Assignment 8 - ForwardList

In dieser Aufgabe übst Du weiter den Umgang mit Templates. Diesmal implementierst Du eine Container-Datenstruktur. Das Klassen-Template `ForwardList` soll eine einfach verkettete Liste implementieren, die Elemente von variablem Typ speichern kann. `ForwardList` hat eine interne Klasse `Node`, die einen Knoten in der Liste repräsentiert. Die Klasse `ForwardList` stellt um diese Knoten herum einen Teil des Interfaces bereit, den man von einer Standard-Library-konformen Container-Datenstruktur erwarten würde. Dazu gehören auch die Funktionen `begin()` und `end()`, die Iteratoren zurückgeben. Zu diesem Zweck hat `ForwardList` eine zweite interne Klasse `Iterator`, die einen einfachen Iterator implementiert.

Folgende Aufgaben musst Du erfüllen:

1. Implementiere das Klassen-Template `ForwardList`. Die Signaturen der Methoden und ihr gefordertes Verhalten sind bereits angegeben. Die interne Klasse `Node` ist bis auf die Angabe des Typen der Elemente bereits vollständig implementiert.
2. Implementiere die Klasse `ForwardList::Iterator`. Dies ist ein einfacher Iterator, der die allgemeinsten Operationen unterstützt und es ermöglicht, `ForwardList` in einer range-for Schleife zu benutzen. Die Signaturen der Methoden sind bereits angegeben.
3. Stelle einen eigenen Test in `main.cpp` bereit, der beide Klassen testet.

⚠ Ab dieser Woche akzeptieren wir nur noch Einreichungen, die keine Warnings beim Kompilieren erzeugen (gcc-Flags `-Wall` und `-pedantic`).

Hinweise

- Die Klasse `Node` ist so implementiert, dass einem `Node` immer alle daran hängenden `Node`s "gehören". Ein `Node` löscht also immer alle seine Nachfolger, wenn er zerstört wird. Dieses Verhalten solltest Du beachten und ausnutzen, wenn Du `ForwardList` implementierst. Achte in diesem Zusammenhang auch unbedingt auf Valgrind-Fehlermeldungen!
- Der Copy-Assignment-Operator von `ForwardList` ist bereits nach dem Copy- and-Swap-Idiom implementiert. Du implementierst daher nicht den Operator selbst, sondern die `swap`-Funktion und den Copy-Konstruktor.
- Tatsächlich sind Iteratoren ein bisschen komplizierter: Wir schränken uns hier auf ein einfaches Beispiel ein. Dieser Iterator wird *nicht* vollständig konform sein zu den Forderungen der Standardbibliothek, aber er genügt für eine range-for Schleife.
- `ForwardList::Iterator` erlaubt immer beides: Schreib- *und* Lesezugriff. Wenn zusätzlich die Einschränkung auf nur Lesezugriff *ohne* Schreiben – also "`const`" – gefordert wäre, dann wäre dafür eine eigene Iterator-Klasse notwendig. (Das brauchen wir hier nicht!)

- Wenn in der Beschreibung einer Methode steht, dass etwas *undefined behavior* ist, bedeutet das, dass die Methode nicht überprüft, ob dieser Fall auftritt, und auch keine Fehlerbehandlung dafür machen muss.

Optional

Schau Dir die folgenden Regeln aus den C++ Core Guidelines an und versuche, sie beim Implementieren Deiner Lösung umzusetzen.

- C.33: If a class has an owning pointer member, define a destructor
- C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&
- C.62: Make copy assignment safe for self-assignment
- C.84: A swap function may not fail
- T.3: Use templates to express containers and ranges