C/C++ Programmierung
Übungsblatt 8

Florian Oetke

1 RAII und Value Categories

Erklären Sie die folgenden Begriffe, aus der letzten Vorlesung, kurz in eigenen Worten:

1. lvalue
2. rvalue
3. rvalue-Referenz
4. Destruktor
5. RAII

Beantworten Sie die folgenden Fragen in eigenen Worten:

1. Nach dem was wir in der Vorlesung zur Bedeutung der unterschiedlichen Arten von Referenzen und der Funktionsweise von Move-Semantics besprochen haben: Welche Probleme, bzw. Konflikte in der Bedeutung, könnten bei Parametern vom Typ `const std::string&&` bestehen?
2. Warum könnte es sinnvoll sein, dass eine lvalue-Referenz nicht auf rvalues verweisen kann, eine const lvalue-Referenz allerdings schon?

2 Automatisches Schreiben von XML

Die in der Vorlesung vorgestellten Special Member Functions können nicht nur zur automatischen Verwaltung von Ressourcen eingesetzt werden (was wir uns später noch im Detail anschauen werden), sondern z.B. auch dazu automatisch bestimmte Aktionen durchzuführen, wenn der entsprechende Scope bzw. Block verlassen wird.

In dieser Aufgabe sollen Sie nun eine `Xml_element` Klasse implementieren, die eine einfache XML-Ausgabe in der Konsole macht und sich dabei automatisch um die Einrückung und das Schließen der Tags kümmert. Um das zu erreichen können Sie einen Konstruktor implementieren, der den Namen des Tags als `const std::string&` Parameter bekommt, sich dieses in einer Member Variablen merkt und ein entsprechendes öffnendes XML-Tag ausgibt. Das schließende Tag können Sie dann anschließend im Destruktor ausgeben lassen, wodurch es automatisch ausgegeben wird wenn das Objekt wieder zerstört wird.

Um die Einrückung ebenfalls automatisch durchzuführen müssen Sie ihrer Klasse noch um eine statische Member Variable ergänzen, in der Sie sich die aktuelle Tiefe merken und diese an geeigneter Stelle in der Klasse inkrementieren/dekrementieren. Basierend auf dieser Variablen können Sie dann z.B. mit einer for-Schleife entsprechende Leerzeichen ausgeben.

Testen Sie Ihre Lösung mit der folgenden `main` Funktion. Die Ausgabe sollte dabei so aussehen:

```
<numbers>
    <natural>
        <0>
        </0>
        <1>
        </1>
        <2>
        </2>
    </natural>
</numbers>
```

```cpp
#include <iostream>
#include <string>

// TODO

int main() {
    auto numbers = Xml_element{"numbers"};
    auto natural = Xml_element{"natural"};

    for(auto i=0; i<3; i++) {
        Xml_element{ std::to_string(i)};
    }
}
```

3 Verkettete Liste

Wie in der Vorlesung erwähnt wurde, können Smart-Pointer nicht nur verwendet werden um polymorphe Objekte als ihre Basisklasse zu speichern, sondern auch um rekursive Datenstrukturen zu definieren.

Diese Eigenschaft sollen Sie sich nun in dieser Aufgabe zu Nutze machen, indem Sie mit ihrer Hilfe eine einfach verkettete Liste implementieren. Schreiben Sie ein Klassen-Template `Linked_list<T>` mit den folgenden Methoden:

- `push_front`: Bekommt ein `T`-Objekt als Parameter und fügt es an den Anfang der Liste hinzu.
- `pop_front`: Entfernt das erste Element in der Liste, falls es eins gibt. Wenn es keine Elemente gibt, soll statt dessen eine `std::out_of_range` Exception geworfen werden.
- `get`: Bekommt einen Index als Parameter und gibt eine Referenz auf das entsprechende Element zurück. Wenn es kein solches Element gibt, soll eine `std::out_of_range` Exception geworfen werden.
- `erase`: Bekommt einen Index als Parameter und entfernt das entsprechende Element aus der Liste. Wenn es kein solches Element gibt, soll eine `std::out_of_range` Exception geworfen werden.
- `size`: Gibt die Anzahl Elemente zurück.
- `empty`: Gibt `true` zurück, wenn die Liste aktuell keine Elemente enthält und sonst `false`.

Die Elemente sollten in einer verketteten Datenstruktur gespeichert werden, d.h. jedes Element soll auf seinen Nachfolger verweisen. Definieren Sie sich hierzu ein privates inneres `struct Entry` innerhalb ihrer `Linked_list`, das neben dem Wert des Elements (vom Typ `T`) einen `std::unique_ptr<Entry>` auf das nächste Element enthält. In ihrer `Linked_list` brauchen Sie dann nur noch eine `std::unique_ptr<Entry>` Member-Variable, die auf das erste Element in der Liste verweist (falls sie nicht leer ist). Es steht ihnen aber frei, weitere Member-Variablen zu ergänzen, wenn Sie diese für notwendig oder sinnvoll erachten.

Um Elemente einzufügen oder zu entfernen müssen Sie also lediglich ändern worauf die entsprechenden `std::unique_ptr<Entry>` für das nächste Element zeigen. Beim Einfügen am Anfang müssten Sie also z.B. einen neuen Eintrag erzeugen, der auf den bisherigen Anfang der Liste verweist und diesen anschließen als Anfang der Liste übernehmen, d.h. in ihrer Member-Variable mit dem ersten Element speichern.

Das Löschen ist nur geringfügig komplizierter. Hierbei müssen Sie den Vorgänger des Elements finden, das Sie entfernen wollen, (oder die Member-Variable, falls sie das erste Element entfernen wollen) und dessen `std::unique_ptr<Entry>` auf den Nachfolger des gelöschten Elements setzen.

Hinweise zur Verwendung von `std::unique_ptr`:

- Ein `std::unique_ptr` kann nicht kopiert werden!
  Sie müssen also ggf. `ptr_a = std::move(ptr_b);` verwenden, was den `ptr_b` auf `nullptr` setzt. Das gilt insbesondere auch, wenn sie einen `std::unique_ptr` als by-value Parameter an einen Konstruktor oder `std::make_unique` übergeben oder von einem Parameter an eine Member-Variable zuweisen wollen.
- Um einen normalen Pointer auf das Objekt zu bekommen, auf das der `std::unique_ptr` zeigt, können sie u.a. seine `get()` Methode verwenden. z.B.: `Entry* entry = uptr.get();`
- `std::make_unique` funktioniert nur mit Konstruktoren und nicht mit Aggregate-Initialisierung. Wenn Sie beim Erzeugen von `std::unique_ptr<Entry>` Objekten die Member-Variablen von `Entry` auf bestimmte Werte setzen wollen, müssen Sie also einen passenden Konstruktor definieren!

Die Ausgabe des fertigen Programms sollte so aussehen:

```
size()   : 2
empty()  : 0

get(0)   : 1
get(1)   : 3
```

Als Basis für die Implementierung können Sie folgenden Programmcode aus dem Handout verwenden:

```
#include <iostream>
#include <stdexcept>
#include <memory>
#include <vector>

int main() {
    auto list = Linked_list<int>();
    list.push_front(3);
    list.push_front(2);
    list.push_front(1);
    list.push_front(0);

    list.erase(2);
    list.pop_front();

    std::cout << "size()  : " << list.size()  << "\n";
    std::cout << "empty() : " << list.empty() << "\n\n";

    for(int i=0; i<list.size(); i++) {
        std::cout << "get("<<i<<")  : " << list.get(i)  << "\n";
    }
}
```