Übungsblatt 7:
Classes, Templates & Operator Overloading

Sommersemester 2016

Programmiertechniken

Prof. Dr. L. Pollet

J. Greitemann, D. Hügel, J. Nespolo, T. Pfeffer

---

1) Konstruktion von Vektor-Objekten

Sie haben bereits verschiedene Container-Datentypen kennengelernt (z.B. `std::vector`, `std::array` und `std::valarray`), die z.T. auch Operatoren überladen. In der Vorlesung haben Sie zudem den Container `std::vector` nachprogrammiert. Diese Container haben allesamt eine dynamische Größe, d.h. ihr Speicherbedarf steht zur Compile-Zeit noch nicht fest (und kann sich teilweise zur Laufzeit ändern). Daher allozieren diese Container ihren Speicher auf dem Heap, womit jedoch ein gewisser Overhead verbunden ist, der sich für große Datenmengen jedoch schnell relativiert. *Heap allocation* bietet zudem eine größere Flexibilität, da es ggf. genügt lediglich den Besitzanspruch des Speichers zu übertragen und somit Kopieroperationen zu sparen (Stichwort: Move semantics).

In physikalischen Anwendungen brauchen wir jedoch häufig zwei- oder dreielementige Vektoren. In diesem Fall steht der Speicherbedarf zur Compile-Zeit fest, sodass der notwendige Speicher auf dem Stack alloziert werden kann. Zum Teil lässt sich damit ein signifikanter Geschwindigkeitsvorteil gegenüber `std::valarray` erreichen.

(a) Definieren Sie eine Klasse `vec`, abhängig von zwei Template-Parametern: der Dimension `N` des Vektors und dem Datentyp `T` der gespeicherten Elemente. Definieren Sie das Datenarray mit passendem Typ als private Member-Variable.

(b) Definieren und implementieren Sie eine Reihe von Konstruktoren:

- einen *default constructor*, der den Nullvektor initialisiert;
- einen *copy constructor*;
- einen *fill constructor*, der alle Elemente mit dem gleichen Wert initialisiert;
- einen *array constructor*, der Elemente von einem Pointer beginnend einliest.

Welche dieser Konstruktoren müssen Sie tatsächlich selbst definieren, welche werden vom Compiler implizit definiert?

Implementieren Sie zusätzlich einen Konstruktor, der eine `std::initializer_list` verarbeitet. Damit können Sie ihren Vektor mit geschweiften Klammern konstruieren, z.B.

```
vec<3, int> v = { 1, 2, 3};
```

Verwenden Sie die C++ Referenz, um sich über die Handhabung zu informieren. Beachten Sie, dass es sich hierbei um ein Feature des C++11 Standard handelt – Sie müssen beim Kompilieren zusätzlich die Flag `-std=c++11` angeben.

(c) Überladen Sie den `<<` Operator, sodass Sie Vektoren ausgeben können (s.u.).

(d) Testen Sie ihren bisherigen Code. Das folgende Programm sollte den untenstehenden Output produzieren:

```
#include <iostream>
#include ”vec.hpp”

int main () {
   vec<3,double> zero;
   vec<3,double> ones(1);
```

```
    char *string = "hello";
    vec<5,char> u(string);
    vec<3,double> v = { 1, 2, 3};
    vec<3,double> w(v);
    std::cout << zero << std::endl
              << ones << std::endl
              << u << std::endl
              << v << std::endl
              << w << std::endl;
    return 0;
}
```

```
$ g++ -std=c++11 -o vec main.cpp
$ ./vec
(0, 0, 0)
(1, 1, 1)
(h, e, l, l, o)
(1, 2, 3)
(1, 2, 3)
```

2) Vektor-Operationen

Im nächsten Schritt implementieren Sie die wichtigsten Operationen, die ein Vektor unterstützen sollte. Nehmen Sie hierbei an, dass `T` eine reelle Zahl repräsentiert. Überlegen Sie sich welche der Operationen Sie sinnigerweise als member functions der Klasse und welche als alleinstehende non-member functions implementieren wollen. Implementieren Sie:

(a) den Elementzugriff mit dem Index-Operator, z.B. `v[i]`;

(b) die unären Vorzeichenoperatoren, `+v`, `-v`;

(c) Gleichheits- und Ungleichheitszeichen, `v == w`, `v != w`;

(d) Addition und Subtraktion, `v + w`, sowie `v += w`;

(e) Multiplikation mit einem Skalar, `s * v`, `v * s`, `v *= s`, `v /= s`;

(f) das Skalarproduct, `v * w`;

(g) das Kreuzprodukt, `cross(v, w)`. Beachten Sie hierbei, dass dies nur für `N` = 3 definiert ist. Verwenden Sie eine teilweise Template-Spezialisierung um die Definition auf diesen Fall einzuschänken.

Beachten Sie, dass Sie aus einer non-member function auf das private Datenarray nur direkt zugreifen können, wenn Sie die entsprechende Funktion als `friend` deklariert haben. Alternativ können Sie indirekt den Index-Operator verwenden.

Ihr Programm sollte nun in der Lage sein, das um den nachfolgenden Codeabschnitt erweiterte Beispiel auszuführen:

```
v *= 2.;
w += ones;
std::cout << v << " * " << w << " = " << (v*w) << std::endl;
vec<3,double> x = cross(v, w);
std::cout << x << " * " << v << " = " << x*v << std::endl;
```

```
...
(2, 4, 6) * (2, 3, 4) = 40
(-2, 4, -2) * (2, 4, 6) = 0
```

3) Explizite und implizite Konvertierung

Die oben implementierten Operationen funktionieren hervorragend, wenn Vektoren des gleichen Datentyps verarbeitet werden. Versuchen Sie jedoch einen `vec<3,double>` zu einem `vec<3,int>` zu addieren, erhalten Sie einen Fehler.

a) Implementieren Sie explizit einen *copy constructor* und *copy assignment operator*, falls Sie dies noch nicht getan haben. Ersetzen Sie den Datentyp `T` des Argument-Vektors durch `T2` und ergänzen Sie eine entsprechende `template` Direktive.

b) Dieser verallgemeinerte copy constructor bzw. copy assignment operator agieren jetzt zusätzlich als Konvertierungsoperatoren. Bilden Sie das Skalarprodukt eines `vec<3,double>` und eines `vec<3,int>`, indem Sie explizit letzteren in ersten konvertieren.

c) Schreiben Sie in ihrem Beispielprogramm eine Funktion

```
vec<2,double> rotate(vec<2,double> v, double phi);
```

die einen 2D-Vektor in der Ebene um den Winkel `phi` dreht. Sie haben weiter den Einheitsvektor in x-Richtung gegeben:

```
vec<2,int> e1 = {1, 0};
```

Beachten Sie, dass der Datentyp hier `int` ist! Rufen Sie `rotate(e1, M_PI/4)` auf. Hätten Sie erwartet, dass dies funktioniert?

d) Lesen Sie sich den Artikel [1] durch. Warum können Sie nicht ohne weiteres Zutun auf die explizite Konvertierung beim Bilden des Skalarproduktes von `vec<3,double>` und `vec<3,int>` verzichten?

4) Bonus: Matrix-Klasse

Implementieren Sie der selben Idee folgend eine Matrix-Klasse `mat<M,N,T>` für $M \times N$ Matrizen mit Elementen vom Typ `T`. Überlegen Sie sich in welchem Format Sie die Daten speichern. Implementieren Sie insb. Matrix-Matrix-, sowie Matrix-Vektor-Multiplikation. Sie sollten das Augenmerk hier keinesfalls auf Effizienz legen und können die Vektor-Klasse verwenden um die Matrix-Operationen zu implementieren. Testen Sie ihre Klasse ausgiebig.

Literatur

[1] C++ Reference: Implicit Conversion,
`http://en.cppreference.com/w/cpp/language/implicit_conversion`