Beispiel `large_rpn_calc`

Dr. Günter Kolousek

17. August 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 **Netbeans** mit CMake!

2 Aufgabenstellung

Bei dieser Aufgabe geht es darum, unseren schon bekannten RPN Taschenrechner auf einen beliebig großen Stack umzubauen. D.h. wir können Code von dem alten Beispiel verwenden. Allerdings soll dieser Taschenrechner bzgl. der Implementierung nicht auf einem Array basieren sondern auf einer verketteten Liste.
Zusätzlich zu der alten Funktionalität soll auch noch

- ein `swap` Kommando eingebaut werden, das die oberste Zahl mit der zweitobersten Zahl vertauscht.
- ein `dup` (duplicate) Kommando eingebaut werden, das die oberste Zahl am Stapel dupliziert. D.h., dass nach dieser Operation die oberste Zahl auch an zweitoberster Stelle am Stack vorhanden ist.
- ein `rot` (rotate) Kommando eingebaut werden, das die oberste Zahl vom Stack nimmt und unten in den Stack einschiebt.

Containerdatentypen aus der Standardbibliothek wie `vector` dürfen in diesem Beispiel wiederum *nicht* verwendet werden.

3 Anleitung

1. Beginnen wir zuerst eine einfach verkettete Liste in C/C++ zu implementieren, die wir in weiterer Folge zur Implementierung unseres Stacks heranziehen werden.

   Legen wir dazu folgendes Interface zugrunde:

   ```
   struct Node {
       double data;
       Node* next;
   };


   struct List {
       Node* head;
   };


   List* create_list();  // creates a new (empty) list
   void delete_list(List* lst);  // deletes the whole list
   // appends element to end and returns appended Node-instance
   Node* push_back(List* lst, double data);
   Node* push_front(List* lst, double data);  // insert element at beginning
   Node* insert(List* lst, int idx, double data);  // insert element at index
   Node* back(List* lst);  // gets access to last Node-instance
   Node* pop_back(List* lst);  // removes and returns last element
   Node* front(List* lst);
   Node* pop_front(List* lst);  // removes and returns first element
   Node* erase(List* lst, int idx);  // removes and returns element at index
   unsigned int size(List* lst);
   bool empty(List* lst);  //
   Node* get(List*, int index);  // get Node instance at giben index
   void clear(List* lst);  // clears the contents of the list
   ```

   Kann eine Operation nicht erfolgreich durchgeführt werden, wird `nullptr` zurückgeliefert.

   Schaue dir einmal dieses Interface (Schnittstelle) für unser Listenmodul an. Wie könntest du das implementieren? Was wären die Vor- und die Nachteile? Gehe danach zum nächsten Punkt.

2. Ok, analysieren wir einmal dieses Interface.

   Vorerst eine einfache Betrachtung: Es soll eine einfach verkettete Liste mit einem Anker und ohne Stoppknoten werden. Also etwas ganz Traditionelles, etwas Stinknormales,...

Was hier nicht zu sehen ist (tja das gehört eigentlich **dokumentiert**), dass die Funktionen immer einen Parameter erwarten, der nicht der Nullpointer (`nullptr`) ist! So etwas nennt man eine Vorbedingung (engl. precondition). Also, dann dokumentiere einmal… In der Schnittstelle entsprechenden Kommentar einfügen!

Offensichtlich bekommen wir bei den Funktionen immer einen Pointer auf eine `Node` Instanz zurück. Es gibt verschiedene Arten von Funktionen:

- Die, die eine neue `Node` Instanz anlegen, wie z.B. `append`. D.h. die Funktion `append` wird mit `new` einen neuen `Node` anlegen.
- Diejenigen, die eine schon bestehende `Node` Instanz zurückliefern, wie z.B. `get`.
- Die spezielle Funktion `erase`, die aus der Liste einen Node aushängt und einen Pointer auf den Node zurückliefert.
- Die spezielle Funktion `clear`, die den Inhalt der gesamten Liste löscht.
- Die spezielle Funktionen `create_list` bzw. `delete_list`, die sich um das korrekte Anlegen bzw. Löschen einer Liste kümmern.

Die Problematik bei dieser Schnittstelle ist, dass…

3. Und, weißt was das Problem darstellt? Es muss klar sein wer einen Speicher anfordert und klar festgelegt sein, wer den Speicher wieder freigibt. Ist es so, dass derjenige, der den Speicher anfordert auch der ist, der den Speicher wieder freigibt, dann sind die Verantwortlichkeiten klar geregelt und das ist gut.

   Das Problem bei der obigen Schnittstelle ist, dass offensichtlich die entsprechenden Listenfunktionen den Speicher anfordern und `clear` auch den Speicher wieder freigibt, aber bei `erase` ist alles anders. `erase` kann zwar die relevante `Node` Instanz aus der Liste aushängen, aber den Speicher nicht freigeben, da ja ein Pointer auf diese `Node` Instanz zurückliefert. Das bedeutet, dass der Aufrufer für das Freigeben dieses Speichers verantwortlich ist. Und diese zweigeteilte Verantwortlichkeiten sind nicht gut.

   Das kann man relativ leicht beseitigen, indem man den Prototypen von `erase` wie folgt umbaut:

   ```
   // removes node at given index and returns true if it was successful
   bool erase(List* lst, int idx);
   ```

   Das würde gehen, aber hübsch sind diese zwei verschiedenen Semantiken in einer Schnittstelle nicht.

4. Nehmen wir weiters an, dass wir die `erase` Funktion ausgebessert haben, aber dann ist es noch immer so, dass die Funktionen jeweils einen Pointer auf eine *intern* verwendete Struktur zurückliefern. Das ist in unserem Fall aber gar nicht notwendig und birgt zusätzlich noch die Gefahr, dass der Aufrufer entweder

- den Speicher freigibt (obwohl er gar nicht dürfte). Erschwerend kommt hinzu, dass damit der Speicher auch zweimal freigegeben werden kann, das überhaupt nicht vorkommen darf!
- auf den Speicher zugreift, obwohl dieser gar nicht mehr exisistiert, wie dies nach dem Aufruf von `clear` oder `erase` durchaus sein könnte.

Aus diesen Gründen sehen wir, dass eine derartige Schnittstelle für diesen Fall absolut ungeeignet ist.

5. Das führt uns dazu das Interface folgendermaßen zu gestalten:

```
struct List;

List* create_list();  // creates a new (empty) list
void delete_list(List* lst);  // deletes the whole list
void push_back(List* lst, double data);  // appends element to end
void push_front(List* lst, double data);  // insert element at beginning
void insert(List* lst, int idx, double data);  // insert element at index
double back(List* lst);
double pop_back(List* lst);  // removes and returns last element
double front(List* lst);
double pop_front(List* lst);  // removes and returns first element
double erase(List* lst, int idx);  // removes and returns element at index
unsigned int size(List* lst);
bool empty(List* lst);
double get(List*, int index);
void clear(List*);
```

Was sehen wir hier?

- Das keine Pointer auf interne `Node` Instanzen zurückgeliefert werden.
- Das auch keine Deklaration von `Node` mehr im Interface notwendig (und sinnvoll) ist. Es handelt sich um reines Implementierungsdetail. Demzufolge ist die Definition dieser Struktur in `sllist.cpp` anzugeben.
- Das auch keine vollständige Definition der Struktur `List` mehr vorhanden ist, sondern nur mehr eine Forward-Deklaration, die einfach nur festlegt, dass es eine Struktur mit dem Namen `List` gibt, aber nicht wie diese aussieht. Die Definition der Struktur hat in der `sllist.cpp` zu erfolgen.
- Es so nicht klar, wie in einem Fehlerfall vorgegangen werden soll. Wir legen als Teil des Interfaces fest, dass in diesem Fall eine Exception geworfen werden soll. Wir legen uns der Einfachheit halber fest, dass wir wiederum `std::logic_error` verwenden.

So, jetzt hast du alle Informationen, um das Modul zu implementieren. Schreibe dazu gleich auch Testcode in `main.cpp`. Außerdem ist es jetzt an der Zeit auch den Debugger in Netbeans anzuwerfen und auszuprobieren.

Hier trotzdem ein Vorschlag zur Reihenfolge der Implementierung der Funktionen:

a) `create`

b) `size`

c) `push_back`

d) `back`

e) `push_front`

f) `front` (teste auch auf leerer Liste)

So *könnte* die Testausgabe bis jetzt aussehen:

```
size: 0
front on empty list: no elements in list
pushed to the end: 1
size: 1
front: 1
back: 1
pushed to back 2
size: 2
front: 1
back: 2
pushed to back: 3
size: 3
front: 1
back: 3
pushed to front: 0
size: 4
front: 0
back: 3
```

Siehst du dir die Ausgaben an, dann kannst du sehen welche Tests sinnvoll sind.

g) `pop_front`

h) `clear`

i) `empty`

j) `get`

Eigentlich kann es ja keine negativen Indizes geben, aber mit der Semantik, die auch in Python implementiert ist, machen negative Indizes sehr wohl Sinn. Erinnerst du dich noch wie das mit `lst[-1]` in Python funktioniert?

.....

Also: `lst[-1] = lst[len(lst) - 1]`

Implementiere das auch, denn das ist praktisch.

Die zusätzlichen Testausgaben *könnten* folgendermaßen aussehen:

```
popped from front: 0
size: 3
front: 1
back: 3
cleared whole list!
size: 0
front on empty list: no elements in list
back on empty list: no elements in list
list is empty: 1
pushed to the front: 3
list is empty: 0
pushed to front: 2
pushed to front: 1
element at index 0: 1
element at index 1: 2
element at index 2: 3
access to element at index 3: index 3 not in list
element at index -1: 3
element at index -2: 2
element at index -3: 1
access to element at index -4: index -1 not in list
```

k) `insert`

l) `pop_back`

m) `erase`

Die weitere Testausgabe *könnte* folgendermaßen aussehen:

```
element 0 inserted at index 0
element at index 0: 0
element at index 1: 1
element at index 2: 2
element at index 3: 3
element -4 inserted at index 2
element at index 0: 0
element at index 1: 1
element at index 2: -4
element at index 3: 2
element at index 4: 3
element -5 inserted at index 4
element at index 0: 0
element at index 1: 1
element at index 2: -4
```

```
element at index 3: 2
element at index 4: -5
element at index 5: 3
popped element from back: 3
element at index 0: 0
element at index 1: 1
element at index 2: -4
element at index 3: 2
element at index 4: -5
popped element from back: -5
popped element from back: 2
popped element from back: -4
popped element from back: 1
popped element from back: 0
size: 0
pushed to the front: 4
pushed to the front: 3
pushed to the front: 2
pushed to the front: 1
erased element at index 0: 1
element at index 0: 2
element at index 1: 3
element at index 2: 4
size: 3
erased element at index 1: 3
element at index 0: 2
element at index 1: 4
size: 2
erased element at index 1: 4
element at index 0: 2
size: 1
```

6. Nachdem du jetzt weißt, dass deine Implementierung der Liste funktioniert, kannst du dich der Implementierung des eigentlichen RPN Rechners widmen.

   Nimm jetzt den alten Code deines `srpn` her und kopiere diesen in dein aktuelles Projekt. Allerdings schlage ich vor, dass du den Code zum Testen der Liste nur auskommentierst. Vergiss nicht auf die zusätzlichen Funktionen deines Rechners. Und damit sind wir auch schon wiederum am Ende.

   Achte auch darauf, dass solche Fälle den Stack nicht verändern, wenn nicht genug Zahlen am Stack sind!

Fertig!

4 Übungszweck dieses Beispiels

- einfache Strukturen
- einfach verkettete Liste mit einem Anker
- Problematik des manuellen Speicherhandlings
- Pointer üben
- debugging