ulm university universität
uulm

Institut für Numerische Mathematik
Dr. Andreas F. Borchert und Dr. Michael C. Lehn

19. Januar 2017
Blatt 11

Objektorientierte Programmierung mit C++ (WS 2016/2017)

Abgabe bis zum 26. Januar 2017, 16:00 Uhr

Lernziele:

- Entwicklung dynamischer Datenstrukturen unter Verwendung intelligenter Zeiger
- Entwicklung von Unterklassen zur Navigation in der übergeordneten Klasse

Aufgabe 13: Navigieren in Tries

Die zuvor erarbeitete Implementierung für die Trie-Datenstruktur sah noch keine Navigationsmöglichkeit vor. Das Problem war, dass keine Zeiger, seien sie direkt oder indirekt, auf die innere Baumstruktur nach außen vergeben konnten, ohne in Schwierigkeiten zu geraten bei der Freigabe der Datenstruktur im Dekonstruktor.

Dieses Problem lässt sich jedoch mit intelligenten Zeigern vermeiden. Im Rahmen dieser Übungsaufgabe können Sie ausgehend von der Ihnen selbst entwickelten Lösung zur Aufgabe 8 oder der Fassung, die am 15. Dezember in den Übungen vorgestellt worden ist, eine Version von *Trie.hpp* entwickeln, die mit intelligenten Zeigern arbeitet und indirekte Zeiger zur Navigation unterstützt.

Konkret ist

- *Trie* auf intelligente Zeiger umzustellen (hierbei ist *std*::*shared_ptr* zu verwenden) und
- eine Unterklasse *Trie*::*Pointer* von *Trie* zu entwickeln, die das Navigieren unterstützt, ohne dabei die internen Zeiger preiszugeben.

Folgendes Code-Beispiel demonstriert die gewünschte Funktionalität. Sei *trie* ein bereits gefüllter *Trie* des Typs *Trie*<*std*::*string*> und *key* eine Zeichenkette des Typs *std*::*string*. Dann gibt das Beispiel alle gefundenen Wörter aus, die aus einem Präfix von *key* bestehen:

```
Trie<std::string>::Pointer ptr = trie.descend();
if (ptr.defined()) {
    for (char ch: key) {
```

```
        if (!ptr.descend(ch)) break;
        if (ptr.exists()) {
            cout << *ptr << endl;
        }
    }
}
```

trie.descend() liefert einen Zeiger auf die Wurzel. Mit der Methode *defined* lässt sich überprüfen, ob der Zeiger ungleich **nullptr** ist; mit *descend* ist ein Abstieg in den Präfixbaum mit dem gegebenen Zeichen möglich, wobei der zurückgegebene **bool**-Wert signalisiert, ob der Zeiger dadurch zum **nullptr** wurde. Mit *exists* lässt sich überprüfen, ob an dem erreichten Knoten im Präfixbaum ein Objekt existiert. Wenn ja, kann dieses mit einer Dereferenzierung abgerufen werden. Ausgehend von den Wörtern in */usr/dict/words* würde beispielsweise der Schlüssel „schoolgirlish“ zur Ausgabe der Wörter „s“, „SC“, „school“, „schoolgirl“ und „schoolgirlish“ führen.

Zu beachten ist bei der Implementierung, dass ein Teilbaum, auf den ein *Trie*::*Pointer* zeigt, dank der intelligenten Zeiger überlebt, selbst wenn der übergeordnete *Trie* bereits freigegeben wurde. Die Methode mit den Referenzzählern funktioniert hier, weil es nur Zeiger in die Abstiegsrichtung gibt, nie zurück.

Sie können wie üblich Ihre Lösung wieder einreichen:

```
thales$ submit cpp 13 Trie.hpp TestTrie.cpp
```

Aufgabe 14: Navigieren in Tries II

Mit dem *Trie*::*Pointer* lässt sich noch nicht auf einfache Weise ermitteln, welche weitergehenden Unterbäume zur Verfügung stehen, d.h. welche Zeichen zu einem Unterbaum führen. Dies könnte mit einem Iterator des Typs *Trie*::*Pointer*::*Iterator* geschehen. Hier ist ein Code-Beispiel, mit dem über die weiterführenden Zeichen iteriert wird:

```
if (ptr.defined()) {
    std::cout << "continuation␣possible␣with:␣";
    for (char ch: ptr) {
        std::cout << ch;
    }
    std::cout << std::endl;
}
```

Damit das Iterieren mit einer **for**-Schleife klappt, müssen die folgenden Bedingungen erfüllt werden:

- *Trie*::*Pointer* muss die Methoden *begin* und *end* unterstützen, die jeweils einen Iterator liefern.
- Der Iterator muss den Operator „!=“ unterstützen:
  bool operator!=(**const** *Iterator*& *other*)**const**

- Der Iterator muss das Prefix-Inkrement unterstützen:
  Iterator& **operator**++()
- Der Iterator muss die Dereferenzierung unterstützen:
  char operator∗()**const**

Wenn Sie Iteratoren haben, können Sie mit deren Hilfe das folgende Rätsel lösen: Welches Wort aus */usr/dict/words* enthält die meisten Präfixe, die zugleich auch Wörter aus der gleichen Sammlung sind?

Wie üblich kann die Lösung wieder eingereicht werden:

```
thales$ submit cpp 14 Trie.hpp TestTrie.cpp
```

Viel Erfolg!