Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Prof. Dr. Christoph Stamm

Übung 4: STL-Container

Lernziele

- Sie repetieren den Stoff aus der Vorlesung: Streams, Templates, STL, Containers, Iterators.
- Sie entwickeln einen persistenten Container auf Basis eines Random-Access-Files.
- Sie testen Ihre Container-Klasse mit Unit-Tests.

1 Einführung in das Thema

In dieser Übung geht es darum, sich mit Teilen der Standardbibliothek auseinanderzusetzen. Dies schliesst sowohl die Arbeit mit Streams als auch mit der Standard-Template-Library ein. Ausserdem soll das Programmieren mit STL-Containern und Iteratoren mit all ihren Fallstricken eingeübt werden.

Die beste Art sich damit zu befassen, ist die Entwicklung eines eigenen STL-Containers. Um eine echte Ergänzung zu STL anzubieten, sollen Sie einen persistenten STL-Container mit Iteratoren aufbauen, dessen Hintergrundspeicher sich in einer Datei befindet.

Die Standardbibliothek von C++ bietet mit den File-Streams die Möglichkeit an, auf Dateien zuzugreifen. Allerdings ist der Austausch von Daten auf die Zeichentypen `char` und `wchar_t` beschränkt, und das Funktionsangebot kann nicht gerade als hochstehend bezeichnet werden. Daher stellen wir Ihnen eine Adapter-Klasse für `fstream` zur Verfügung, deren Funktionalität sich an derjenigen der Java-Klasse `RandomAccessFile` anlehnt.

Der Konstruktor von `RandomAccessFile` öffnet eine bestehende Binärdatei mit gegebenen Dateinamen oder erstellt eine neue Binärdatei, falls die Datei unter dem gegebenen Pfad nicht gefunden werden kann. `RandomAccessFile` soll das Lesen und Schreiben aller primitiven Datentypen über generische Funktionen ermöglichen.

1.1 Teilaufgaben

Die Übung setzt sich aus folgenden Teilaufgaben zusammen:

1. Ergänzen der gegebenen Adapter-Klasse `RandomAccessFile` um die generischen Funktionen `read` und `write`.

2. Auf der Basis des `RandomAccessFiles` soll ein STL-Container `PersistentVector` gebaut werden, dessen Interface sich an die Klasse `vector` anlehnt.

3. Die Klasse `PersistentVector` soll um einen eigenen Iterator ergänzt werden, der sich so bedienen lässt, wie Sie es von STL-Iteratoren kennen.

4. Die korrekte Funktionsweise der Klasse `PersistentVector` soll mittels UnitTests nachgewiesen werden.

1.2 Generisches Lesen und Schreiben von Binärdaten

Die Schnittstellen der beiden generischen Methoden `read` und `write` sind bereits gegeben. Beide Methoden besitzen einen Parameter `pos` mit dem Vorgabewert `-1`. Ein positiver Wert von `pos` gibt die Stream-Position in Bytes an, wo gelesen bzw. geschrieben werden soll. Ein negativer Wert von `pos` bedeutet, dass sequentiell gelesen bzw. geschrieben werden soll. Auf diese Art und Weise können `RandomAccessFiles` sowohl für den wahlfreien als auch den sequentiellen Zugriff verwendet werden.

Implementieren Sie die generischen Methoden `read` und `write`. Sollte das Lesen oder Schreiben von Binärdaten aus einem Grund nicht möglich sein, so sollen die beiden Methoden eine `RandomAccessFile::IOException` werfen.

1.3 Persistenter Vektor

In dieser Teilaufgabe sollen Sie einen persistenten Container implementieren, einen `PersistentVector`, welcher die Elemente in einem `RandomAccessFile` speichert.

In der MSDN-Library finden Sie das Funktionsangebot eines regulären Vektors. Zusätzlich stellt die Klasse mehrere verschiedene Typen und Operatoren zur Verfügung. Implementieren Sie die generi-

sche Klasse `PersistentVector`, wobei Sie das Funktionsangebot des regulären Vektors als Vorbild nehmen (lesen Sie die entsprechenden Spezifikationen). Methoden, welche mit Iteratoren zu tun haben, können Sie vorerst beiseitelassen.

Da ein `PersistentVector` seine Daten nicht im Hauptspeicher, sondern in einer Datei hält, ist es nicht möglich, eine Referenz auf ein Element zu bekommen. Dies führt zu gewissen Änderungen in den Funktionssignaturen des Containers für alle Methoden, die eine Referenz auf ein Element zurückgeben. Anstatt einer Referenz auf ein Vektorelement verwenden wir eine Instanz eines `VectorAccessors`. Die Klasse `VectorAccessor` müssen Sie selber entwickeln. Sie verwaltet mindestens einen Zeiger auf einen `PersistentVector` und den Index, wo der Zugriff stattfinden soll. Durch geschicktes Überladen des Zuweisungsoperators und des Typumwandlungsoperators kann damit der lesende und schreibende Zugriff auf die Datei ermöglicht werden.

1.4 Iterator

Ergänzen Sie die Klasse `PersistentVector` um einen eigenen bidirektionalen Iterator. Implementieren Sie dazu eine generische Klasse `VectorIterator`. Orientieren Sie sich dabei an der Art und Weise, wie in der STL Iteratoren erstellt und benutzt werden, zum Beispiel:

```
PersistentVector<double> v;
for (int i=100; i ≥ 0; --i) v.pushBack(i);
PersistentVector<double>::iterator a = v.begin();
PersistentVector<double>::iterator z = v.end();
sort(a, z);
```

Diese Zeilen zeigen, dass Iteratoren nicht isoliert existieren, sondern nur in Zusammenhang mit einem entsprechenden Container. Deshalb werden sie auch mit `PersistentVector<T>::iterator` deklariert.

Im ersten Entwurf ist man versucht, die Iterator-Klasse als innere Klasse von `PersistentVector` zu implementieren. Dies führt allerdings zu Problemen mit dem Template-Mechanismus. Es ist deshalb ratsam, den Iterator als eigenständige, befreundete Klasse zu implementieren. Damit diese Iteratoren nicht unkontrolliert instanziiert werden können, ist der Konstruktor privat zu deklarieren.

1.5 Unit-Tests

Entwickeln Sie sinnvolle Unit-Tests, um die Funktionsweise Ihres persistenten Vektors zu überprüfen. Wenn Sie es schaffen, den Sortieralgorithmus der Standardbibliothek korrekt auf Ihrem persistenten Vektor auszuführen, dann haben Sie die Übung mit Bravour bestanden.