Übungsblatt-(Programmieren-in-C++)

“Anfangen im Kleinen, Ausharren in Schwierigkeiten, Streben zum Großen.”

– Friedrich Alfred Krupp -

Einige Hinweise zur Bearbeitung dieser Übungsaufgaben

Beginnen Sie jede Datei mit einem Header analog zu diesem:

```
/**
 @file    file name
 @author  your name
 @date    version
 @brief   a short description of the program
*/
```

Kommentieren Sie Ihre Programme auch ansonsten wie gewohnt ausführlich. Nutzen Sie Doxygen-Kommentare. Eine Doxygen-Dokumentation muss nicht abgegeben werden.

Die Idee zu dieser Übungsaufgabe stammt aus einer Lehrveranstaltung des Studiengangs Informatik der TU Dresden. Das m.E. Tolle an der Aufgabe ist, dass viele wesentliche Grundlagen und Besonderheiten von C++ zur Lösung beachtet werden müssen. Damit ist die Bearbeitung dieser Aufgabe eine sehr gute Nachbereitung zu Ihrer eigenständigen Einarbeitung in die Grundlagen von C++.

Notieren Sie das jeweilig genutzte Sprach-Feature in Form eines Kommentars im Quellcode Ihres Programmes!

Zur Abgabe

Alle Aufgaben sind Pflichtaufgaben. Jeder muss die Aufgaben selbständig bearbeiten. Gruppenabgaben sind nicht erlaubt.

Abzugeben ist ein Zip-File mit folgenden drei Dateien: `Node.h`, `Node.cpp`, `TreeUI.cpp` (siehe Aufgabenstellung zu den Details). Es dürfen keine Projekte (Eclipse, Qt, XCode,….) abgegeben werden!

Der Quellcode muss plattformunabhängig kompilierbar sein. Verwenden Sie also keine betriebssystemspezifischen Funktionen o.ä.. **Als C++-Version ist die Version ISO/IEC 14882:2003 zu verwenden (wie auf dem NAO zur Verfügung stehend).**

Die Abgabe erfolgt in unserem Moodle-Kurs. Der konkrete Abgabetermin wird ebenfalls dort angegeben.

Abgrenzung

Folgende Themen werde in dem Übungsblatt nicht behandelt: Vererbung, dynamische Bindung, Überschreiben von Methoden, Mehrfachvererbung, polymorphe Funktionen, Definition eigener Templates.

Teil 1 - Implementieren Sie die folgende Baumdatenstruktur:

1. **Grundgerüst der Baumdatenstruktur:** Erstellen Sie ein neues Projekt, zum Beispiel `EmbCPP_TREE`, mit zwei neuen Dateien `Node.h` und `Node.cpp`. Implementieren Sie eine Baumdatenstruktur wie folgt. Die Knotenklasse `Node` soll einen Namen vom Typ `std::string` speichern können und folgende Methoden bereitstellen (Typen der Argumente und Rückgabewerte sind bewusst weggelassen und müssen erschlossen werden):
   - Konstruktor mit einem Argument vom Typ `const std::string&`, der den Knotennamen initialisiert. Um welche Art Parameterübergabe handelt es sich hier? Warum ist diese Art der Übergabe eines Objektes als Parameter günstig? Notieren Sie Ihre Antwort als Kommentar am Konstruktor. (im Header)
   - Destruktor zum Löschen aller Kindknoten mit dem `delete`-Operator. Deklarieren Sie den Destruktor als virtuelle Methode. Notieren Sie in einem Kommentar, was das bedeutet. (im Header)
   - `getName() const` … gibt den Namen des Knotens zurück. Was bewirkt das Schlüsselwort `const`? Notieren Sie Ihre Antwort als Kommentar (im Header).
   - `setName(name)` ... setzt den Namen des Knotens auf einen neuen Namen.
   - `getNrOfChildren() const` ... gibt die Anzahl der direkten Kindknoten an.
   - `getChild(i) const` ... gibt einen Zeiger auf den i-ten direkten Kindknoten zurück. Was sind gültige Werte für i? Testen Sie, was passiert, wenn i einen nicht gültigen Wert übergibt. Notieren Sie Ihre Antwort (wieder im Header).
   - `addChild(child)` ... fügt am Ende einen neuen direkten Kindnoten hinzu.

   Nutzen Sie zum Speichern der Kindknotenzeiger die Template Klasse `std::vector` der Standard Template Library, die im Header `<vector>` deklariert ist.

2. **Programmstruktur:** Erstellen Sie eine dritte Datei `TreeUI.cpp`. Binden Sie `Node.h` mit einem entsprechenden `#include`-Befehl ein. Implementieren Sie eine `main`-Funktion, die einen Baum mit einem Wurzelknoten namens "`root`" und zwei Kindern namens "`left child`" und "`right child`" erzeugt und danach den ganzen Baum, mit dem `delete`-Operator angewendet auf den Wurzelknoten, wieder löscht.

3. **Debugging**: Setzen Sie im Destruktor der Knotenklasse einen Break-Point und starten Sie die Anwendung in der Debug-Konfiguration im Debug-Modus. Beobachten Sie, wie der Destruktor rekursiv aufgerufen wird. Erweitern Sie den Destruktor so, dass folgende Ausgabe entsteht (nicht vergessen, `<iostream>` zu inkludieren):

```
enter ~node() of "root"
enter ~node() of "left child"
leave ~node() of "left child"
enter ~node() of "right child"
leave ~node() of "right child"
leave ~node() of "root"
```

   Begründen Sie im Quellcode (in der main-Methode) die Reihenfolge der oben gelisteten Ausgaben.

   Ergänzen Sie nun in `Node.h` die Anweisung `#define DEBUG`. Nutzen Sie dies, um sich die erzeugten Ausgaben nur in einem `DEBUG`-Modus ausgeben lassen zu können (Ausgabe nur, wenn `DEBUG` definiert ist).

Ergänzen Sie auch eine Debug-Ausgabe, die Ihnen mitteilt, wenn ein neuer Knoten angelegt wurde:

```
New node created: root
New node created: left child
New node created: right child
```

Lassen Sie die Debug-Ausgaben für die Projektabgabe eingeschaltet.

Teil 2 - Implementieren Sie eine rekursive Traversierung der Struktur wie folgt:

4. **Globale Knotenzählung**: Erweitern Sie die Knotenklasse um eine statische Variable `nodeId`, die eine globale Knotennummer mitzählt. Initialisieren Sie diese in `Node.cpp` auf 0 und zählen Sie sie im Knotenkonstruktor um eins hoch. Geben Sie dem Namensparameter im Knotenkonstruktor einen leeren String als Defaultparameterwert und setzen Sie im Konstruktor den Knotennamen auf `"node_<nodeId>"`, falls kein Knotenname angegeben wurde. Dabei sollen die automatisch erzeugten Knotennamen mit `"node_1"` beginnen. Nutzen Sie zur Umwandlung der Knotennummer in einen String die `std::stringstream`-Klasse aus dem Header `<sstream>`, die wie folgt verwendet wird:

```
std::stringstream strSm;
strSm << nodeId;
std::string nodeIdStr = strSm.str();
```

   Informieren Sie sich über weitere “Angebote” dieser Standardklasse.

5. **Rekursive Baumerstellung:** Implementieren Sie in `Node.cpp` eine Funktion `createCompleteTree(nrChildNodes, treeDepth)`, die rekursiv einen Baum erstellt, bei dem alle Knoten bis auf die Blattknoten genau `nrChildNodes` Kindknoten haben und bei dem die Pfade von der Wurzel bis zu den Blättern genau `treeDepth` Knoten enthalten (dabei ist der Wurzelknoten mitzuzählen). Deklarieren Sie die Methode in `Node.h` und rufen Sie die Methode mit den Parameterwerten (2,4) in der `main`-Methode auf. Werfen Sie außerdem eine Exception, wenn die Parameterwerte ungültig sind. Rufen Sie die Methode in der main-Methode ein zweites Mal mit den Parameterwerten (2,-1) auf. Behandeln Sie die zu erwartende Exception (Abfangen plus Ausgabe in der Konsole. Kein Programmabbruch!).

6. **Stream-Ausgabe:** Implementieren Sie in der Knotenklasse eine Methode `print(std::ostream & str, ...)`, die einen Baum in dem angegebenen Stream ausgibt. Bei `std::cout` handelt es sich um einen solchen Stream. Bemerkung: “...” steht hier als Platzhalter für eventuell weitere, notwendige Parameter. Folgende Ausgabe soll entstehen, wenn man den Wurzelknoten des von `createCompleteTree(2,4)` erzeugten Baumes ausgibt:

```
node_1
   node_2
      node_3
         node_4
         node_5
      node_6
         node_7
         node_8
   node_9
      node_10
         node_11
         node_12
```

```
node_13
    node_14
    node_15
```

Überlegen Sie sich eine Strategie, wie Sie die Informationen über die Einrückungen mitführen können (bspw. über Funktionsparameter oder statische Variablen).

Überladen Sie den <<-Operator für die Knotenklasse so, dass die `print`-Methode aufgerufen wird. Dadurch soll es möglich sein, einen Knoten und seine Kindelemente mittels `std::cout << node;` auszugeben. Eine Deklaration des überladenen Operators soll wieder in `Node.h` erscheinen (Achtung: Freunde nicht vergessen.). Die Implementierung kommt in `Node.cpp`.