13_vmath: Vektoren

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

2 Aufgabenstellung

In diesem Beispiel wollen wir eine header-only Bibliothek erstellen, die eine Klasse für Vektoren im dreidimensionalen Raum enthält.

Auf Unit-Tests wollen wir aus Zeitgründen wiederum verzichten.

3 Anleitung

1. Schreibe eine Headerdatei `vmath.h`, die eine Klasse `Vector3D` enthält, die über einen Defaultkonstruktor (nicht selber entwickeln!), einen Konstruktor mit 3 `double` Parametern, sowie 3 öffentliche Attribute für die 3 Koordinaten enthält. Weiters wollen wir so einen Vektor wieder auf `stdout` ausgeben können.

   Also nichts besonderes.

   Teste "händisch" in `main()`.

2. Von der Klasse sollte nicht mehr abgeleitet werden können... `final` als Spezifizierer der Klasse hilft weiter.

3. Soweit sogut, aber wer sagt, dass wir immer Vektoren mit `double` Werten haben wollen? Gerade wenn wir mit GPUs arbeiten, sind `float` Werte besser, da diese für `floats` optimiert sind. Noch eine solche Klasse schreiben ist auch keine Lösung, also lassen wir den Compiler die Arbeit erledigen: Stelle die Klasse auf Template um. Besser jetzt als später, da wir dann weniger umschreiben müssen!

   Beachte, dass im Template der Typ `Vector3D<T>` ist, aber die Konstruktoren,... immer nur als `Vector3D` geschrieben werden!

4. Was meinst du:

   - Müssen wir einen copy-constructor definieren?
   - Müssen wir einen copy-assignment-operator definieren?
   - Müssen wir einen move-constructor definieren?
   - Müssen wir einen move-assignment-operator definieren?

   Denken!

5. Implementiere jetzt die benötigten Konstruktoren.
6. Jetzt geht es weiter mit den arithmetischen Operatoren. Beginnen wir mit den einwertigen (unary) Operatoren:

   a) Probiere testhalber aus:

   ```
   Vector3D<double> v{1, 1, 1};
   cout << +v << endl;
   ```

   Du wirst einen Compilerfehler erhalten (eh klar), aber eigentlich erwarten wir uns, dass dies funktionieren würde, da wir es von der Mathematik so gewohnt sind, nicht wahr?

   Implementiere daher die beiden Methoden `operator+` und `operator-`. Überlege dir welchen Rückgabetyp diese Methoden haben sollten! `operator-` sollte klar sein, aber wie sieht es mit `operator+` aus?

   b) Als nächstes wären die einwertigen Operatoren `++` und `--` dran. Diese gibt es allerdings sowohl als Präfix- als auch als Postfixoperatoren. Hier sind die entsprechenden Prototypen (auch alles Methoden):

   ```
   Vector3D<T>& operator++();  // prefix
   Vector3D<T> operator++(int);  // postfix
   Vector3D<T>& operator--();  // prefix
   Vector3D<T> operator--(int);  // postfix
   ```

   Der Parameter vom Typ `int` dient nur zur Markierung, dass es sich um die Postfixvariante handelt. Dieser Parameter wird nicht verwendet!!! Die Postfixvariante ist ein bisschen schwieriger, da der alte Wert als Kopie zurückgeliefert werden muss. Dafür kann die Postfixvariante auf die Präfixvariante zurückgreifen. DRY und so halt. Na ja, auch nicht wirklich schwieriger.

   c) Weiter mit den Operatoren `+=`, `-=` und `*=`, die wiederum Methoden sein sollten (eh klar). Addition und Subtraktion von Vektoren sollte kein Problem sein und bei der Multiplikation denken wir an das Kreuzprodukt:

   $$\vec{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}, \vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, \vec{u} \times \vec{v} = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix}$$

   Prototypen gefällig?

   ```
   Vector3D<T>& operator+=(const Vector3D<T>& op2);
   Vector3D<T>& operator-=(const Vector3D<T>& op2);
   Vector3D<T>& operator*=(const Vector3D<T>& op2);
   ```

   Falls dir im Moment nicht bewusst ist, was das Kreuzprodukt wirklich ist, dann schaue in deinen Mathe-Unterlagen nach!

   d) So, jetzt geht es weiter zu den binären arithmetischen Operatoren, die *nicht* Methoden sondern freie Funktionen sein sollten:

   ```
   Vector3D<T> operator+(const Vector3D<T>&, const Vector3D<T>&);
   Vector3D<T> operator-(const Vector3D<T>&, const Vector3D<T>&);
   Vector3D<T> operator*(const Vector3D<T>&, const Vector3D<T>&);
   ```

   Hier kannst du direkt auf die Implementierung der Operatoren `+=`, `-=` und `*=` zurückgreifen!

   e) Abgesehen von dem Kreuzprodukt gibt es noch zwei weitere "Multiplikationsoperationen":

- Vektor multipliziert mit einem Skalar. Dazu gibt es eigentlich bzgl. der Mathematik nichts mehr zu sagen.

  Teste folgendermaßen:

  ```
  Vector3D<double> v{1, 1, 1};
  cout << v * 3. << endl;
  cout << 3. * v << endl;
  // cout << 3 * v << endl; // gibt compile error!!!
  ```

- Skalarprodukt oder auch inneres Produkt genannt. Dafür wird in der Mathematik oft der Punkt · verwendet. Diesen Operator haben wir beim Programmieren nicht. Es gibt entweder die Möglichkeit auf eine Funktion `dot` mit geeigneten Parametern zurückzugreifen oder einen anderen Operator zu verwenden. Ich denke hier können wir den Operator % durchaus zweckentfremden. Los geht's!

f) Jetzt fehlt eigentlich nicht mehr viel an Funktionalität. Im Moment fällt mir nur mehr der Betrag eines Vektors ein. Ich denke hier wäre es durchaus sinnvoll eine Funktion `abs()` zu schreiben, die genau dies tut.

Damit sind die Rechenoperationen abgeschlossen.

g) Was bedeutet es, dass der Betrag eines Vektors genommen wird, den wir bis jetzt mit der Funktion `abs()` berechnet haben. Wir ermitteln einen Wert vom Typ `T`, der bei uns bis jetzt immer `double` gewesen ist, aber auch z.B. `float` oder `int` sein könnte.

Wir könnten übungshalber einen Konversionsoperator implementieren, damit wir dann den Betrag auch auf folgende Art bekommen könnten:

```
cout << (double)v << endl;
```

aber auch

```
cout << static_cast<double>(v) << endl;
```

wird dann funktionieren.

Dazu musst du eine Methode mit folgendem Prototypen implementieren:

```
operator T() const;
```

Beachte, dass kein Rückgabetyp angegeben ist. Die Implementierung ist trivial...

h) Aber teste einmal folgenden Code:

```
cout << v1 + 4 << endl;
```

Was passiert hier?

Ist das beabsichtigt? Sicher nicht!

Das kann dadurch umgangen werden, indem der gerade implementierte Konvertierungsoperator mit `explicit` markiert wird, sodass dieser nicht mehr zu einer impliziten Konvertierung herangezogen wird.

Also:

```
explicit operator T() const;
```

Damit bekommen wir wieder einen Compilerfehler und sind zufrieden.

i) Weiter geht es mit den Vergleichsoperatoren, denn wir können derzeit nicht einmal zwei Vektoren auf Gleichheit untersuchen. Wie sieht es wieder mit < und dergleichen aus? Wir halten uns wiederum an die lexikographische Ordnung wie wir es schon von Tupeln in Python gewohnt sind.

   Implementiere alle Vergleichsoperatoren, aber man kann und soll durchaus den Operator != mit Hilfe des Operators == implementieren, usw.

   Der Prototyp für == sieht folgendermaßen aus:

```
bool operator==(const Vector3D<T>&, const Vector3D<T>&);
```

4 Übungszweck dieses Beispiels

- Vektorarithmetik
- `final` bei Klassen
- Üben einfacher Templates
- Implementieren von Operatoren
  - unär: +, -
  - Präfix und Postfix ++, --
  - +=, -=, *=,...
  - binär: +, -, *, %,...
  - Konversionsoperatoren implementieren
  - `explicit`
  - Vergleichsoperatoren