C/C++ für Java-Programmierer
Aufgabe 2.3: Templates, Map<> und Map<>::Iterator

```
Map<int,string> m;
m[4] = "vier";
m[7] = "sieben";
m[2] = "zwei";

Map<int,string>::iterator i;
for(i = m.begin(); i!=m.end(); i++) {
  cout << i->first << "/" << i->second << endl;
}
```

Aufgabe 2 besteht aus drei aufeinander aufbauenden Teilaufgaben und kann/soll über mehrere Wochen bearbeitet werden (siehe *Abgabe*). Dieses Blatt enthält die dritte und letzte Teilaufgabe 2.3.

Vorbereitung und Ausblick

In Aufgabe 2.1 implementieren Sie einen Typ `RationalNumber`, in 2.2 eine assoziative Map, die es erlaubt, Paare `<RationalNumber, int>` zu speichern. In 2.3 schließlich verallgemeinern Sie diese Map mittels Templates und fügen Iteratoren hinzu.

Lösen Sie Aufgabe 2.2 zunächst möglichst vollständig, bevor Sie 2.3 implementieren; diese Aufgabe ist vor allem eine Portierung Ihres Codes auf Templates.

Aufgabe 2.3.1: Portierung auf Templates

Legen Sie ein Modul für eine neue Klasse `Map` an, und kopieren Sie dann den entsprechenden Code aus Aufgabe 2.2 in dieses neue Modul. Den Namensraum `rn` können Sie entfernen oder durch einen Namensraum Ihrer Wahl ersetzen. Das neue Modul soll im gleichen Projektverzeichnis wie Aufgabe 2.2 liegen; erweitern Sie das Unit-Testprogramm entsprechend.

Generalisieren Sie die Klasse, indem Sie die beiden "fest eingebauten" Typen `key_type` und `mapped_type` mit Hilfe von Templates generisch machen. Dazu empfehlen sich folgende erste Schritte:

- Vor die Klassendefinition sowie vor jede einzelne Methodendefinition schreiben Sie `template<class KeyT, class T>`.
- In der Klassendefinition (hoffentlich nur in den beiden `typedefs`) ersetzen Sie die beiden "starren" Typen `RationalNumber` und `int` durch die beiden Template-Parameter `KeyT` und `T`.

- Überall dort, wo der Klassen-Name `Map::` als Präfix für einen Methodennamen auftaucht, ersetzen Sie ihn durch den generischen Klassen-Namen `Map<KeyT, T>::`
- Genauso müssen Sie überall dort verfahren, wo `Map` als Typname verwendet wird (z.B. bei Methoden, die Referenzen auf `Map` zurückgeben). Die Methodennamen von Konstruktor und Destruktor werden jedoch nicht verändert.
- Bringen Sie Ihren Code zum Compilieren, bevor Sie weitermachen. Wahrscheinlich wird Ihr Compiler an einigen Stellen, an denen Sie Typausdrücke wie `Map<KeyT,T>::mapped_type` verwenden, Fehler melden. Hier müssen Sie wie in der SU besprochen das Schlüsselwort `typename` einfügen, damit der Compiler weiß, dass ein Typ und kein Attribut- oder Methodenname gemeint ist.

- Kopieren Sie Ihre Tests für `rn::Map` und portieren Sie sie für `Map<>`, indem Sie einfach den Typ der erzeugten Objekte von `rn::Map` auf `Map<rn::RationalNumber,int>` ändern.
- Wenn Sie nun versuchen zu kompilieren, erhalten Sie eine Fehlermeldung vom Linker (z.B. "symbols not found"). In "Compile Output" sehen Sie, dass beim Übersetzen von `main.cpp` die Implementierungen der in der `.cpp`-Datei definierten Template-Methoden nicht bekannt sind (siehe Handout).
- Wenden Sie die im SU vorgestellte Inklusions-Lösung an, d.h. binden Sie die Datei mit den Template-Definition in Ihre Header-Datei ein. Dabei empfiehlt es sich, die `.cpp`-Datei umzubenennen, z.B. in `_map.h`. Im allgemeinen Fall soll verhindert werden, dass diese Datei wie andere `.cpp`-Dateien separat kompiliert wird; sie soll vom Build-System eher wie eine Header-Datei verstanden werden.
- Fügen Sie weitere Tests mit anderen Template-Parametern hinzu, z.B. `int/string`, `string/string`, o.ä.

Aufgabe 2.3.2: Iteratoren

Ein richtiger C++-Container ist nichts ohne Iteratoren. Z.B. ist es noch nicht leicht möglich, in der Map in der richtigen Reihenfolge vom ersten bis zum letzten Element zu laufen, oder von einem Element effizient zum nächsten im Baum zu springen.

Iteratoren sind Objekte, die mit der jeweiligen Containerklasse "intim" verbunden sind; im Fall der Map bedeutet dies, dass der Map-Iterator den Typ `KeyValueNode` kennen darf. Implementieren Sie den Iterator als innere Klasse von `Map`.

`Map::Iterator` ist eine Klasse, die einen Zeiger auf einen `KeyValueNode` (im folgenden kurz `Node`) verwaltet. D.h. der Iterator kennt den Baumknoten, auf den er verweist. Die Klasse besitzt folgende Methoden:

- **Konstruktor Iterator(Node* node=0):** erzeugt einen Iterator, der auf den übergebenen Knoten in der Map verweist.
- **Zuweisungsoperator, Copy-Konstruktor:** Da der Iterator keinen eigenen Speicher dynamisch verwaltet, können hier die Standard-Operatoren verwendet werden.

- **Vergleich zweier Iteratoren mittels == und !=:** zeigen die Iteratoren auf den gleichen Knoten in der gleichen Map?
- **Dereferenzierung mittels *:** liefert eine Referenz auf den Datenwert des vom Iterator referenzierten Knotens vom Typ `Map::value_t`. Im Fall einer Map ist der Rückgabewert üblicherweise ein Key-Value-Paar! Siehe Anmerkungen zum Typen `Map::value_t` weiter unten!
  - *Wenn wie bei dem Beispiel `*(Map::end())` ein nicht gültiger Knoten dereferenziert wird, ist das Verhalten undefiniert (d.h. Sie müssen sich hier nicht darum kümmern, das Programm darf dann abstürzen)*
- **Dereferenzierung mittels ->**: wie `*`, jedoch liefert dieser Operator einen Zeiger auf das Paar, nicht eine Referenz.
- **Inkrement mittels `operator++()`** bzw. **`operator++(int)`:** nach dieser Operation soll der Iterator auf das nächste Element im Sinne der Ordnung der Elemente verweisen.
  - *Achtung(1) - diese Operation ist nicht ganz trivial - testen Sie Ihren Algorithmus ausgiebig mit nicht-trivalen Baum-Strukturen!*
  - *Achtung(2): Der Post-Inkrement-Operator verändert den Wert des Iterators, liefert aber eine Kopie des alten Iterators vor der Veränderung zurück - siehe Handout.*

Die zu implementierenden Iterator-Methoden der Klasse `Map` orientieren sich an einer Teilmenge des Iterator-Schnittstelle von `std::map` und sind wie folgt definiert:

- **`Map::iterator`** ist ein `typedef`-Alias für Map::Iterator
- **`Map::value_t`** ist ein `typedef`-Alias für den Typ der Nutzdaten, den ein Map-Knoten trägt, in der STL ist das: **`std::pair<key_type, mapped_type>`** . Dieser Typ ist wichtig, weil er als Rückgabetyp bei der Dereferenzierung eines Iterators verwendet wird (siehe unten).
  - Ändern Sie Ihre Implementierung von `KeyValueTree` so, dass dieser intern ein solches Pair zur Speicherung der Nutzdaten verwendet und zurückliefert.
- **`Map::begin()`** liefert einen `Map::iterator`, der auf das erste Element verweist. Hierzu muss im Baum das kleinste Element gefunden werden.
- **`Map::end()`** liefert einen `Map::iterator`, der konzeptionell hinter das letzte Element verweist. Tatsächlich verweist er auf keinen Knoten oder einen speziellen Knoten, der nicht im Baum aufgehängt ist. Mehr dazu unter `operator->()` und `operator*()`.

Achten Sie bitte unbedingt darauf, dass die Map keine Iteratoren mit **`new`** erzeugt, die dann nicht wieder gelöscht werden! Der Iterator ist ein sehr leichtgewichtiger Typ, der ruhig auch mal kopiert werden kann, anstatt ihn dynamisch zu erzeugen.

Hier sind ein paar Beispiele der Einsatzmöglichkeiten:

```
Map<int,string> m;
m[4] = "vier";
m[7] = "sieben";
m[2] = "zwei";
Map::iterator i = m.begin();
cout << "kleinstes Element: " << i->first << "/"
     << i->second << endl;
cout << "weitere Elemente, sortiert: " << endl;
while(i!=m.end()) {
  i++;
  pair<int,string> p = *i;
  cout << "  " << p.first << "/"
       << p.second << endl;
}
m.begin()->second = "neuer Wert";
```

Jetzt gibt es noch ein letztes Problem: mit der aktuellen Konstruktion ist es möglich, über einen Iterator auch den Key in einem Knoten des Suchbaums zu verändern!

```
m.begin()->first = "9"; // PROBLEM!!!
```

Nach einer solchen Anweisung stimmt potentiell die Sortier-Reihenfolge im Suchbaum nicht mehr (testen Sie es - Ihre schöne Baum-Ordnung kann über den Iterator zerstört werden).

Was tun? Vergleichen Sie die Definition des `typedef`-Alias `std::map::value_t` mit der von `Map::value_type`. Was fällt auf? Ändern Sie Ihren Code entsprechend und stellen Sie sicher, dass der "böse" Code nicht mehr compiliert.

Abgabe und Demonstration

Die Abgabe der gesamten Aufgabe 2 soll bis zu dem in Moodle festgelegten Termin erfolgen. Verspätete Abgaben werden wie in den Handouts beschrieben mit einem Abschlag von 2/3-Note je angefangener Woche Verspätung belegt. Geben Sie bitte pro Gruppe jeweils nur eine einzige .zip-Datei mit den Quellen Ihrer Lösung ab. Bitte keine Kompilate in der Abgabe!

Demonstrieren und erläutern Sie dem Übungsleiter Ihre Lösung in der Übungen nach dem Abgabedatum. Die Qualität Ihrer Demonstration ist, neben dem abgegebenen Code, ausschlaggebend für die Bewertung! Es wird erwartet, dass alle Mitglieder einer Gruppe anwesend sind und Fragen beantworten können.