

Übung 3: Templates

Lernziele

- Sie repetieren den Stoff aus Vererbung, Operatoren und Templates.
- Sie lernen „meta template programming“.
- Sie entwickeln eine C++-Klasse, welche die Manipulation von Vektor-Komponenten mit den gewöhnlichen algebraischen Operatoren ermöglicht und eine Form der Lazy-Evaluation anbietet.

1 Einführung in das Thema

Ein Vektor v im dreidimensionalen Raum hat drei Komponenten (v_x, v_y, v_z) , welche die Projektion des Vektors auf die drei orthogonalen x -, y - und z -Achsen des Koordinatensystems darstellen. Man kann diese Idee verallgemeinern und sagen, dass ein Vektor r in einem n -dimensionalen Raum n Komponenten (r_1, r_2, \dots, r_n) hat.

Oft werden Vektoren in Form von eindimensionalen Arrays abgespeichert. Die Elemente eines Arrays werden dann als die Komponenten eines Vektors interpretiert. Dieser Interpretation folgend, lassen sich einige algebraische Vektoroperationen wie folgt definieren:

- Addition: $a + b = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$
- Subtraktion: $a - b = (a_1 - b_1, a_2 - b_2, \dots, a_n - b_n)$
- punktweise Multiplikation: $a * b = (a_1 b_1, a_2 b_2, \dots, a_n b_n)$
- punktweise Division: $a / b = (a_1 / b_1, a_2 / b_2, \dots, a_n / b_n)$
- Addition mit einem Skalar k von links oder rechts: $k + a = a + k = (k + a_1, k + a_2, \dots, k + a_n)$
- Subtraktion mit einem Skalar k von links: $k - a = (k - a_1, k - a_2, \dots, k - a_n)$
- Subtraktion mit einem Skalar k von rechts: $a - k = (a_1 - k, a_2 - k, \dots, a_n - k)$
- Multiplikation mit einem Skalar k von links oder rechts: $k a = a k = (ka_1, ka_2, \dots, ka_n)$
- Division mit einem Skalar k von links: $k / a = (k / a_1, k / a_2, \dots, k / a_n)$
- Division mit einem Skalar k von rechts: $a / k = (a_1 / k, a_2 / k, \dots, a_n / k)$
- Skalarprodukt: $a ** b = (a_1 b_1 + a_2 b_2, + \dots + a_n b_n)$

Diese Operationen sind nur dann definiert, wenn die Dimensionen der beiden Vektoren gleich sind. Beim Skalarprodukt ist das Resultat ein Skalar, unabhängig der Dimension der Vektoren.

1.1 Teilaufgaben

Die Übung setzt sich aus folgenden vier Teilaufgaben zusammen:

1. Vektoroperationen ohne Skalar und Skalarprodukt realisieren.
2. Zusätzlich die Operationen mit einem Skalar realisieren.
3. Summenbildung realisieren.
4. Optional das Skalarprodukt realisieren.

1.2 Klasse Vector

Für unsere Vektoren entwickeln wir eine generische Klasse Vector. Schreiben Sie den gesamten Code der Klasse in einer Datei vector.hpp. In der Gestaltung der Klasse Vector sind Sie dahingehend frei, dass Sie mindestens das Speichern von beliebig aber endlich langen Vektoren eines beliebigen numerischen Typs unterstützen müssen. In der einfachsten Variante leiten Sie Ihre Klasse von `std::array<T, S>` ab, wobei T der Elementdatentyp und S die Länge des Vektors ist. In den gezeigten Beispielen gehe ich von dieser einfachen Variante aus.

Stellen Sie sicher, dass Ihre Klasse Vector einen Standardkonstruktor besitzt, einen Konstruktor mit Initialisierungsliste als Parameter (siehe nachfolgendes Beispiel) und einen Ausgabeoperator. Später werden Sie diese Klasse noch um weitere Methoden ergänzen.

```
Vector(const std::initializer_list<T>& data) {
    size_t s = __min(data.size(), S);
    auto it = data.begin();
    for (size_t i = 0; i < s; i++) this->at(i) = *it++;
}
```

1.3 Klasse Expression

Das Herzstück dieser Übung ist die generische Klasse `Expression`. Sie ermöglicht das Erzeugen von beliebig komplexen algebraischen Ausdrücken basierend auf den einleitend genannten Vektor-Operationen und die späte Auswertung derselben (Lazy Evaluation).

Ein Auszug aus dieser Klasse könnte wie folgt ausschauen:

```
template<typename Left, typename Op, typename Right> class Expression {
    const Left& m_left;
    const Right& m_right;

public:
    using value_type = typename Left::value_type;

    Expression(const Left& l, const Right& r) : m_left{ l }, m_right{ r } {}

    size_t size() const { return m_left.size(); }

    value_type operator[](size_t i) const {
        return Op::template apply<value_type>(m_left[i], m_right[i]);
    }
};
```

Ein algebraischer Ausdruck (`Expression`) ist in diesem Fall eine binäre Operation `Op` mit den zwei Argumenten `l` und `r` von den jeweiligen Typen `Left` bzw. `Right`. Die Operation `Op` wird in Form einer eigenen Klasse pro Operation angegeben. Jeder dieser Operationsklassen muss lediglich eine statische, generische Methode `apply(...)` anbieten, welche dann beim Indexoperator zum Einsatz kommt. Für die Addition könnte diese Klasse wie folgt ausschauen:

```
struct Add {
    template<typename T> static T apply(T l, T r) {
        return l + r;
    }
};
```

Der Datentyp `T` entspricht dem Elementdatentyp des Vektors und muss die binäre Operation `+` für zwei Operanden vom Typ `T` unterstützen, damit dieser Code fehlerfrei kompiliert.

1.4 Auswertung eines Ausdrucks (Lazy-Evaluation)

Unsere Klasse `Expression` ist so aufgebaut, dass algebraische Ausdrücke erst dann ausgewertet werden, wenn dies notwendig wird. Das nennt man Lazy-Evaluation. In dieser Übung erfolgt die Auswertung beim Einsatz des Indexoperators. Dadurch ist es möglich, einen (komplexen) algebraischen Ausdruck aufzubauen und diesen dann für verschiedene Eingabedaten auszuwerten. Folgender Anwendungscode sollte möglich sein:

```
Vector<double, 4> A({ 1, 2, 3, 4 });
Vector<double, 4> B({ 2, 1, 0, 1 });
Vector<double, 4> D;
auto e = (A - B) * (A + B);
D = e;
cout << D << endl;           // [-3,3,9,15]
B = { 3, 0, 2, 5 };
cout << e << endl;           // [-8,4,5,-9]
B = { 4, 3, 3, -2 };
cout << e[1] << endl;         // -5
```

In diesem Code ist `e` ein nicht ausgewerteter Ausdruck, welcher erst bei der Zuweisung zum Vektor `D` ausgewertet wird, da dort für jedes Element des Vektors der Indexoperator des Ausdrucks aufgerufen werden muss. Bei der Ausgabe von `D` auf die Konsole kommt der Ausgabeoperator der Klasse `Vector` zum Einsatz. Wenn die Eingabedaten des Ausdrucks `e` verändert werden (im Beispiel wird der Vektor `B` mit neuen Daten überschrieben), so kann der Ausdruck `e` erneut evaluiert werden, was hier bei der Ausgabe von `e` auf die Konsole geschieht. Ein Ausdruck muss nicht immer vollständig ausgewertet werden. In der letzten Zeile des Beispiels wird der Ausdruck `e` nur an der Stelle mit Index 1 ausgewertet. Das heisst, es wird nur der Wert -5 berechnet und ausgegeben, die anderen Werte des Ausdrucks werden erst gar nicht ausgerechnet.

Damit der vorangegangene Code ausführbar wird, müssen Sie folgende Teilprobleme lösen:

- Operationen Subtraktion, Multiplikation und Division analog zur Klasse Add realisieren;
- Klasse Vector: Zuweisungsoperator implementieren, so dass eine beliebige Expression einem Vektor zugewiesen werden kann;
- Klasse Expression: Ausgabeoperator analog zum Ausgabeoperator der Klasse Vector implementieren, wobei für jedes Element der Indexoperator aufgerufen werden soll;
- Klasse Expression: Operatoren +, -, * und / überladen, so dass sowohl der linke wie auch der rechte Operand ein Vektor oder ein Ausdruck sein dürfen. Der nachfolgende Code gibt einen Eindruck, wie das ausschauen könnte.

```
template<typename Left, typename Right>
Expression<Left, Add, Right> operator+(const Left& l, const Right& r) {
    return Expression<Left, Add, Right>(l, r);
}
```

1.5 Tests

Die Datei „unittest1.cpp“ enthält eine Vielzahl von Tests, mit der Sie Ihre Implementierung überprüfen können. Die Tests sind gemäss den drei Aufgaben unterteilt. Damit Sie die beiden Tests zur Aufgabe 1 ausführen können, sollten Sie sowohl in der Klasse Vector als auch in der Klasse Expression einen Gleichheitsoperator als generische Instanzmethode implementieren. Diese Gleichheitsoperatoren müssen die Vektoren bzw. Ausdrücke komponentenweise auf Gleichheit überprüfen. Testen Sie Ihren Code auch in einer Konsolenapplikation. Achten Sie darauf, dass nur Ihre Datei vector.hpp inkludiert werden muss.

2 Operationen mit nur einem Skalar

Packen Sie diese Aufgabe erst an, wenn Sie die Aufgabe 1 vollständig gelöst haben und die mitgelieferten Unit-Tests zur Aufgabe 1 korrekt ausgeführt werden können.

Nun wollen wir unsere algebraischen Ausdrücke dahingehend erweitern, dass nur einer der beiden Operanden ein Skalar (numerischer Wert) sein darf, während der andere Operand nach wie vor ein Vektor oder ein Ausdruck sein soll. Folgendes Beispiel zeigt, was möglich sein soll:

```
auto e = (2.0*(A - B)/2.0 + B + 5.0) * (A - 4.0 + 4.0*B)/4.0;
```

Wie diese Skalar-Operationen zu verstehen sind, ist einleitend beschrieben worden.

Stellen Sie sich vor, dass Sie einen Ausdruck haben, deren einer Operand ein Vektor und der andere ein Skalar ist. Wird auf diesem Ausdruck der Index-Operator der Klasse Expression angewandt, so kommt es zu einem Kompilationsfehler, weil der Index-Operator der Klasse Expression für jede Komponente des Vektors den Index-Operator des Vektors aufruft, dies aber nicht für den Skalar tun kann, da ein Skalar keinen Indexoperator hat. Diese Schwierigkeit können Sie beispielsweise durch zwei Spezialisierungen der generischen Klasse Expression lösen, indem diese Spezialisierungen den Index-Operator derart neu implementieren, dass nur beim einen Operand, welcher ein Ausdruck oder ein Vektor ist, der Index-Operator für den Zugriff auf die Komponenten ausgeführt wird und beim skalaren Operand immer der gleiche skalare Wert eingesetzt wird. Die dazu notwendigen Erweiterungen betreffen ausschliesslich die Klasse Expression.

Beachten Sie, dass in einer Spezialisierung einer generischen Klasse alle Instanzmethoden und Konstruktoren der allgemeinen Version der Klasse nochmals re-implementiert werden müssen. Daher bietet es sich an, die gemeinsamen binären Operatoren ausserhalb der beiden Klassen als globale Funktionen zu realisieren. Testen Sie Ihre Erweiterung wiederum mit den mitgelieferten Unit-Tests.

3 Summenbildung

Packen Sie diese Aufgabe erst an, wenn Sie die Aufgabe 2 vollständig gelöst haben und die mitgelieferten Unit-Tests zur Aufgabe 2 korrekt ausgeführt werden können.

Implementieren Sie für beide Klassen Vector und Expression eine globale Funktion sum(...), welche die Werte in einem vorgegebenen Index-Bereich zusammenaddiert und zurückgibt. Der erste Parameter von sum soll entweder ein Vector oder eine Expression sein. Mit dem zweiten und dritten Parameter wird der Wertebereich [from, to) des Index festgelegt. Die beiden Parameter für den Wertebereich sollen optional sein. Falls from nicht angegeben wird, soll beim Index 0 gestartet werden. Falls to nicht angegeben wird, soll bis zum letzten Element iteriert werden.

```
double d = sum(2.0*A, 1, 3);
double e = sum(2.0*A*A*3.0, 2);
```

Achten Sie bei der Umsetzung darauf, dass Sie möglichst wenig Code schreiben.

4 Skalarprodukt (Optional)

Packen Sie diese Aufgabe erst an, wenn Sie die Aufgabe 3 vollständig gelöst haben und die mitgelieferten Unit-Tests zur Aufgabe 3 korrekt ausgeführt werden können.

Während in Aufgabe 2 die Resultate der Operationen nach wie vor Vektoren sind, wollen wir jetzt auch das Skalarprodukt ermöglichen, welches aus zwei Vektoren bzw. Ausdrücken einen Skalar berechnet. Selbstverständlich soll das Ergebnis eines Skalarprodukts auch als Skalar in einer Operation aus Aufgabe 2 verwendet werden können. Folgende Ausdrücke sollen beispielsweise möglich sein:

```
double d = ((A - B)*(B**A))**B;  
auto e = (A**B)*A - B*(B**A);
```

Beachten Sie, dass als Operator für das Skalarprodukt `**` verwendet wird. Dies ist kein üblicher C++-Operator. Überlegen Sie sich, wie Sie diese Syntax ohne Präprozessor realisieren können.

Testen Sie Ihre Erweiterung wiederum mit den mitgelieferten Unit-Tests.