignorieren und den Benutzer wieder zu einer Eingabe auffordern.

   Trotzdem werden wir diese Funktion jetzt umbauen, sodass wir prinzipiell diese beiden Fälle unterscheiden können. Stelle daher den Prototypen der Funktion auf folgende Gestalt um: `optional<string> input(string prompt="» ")`, ändere dein Hauptprogramm um und teste, ob die Funktion gleichgeblieben ist.

c) Jetzt können wir noch immer nicht unterscheiden, ob einfach der Eingabestrom geschlossen wurde (also EOF) oder, ob der Eingabestrom in einen Fehlerzustand übergegangen ist.

   Wenn der Eingabestrom geschlossen wurde, dann soll, wie schon programmiert, kein Wert zurückgeliefert werden. Handelt es sich allerdings um einen Fehler, dann wollen wir eine Exception werfen. Welche Exception? Am besten eine anwendungsspezifische Exception.

   Also, schreibe eine Klasse `IOException`, die von `std::exception` abgeleitet ist und vergiss nicht die `what()` Methode zu überschreiben!

d) Entwickeln wir unser Programm weiter "bottom-up": Was wir eigentlich wirklich brauchen bzgl. der Eingabe ist nicht ein String sondern schon die eigentliche Zahl oder das eigentliche Kommando.

   Lege daher eine Aufzählung (`enum class`, eh klar) – z.B. mit dem Namen `Command` – an, die für jedes mögliche Kommando (z.B. `exp` oder `swap`) und auch jeden Operator (also auch für `+`, z.B. `add`) einen Wert enthält.

   Weiters benötigen wir auch eine Möglichkeit, entweder eine Zahl oder ein Kommando zurückliefern kann. Natürlich könnte man

   - eine eigene Struktur verwenden, aber dann wird zuviel Speicherplatz benötigt und man müsste sich merken welches Feld den Wert enthält.
   - ein `union` verwenden, aber das hat den Nachteil, dass es nicht typsicher ist.

   Wir greifen daher auf `std::variant` zurück und greifen wieder einmal in die Trickkiste des "modernen `C++` " und deklarieren uns die folgende Funktion: `std::variant<double, Command> input(std::string prompt="» ")`. Das gibt aber ein Problem, da die *Signatur* von zwei Funktionen beim Überladen *nicht* gleich sein darf (die Prototypen sind ja nicht gleich, aber das zählt nicht).

   Daher werden wir ein weiteres Refactoring starten und die alte Funktion `input` in `raw_input` umbenennen. Damit spiegelt der Name durchaus auch die richtige Funktionalität wider.

   Was soll die neue Funktion `input` genau tun?

   i. Erstens bedient sie sich der Funktion `raw_input`, um einen String vom Benutzer zu erhalten.
   ii. Die Eingabe des Kommandos sollte case-insensitiv erfolgen können.
   iii. Zweitens parst sie den erhaltenen String und liefert die Information gemäß dem Rückgabewert zurück. Handelt es sich um einen ungültigen Wert, dann soll die Exception geworfen werden. Welche Exception? Leite wiederum von `std::exception` eine Klasse (was hältst du von `ParseException`) ab und verwende diese!

   Programmiere die Funktion einmal wie beschrieben!

e) Wie sieht dies jetzt mit deinen Exceptionklassen so aus?

- Du merkst, dass du die Klasse `ParseException` genauso zu implementieren hast wie die Klasse `IOException`. Das schreit förmlich nach einer gemeinsamen Überklasse `ReplException`. Einerseits weil wir dann das DRY Prinzip einhalten und andererseits weil wir für unser Modul eine einzige Klasse haben, mit der alle Exceptions unseres Moduls mit einem Exceptionhandler abgefangen werden können.

  Implementiere daher die Klasse `ParseException` so, dass du diese von deiner neu angelegten Klasse `ReplException` ableitest und zwar so, dass du in der Initialisierungsliste des Konstruktors den Konstruktor der Basisklasse aufrufst.

- Und jetzt weiter, sodass du die Klasse `IOException` ebenfalls von deiner neuen Klasse `ReplException` ableitest. Gerade vorhin hast du gesehen, dass dein Konstruktor eigentlich ziemlich eintönig ausgefallen ist. Besser wäre es, wenn man einen Konstruktor vererben könnte. Hier bietet `C++` an, dass du den Konstruktor der Oberklasse in der Unterklasse "verwenden" kannst: `using ReplException::ReplException;` innerhalb der Klasse `IOException` und alles ist gut. D.h. du implementierst keinen eigenen Konstruktor. Ist zumindest praktisch.

f) Da die Funktion `raw_input()` jetzt nicht mehr im API benötigt wird, werden wir jetzt den Prototypen aus der Datei `repl.h` entfernen!

g) Bei der Implementierung der Funktion `input()` haben wir bemerkt, dass wir bei der Rückgabe des `enum class` Wertes zuerst händisch den eingegebenen String parsen mussten und danach den entsprechenden Aufzählungswert zurückliefern konnten.

   In weiterer Folge müssten wir, wenn wir den Wert ausgeben wollten auch entsprechende Abragen programmieren.

   Das ist erstens unpraktisch und zweitens fehleranfällig. Ok, wir könnten zwei Funktionen schreiben, die diese Funktionalität erfüllen und müssten nur an genau diesen Stellen die Änderungen vornehmen.

   Schöner wäre es, wenn diese Funktionalität in `C++` eingebaut wäre, ist es aber nicht. Wir werden die Bibliothek `magic_enum` verwenden, die uns so eine Funktionalität anbietet, aber `C++17` und einen modernen Compiler wie `g++` in der Version **9** voraussetzt.

   Schreibe daher deine Funktion `input()` dementsprechend um und aktualisiere deinen Testcode in der Funktion `main()`, sodass dieser jetzt auch das eingegebene Kommando ausgibt!

   Achtung: die Operatoren `+`, `-`, `*`, `/` und `**` müssen händisch in die entsprechenden Aufzählungswerten ersetzt werden.

h) Wenn wir uns den Prototypen von `input` ansehen, dann ist der Typ des Rückgabewertes doch recht unhandlich. Hier ist es eindeutig besser einen `using`-Typalias zu verwenden. Also so etwas wie `using Value=std::variant...`! Baue dies in dein Programm ein.

i) Implementiere jetzt die Funktionalität nur für `+`, `-`, `/`, `*` bzw. `add`, `sub`, `div` und `mul` indem du eine Klasse `Calc` schreibst, die über einen Stack und eine Methode `double eval(Value)` verfügt.

   Teste händisch in `main`!

Damit soll unser erster Prototyp fertig sein.

4 Übungszweck dieses Beispiels

- `unique_ptr` verwenden und die Auswirkungen und Konsequenzen kennenlernen
- `make_unique<>()` kennenlernen

- lvalue- und rvalue Referenzen kennenlernen, `std::move()` verwenden
- die weiteren Sequenzcontainer und die Adaptoren der Standardbibliothek kennen lernen
- `pystring` verwenden
- Erstellen und verwenden einer statischen Bibliothek
- `std::optional` einsetzen
- Vertiefung der Streams bzgl. Fehlerbehandlung
- Eigene Exceptionklassen anlegen
- einfache Vererbung
- "Vererbung" von Konstruktoren
- `std::variant` einsetzen
- Signatur vs. Prototyp einer Funktion und Zusammenhang zum Überladen von Funktionen
- Erste Überlegungen zum Design eines API anstellen
- `magic_enum` verwenden
- `using`-Typalias einsetzen