# 1 Testat-Übung: Expression

In dieser Übung sollen Sie schrittweise eine eigene `Expression`-Klasse schreiben, welche soweit erweitert wird, damit wir diverse algebraische Audrücke ohne explizite Funktionsaufrufe schreiben können. Dazu nutzen wir Überladung, Operatoren und Template Meta Programming.

## 1.1 Vorgehen

Das entsprechende Projekt ist bereits vorbereitet. Ebenfalls finden Sie die vorbereiteten leeren Header für ihre Implementation: `vector_operations.h`, `element_operations.h`, `expression.h` und `expression_operations.h`. Sie sollen die entsprechend geforderte Funktionalität in diesen Files implementieren.

Beachten Sie, dass wir hier nur Template Klassen implementieren und deshalb alles direkt im Header implementieren müssen. Vergessen Sie auch das wichtige `#pragma once` am Anfang jedes Headers nicht.

Die dazu passenden Unit-Tests sind bereits implementiert und erlauben ihnen, ihre Implementation zu testen. Zudem existiert auch eine kleine Applikation, welche Sie zum Experimentieren verwenden können. Oder Sie schreiben zusätzliche Unit-Tests. Das verwendete Test-Framework Catch2 haben Sie bereits kennengelernt.

Öffnen Sie das `workspace`-Repo in CLion als Root-Projekt und navigieren Sie zu `uebungen/expression`. Hier finden sie das vorbereitete Projekt. Aktivieren Sie es indem Sie in `uebungen/CMakeLists.txt` die folgende Zeile aktivieren:

```
1 add_subdirectory(expression)
```

Die Aufgaben sind Schritt für Schritt abzuarbeiten. Bis auf den mit *optional* markierten Teil sind alle zwingend korrekt zu implementieren. Aktivieren Sie die einzelnen Tests/Signaturen, indem Sie die entsprechenden `#defines` in `element_operations.h` und `expression.h` aktivieren.

## 1.2 Testat

Geben Sie die geforderten Implementationen bis zum vereinbarten Datum ab. Senden Sie die Dateien `vector_operations.h`, `element_operations.h`, `expression.h` und `expression_operations.h` per Mail an christian.lang@fhnw.ch. Die Lösung muss einerseits alle Unit-Tests, anderseits auch bestimmte Qualitäts-Merkmale erfüllen. Diese Qualitätsmerkmale werden automatisch durch cpplint und manuell durch den Dozenten geprüft. Sie erhalten dementsprechend Feedback zu ihrer Abgabe.

### 1.3 Ziele

- Sie repetieren den Stoff aus Vererbung, Operatoren und Templates.
- Sie lernen *Template Meta Programming*.
- Sie entwickeln C++-Klassen, welche die Manipulation von Vektor-Komponenten mit den gewöhnlichen algebraischen Operatoren ermöglicht und eine Form der Lazy Evaluation anbietet.

## 2 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 $(r_1, r_2, \ldots, r_n)$ 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` $= (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n)$
- Subtraktion: `a - b` $= (a_1 - b_1, a_2 - b_2, \ldots, a_n - b_n)$
- punktweise Multiplikation: `a * b` $= (a_1 b_1, a_2 b_2, \ldots, a_n b_n)$
- punktweise Division: `a / b` $= (a_1/b_1, a_2/b_2, \ldots, a_n/b_n)$
- Addition mit einem Skalar $k$ von links oder rechts: `k + a` $=$ `a + k` $= (k + a_1, k + a_2, \ldots, k + a_n)$
- Subtraktion mit einem Skalar $k$ von links: `k - a` $= (k - a_1, k - a_2, \ldots, k - a_n)$
- Subtraktion mit einem Skalar $k$ von rechts: `a - k` $= (a_1 - k, a_2 - k, \ldots, a_n - k)$
- Multiplikation mit einem Skalar $k$ von links oder rechts: `k * a` $=$ `a * k` $= (ka_1, ka_2, \ldots, ka_n)$
- Division mit einem Skalar $k$ von links: `k / a` $= (k/a_1, k/a_2, \ldots, k/a_n)$
- Division mit einem Skalar $k$ von rechts: `a / k` $= (a_1/k, a_2/k, \ldots, a_n/k)$
- Skalarprodukt: `a ** b` $= (a_1 b_1 + a_2 b_2, + \cdots + a_n b_n)$

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

## 3 Basis

Im ersten Teil dieser Aufgabe erweitern wir die Klasse `std::vector` und erstellen die später benötigten Operations-Klassen.

### 3.1 Verwendung von `std::vector`

Für die Speicherung der Vektor-Daten benötigen wir eine Klasse, welche eine beliebige Anzahl Elemente eines bestimmten Datentyps speichern kann. Zudem möchten wir die Daten einfach auf die Konsole ausgeben können und später weitere Operatoren zur Klassen hinzufügen.

Dazu könnte man nun einen neue Klasse definieren, welche intern z.B. `std::vector` als Datenspeicher verwendet. Viel sinnvoller ist es allerdings, einfach `std::vector` direkt zu verwenden. Zusätzliche Operatoren können als freie Funktionen, ausserhalb der Klasse, definiert werden.

### 3.2 Zusätzliche Operatoren für `std::vector`

Diese implementieren Sie in der Datei `vector_operations.h`. Da sich die `vector`-Klasse im Namensraum `std` befindet, sollten Sie die Erweiterungen ebenfalls in diesem definieren. In diesem ersten Teil implementieren Sie erst den `operator<<` um Instanzen von `std::vector` direkt auf die Konsole ausgeben zu können. Denken Sie daran, dass dies eine Template-Funktion sein muss und dementsprechend direkt im Header implementiert wird.

### 3.3 Element-Operations-Klassen

Um später algebraische Ausdrücke zu schreiben, welche aus jeweils zwei Datenbehältern (z.B. `std::vector`-Instanzen) und einer Operation bestehen, benötigen wir öffentliche Klassen, welche diese Operationen einerseits als spezifische Klasse verkörpern aber auch die entsprechende Funktionalität liefern können.

Schreiben Sie dazu in `element_operations.h` die Klassen für die Operationen Addieren, Subtrahieren, Multiplizieren und Dividieren. Als Beispiel folgend die Implementation der Operation Addieren. Danach können Sie zum ersten Mal die Unit-Tests laufen lassen.

```
1 struct Add {
2   template<typename T>
3   static T Apply(T lhs, T rhs) {
4     return lhs + rhs;
5   }
6 };
```

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.

# 4 Algebraische Ausdrücke

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

## 4.1 Klasse `Expression`

Ein Auszug aus dieser Klasse in `expression.h` könnte wie folgt aussehen:

```
1  template<typename Left, typename Op, typename Right>
2  class Expression {
3   public:
4    using value_type = typename Left::value_type;
5    Expression(const Left& lhs, const Right& rhs) : lhs_{lhs}, rhs_{rhs} {
6      assert(lhs_.size() == rhs_.size() || lhs_.size() == 1 || rhs_.size() == 1);
7    }
8
9    size_t size() const {
10     return std::max(lhs_.size(), rhs_.size());
11   }
12   value_type operator[](size_t i) const {
13     return Op::Apply(lhs_[i], rhs_[i]);
14   }
15
16  private:
17   const Left lhs_;
18   const Right rhs_;
```

Ein algebraischer Ausdruck (engl. *Expression*) ist in diesem Fall eine binäre Operation `Op` mit den zwei Argumenten `lhs` und `rhs` von den jeweiligen Typen `Left` bzw. `Right`. Die Operation `Op` ist nur als Typ vorhanden und nicht als Instanz. Dies ist möglich, weil wir die Funktion `Apply` als **`static`** markiert haben und dementsprechend als Klassenfunktion aufrufbar ist.

Die Klasse besteht hauptsächlich aus einem entsprechenden Konstruktor, welche unveränderbare Kopien von `lhs` und `rhs` speichert. Im Weiteren bietet sie den Index-Operator an, welcher beim Aufruf genau nur das angefragte Element der Expression auswertet und zurückgibt. Zudem bietet sie die Methode `size()` an. Hier ist wichtig, dass der Methoden-Namen genau gleich geschrieben ist wie im `std::vector`, da `Left`/`Right` jeweils ein `std::vector` oder eine `Expression` sein kann.

## 4.2 Auswerten einer `Expression`

Unsere Klasse `Expression` ist so aufgebaut, dass algebraische Ausdrücke erst dann ausgewertet werden, wenn dies vom Benutzer erwünscht wird. Die Auswertung erfolgt beim Aufruf des Index-Operators. Dadurch ist es möglich, einen (komplexen) algebraischen Ausdruck aufzubauen und diesen dann für verschiedene Eingabedaten auszuwerten. Folgender Anwendungscode soll möglich sein:

```
1     std::vector a = {1, 2, 3, 4};
2     std::vector b = {2, 1, 0, 1};
3     const auto e = (a - b) * (a + b);
4     std::cout << e << std::endl;        // [-3,3,9,15]
5     std::cout << e[0] << std::endl;     // -3
```

In diesem Code ist `e` ein nicht ausgewerteter Ausdruck, welcher erst beim Ausgeben auf der Konsole ausgewertet wird, da dort für jedes Element des Vektors der Index-Operator des Ausdrucks aufgerufen werden muss.

Ein Ausdruck muss nicht immer vollständig ausgewertet werden. In der letzten Zeile des Beispiels wird der Ausdruck `e` nur an der Stelle mit Index $0$ ausgewertet. Das heisst, es wird nur der Wert $-3$ berechnet und ausgegeben, die anderen Werte des Ausdrucks werden erst gar nicht ausgerechnet.

Damit solche Ausdrücke geschrieben werden können, benötigen wir noch weitere Operatoren, welche Sie alle in der Datei `expression_operations.h` implementieren:

- Operatoren für $+$, $-$, $*$ und $/$ überladen, so dass sowohl der linke wie auch der rechte Operand ein `std::vector` oder eine `Expression` sein dürfen. Der nachfolgende Code gibt einen Eindruck, wie das ausschauen könnte.

  ```
  1  template<typename Left, typename Right>
  2  auto operator+(const Left& lhs, const Right& rhs) {
  3    return Expression<Left, Add, Right>(lhs, rhs);
  4  }
  ```

  Diese Operatoren sorgen dafür, dass aus einem Ausdruck wie

  ```
  1  const auto e = a + b;
  ```

  eine `Expression`-Instanz vom Typ

  ```
  1  Expression<std::vector<int>, Add, std::vector<int>
  ```

  erzeugt wird.

- Für die Unit-Tests benötigen wir noch Vergleichs-Operatoren. Dieser muss so überladen werden, dass er einerseits mit zwei `Expression`s, andererseits mit einer `Expression` und einem `std::vector` funktioniert. Er soll nicht für Vergleiche von zwei `std::vector` oder mit beliebigen

primitiven Typen verwenden werden können.

- Für Debugging und Ausdrucke auf die Konsole sollen Sie auch hier einen Stream-Operator implementieren.

Testen Sie ihre Implementation mittels definieren von `#define EXPRESSION_BASE` in `expression.h`.

## 5 Operationen mit einem Skalar

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

```
1 const auto e = (2 * (a - b) / 2 + b + 5) * (a - 4 + 4 * b) / 4;
```

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 Index-Operator hat.

Diese Schwierigkeit können Sie durch zwei Spezialisierungen der generischen Klasse `Expression` lösen. Diese Spezialisierungen implementieren den Index-Operator neu, 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`.

Verwenden Sie hier nun den öffentlichen Typ `Expression::value_type` , welcher ihnen erlaubt, denn Element-Typ einer `Expression` zu erhalten. In der einen Spezialiserung von `Expression` welche einen Skalar von Links nimmt, verwenden Sie nun nicht mehr den Template-Parameter `Left` sondern denn Element-Typ, welcher durch den Typ `Right` implizit definiert ist. Damit dies funktioniert ist auch hier wichtig, dass `value_type` genau gleich geschrieben ist wie in `std::vector`.

Beachten Sie, dass in einer Spezialisierung einer generischen Klasse alle Instanzmethoden und Konstruktoren der allgemeinen Version der Klasse nochmals re-implementiert werden müssen.

Testen Sie ihre Implementation mittels definieren von `#define SCALAR_FROM_RIGHT` und `#define SCALAR_FROM_LEFT` in `expression.h`.

# 6 Skalarprodukt

Während die Resultate der bisher implementierten 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 der anderen Operationen verwendet werden können. Folgende Ausdrücke sollen beispielsweise möglich sein:

```
1 double d = ((a - b) * (b**a)) ** b;
2 auto e = (a**b) * a - b * (b**a);
```

## 6.1 `Dot`-Operation

Damit das Skalarprodukt nun eine `Expression` mit der entsprechenden Operation für das Skalarprodukt erstellen kann, benötigen Sie noch eine neue Klasse in `element_operations.h` Diese soll nun eine einzelnee Klassenfunktion `DotApply` implementieren, welche nicht einzelne Elemente vom Typ `T` verrechnet, sondern direkt das Skalarprodukt von `Left` und `Right` evaluiert.

Diese `Dot` Klasse testen Sie mittels definieren von `#define ELEMENT_OPERATIONS_DOT` in `element_operations.h`.

## 6.2 `**`-Operator emulieren

Beachten Sie, dass als Operator für das Skalarprodukt `**` verwendet wird. Dies ist kein erlaubter C++-Operator. Damit dies möglich ist, benötigen wir einen kleinen Trick. Und zwar überladen wir den Dereferenzierungs-Operator von `Expression` so, dass er einen Pointer auf die `Expression` zurückgibt. Das gleiche benötigen wir auch für `std::vector`. Implementieren Sie diesen Dereferenzierungs-Operator in `vector_operations.h` und `expression_operations.h`

Nun können wir den Multiplikations-Operator für `Expression` so überladen, dass er als `rhs` keine Referenz einer `Expression` sondern einen Pointer erwartet. Diesen Operator implementieren Sie in `expression_operations.h`

## 6.3 Resultat als Skalar

Als letztes benötigen wir noch einen Weg um eine `Expression`, welche zu einem Skalar evaluiert wird, auch wirklich als primitiven Typ zu interpretieren. Dazu verwenden wir den Casting-Operator, welcher uns erlaubt, eine `Expression` in einen anderen Typ zu konvertieren. In diesem Fall ist es immer der

Element-Typ der durch `Expression::value_type` jederzeit verfügbar ist. Im Basis-Fall sieht dieser Operator z.B. folgendermassen aus:

```
1  operator value_type() const {
2    return Op::Apply(static_cast<value_type>(lhs_), static_cast<value_type>(
       rhs_));
3  }
```

Diesen Operator benötigen Sie in jeder Spezialisierung der Klasse `Expression`, weil Sie transitiv verwendet wird. Z.B. in einem Ausdruck `auto e = (a**b)* 2;` wird zuerst das Skalarprodukt von $a$ und $b$ gerechnet, welches einen skalaren Wert zurückgibt. Hier wird also der Cast-Operator der Spezialisierung für `Dot` benötigt. Danach wird aber erneut einen Expression erstellt, welche als `Left` diese `Dot`-Expression hat, als `lhs` aber einen Skalar. Somit brauchen wir hier den Cast-Operator auch für die Spezialisierung von `Expression` mit einem Skalar von Rechts.

Beachten Sie, dass `clang-tidy` (in CLion integriert) hier vorschlägt, diesen Casting-Operator `explicit` zu markieren. Dies ist in diesem Fall aber kontraproduktiv, da wir genau die implizite Anwendung dieses Operators beabsichtigen.

Zudem benötigen Sie eine komplett neue Spezialisierung von `Expression`, welche zwar immer noch mit `Left` und `Right` flexibel bezüglich ihrer Operanden ist, aber die Operation fix auf `Dot` spezialisiert hat. Diese Spezialisierung implementiert nun den Index- und Cast-Operator so, dass die Operation `Dot::DotApply` verwendet wird und somit nur ein einzelner skalarer Wert zurückgegeben wird.

Testen Sie ihre Implementation mittels definieren von `#define DOT_PRODUCT` in `expression.h`.

## 7 Spezialfall Index-Operator beim Skalarprodukt *(optional)*

Eventuell haben Sie bemerkt, dass der Parameter `i` im Index-Operator der `Dot`-Spezialisierung gar nicht verwendet wird. Trotzdem wird der Operator nicht nur mit $i = 0$ aufgerufen, sondern mit jedem Index, der entsprechenden Vektoren der verknüpften `Expression`s. Dies führt dazu, dass die Skalarprodukt-Operation $n$-mal ausgeführt wird, obwohl sie jedesmal das gleiche Resultat berechnet.

Diese Verschwendung von Performance sollen Sie nun optimieren, indem Sie eine neue Methode `DotApplyCached()` in der `Dot`-Spezialisierung implementieren. Diese Methode verwenden Sie dann einerseits im Index- aber auch im Cast-Operator. Die Methode soll sicherstellen, dass das Skalar-Produkt nur einmal gerechnet und danach nur noch das Resultat zurückgegeben wird.

Um dies korrekt zu implementieren, werden Sie das Keyword `mutable` benötigen, damit Sie die **`const`**-Korrektheit von `DotApplyCached` nicht zerstören müssen. Ein guter Weg um einen Cache zu implementieren, wäre zudem die Klasse `std::optional`.

Diese Erweiterung können Sie mit den bestehenden Unit-Tests testen. Funktional sollte nichts geändert haben.