Anmerkung: Hinweise zur Klausur wurden in den letzten Tagen per Mail an Sie verschickt. Lesen Sie sich diese durch und wenden Sie sich bei Problemen an info14@conan.iwr.uni-heidelberg.de.

ÜBUNG 12.1 DIE STANDARD TEMPLATE LIBRARY (STL)

Die C++ Standard Library, die in der Regel mit dem Akronym ihrer Vorgängerin STL bezeichnet wird, stellt Container und Algorithmen zur Verfügung. Fast jedes größere Programm profitiert von der Verwendung der STL. Nicht nur erspart man sich auf diese Weise viel Programmieraufwand, weil die abstrakten Datentypen nicht selbst implementiert werden müssen, man reduziert auch die Wahrscheinlichkeit, subtile Bugs zu produzieren.

Schreiben Sie eine Vektorklasse, die einen `std::vector` zur Verwaltung der Einträge verwendet. Übergeben Sie den Typ der Elemente und die Länge des Vektors als Templateparameter. Die Klasse soll Methoden bereit stellen, die folgende Operationen erlauben:

- Addition zweier Vektoren
- Multiplikation mit einem Skalar
- Skalarprodukt
- Anwendung einer beliebigen Funktion (als Funktor) auf die Einträge
- Berechnung von Maximum, Minimum und Mittelwert

Das genaue Interface ist Ihnen überlassen. Übergeben Sie den Methoden ihre Argumente templatisiert, z.B. muss der Skalar bei der Skalarmultiplikation nicht den Typ der Vektoreinträge haben. Schreiben Sie ein Programm, das Ihre Klasse testet. Benutzen Sie jeweils in mindestens einer Methode

- den Elementzugriff über `operator[]`, durch den sich der `std::vector` wie ein C-Array verwenden lässt
- den Zugriff über die Containeriteratoren, der auch mit anderen Containern (z.B. `std::list`) funktioniert
- die vordefinierten STL-Algorithmen, die ebenfalls mit anderen Containern funktionieren

Die in der STL definierten Container und die Komplexität ihrer Methoden können Sie z.B. unter `http://www.cplusplus.com/reference/stl/` finden, während die Algorithmen unter `http://www.cplusplus.com/reference/algorithm/` stehen.

10 Punkte

ÜBUNG 12.2 FUNKTOREN UND STATISCHER POLYMORPHISMUS

Funktoren sind, wie in der Vorlesung bereits definiert, Klassen, deren `operator()` überladen ist. Sie verhalten sich damit effektiv, wie Funktionen, haben jedoch durch ihre privaten Mitglieder ein "Gedächtnis". Betrachten Sie das folgende Beispiel eines Funktors:

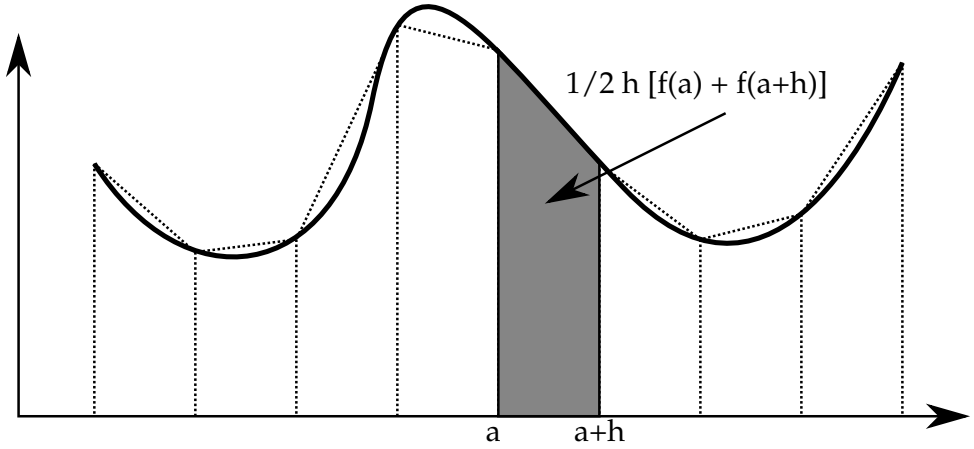
```
class Quadrat
{
  public:
  double operator()(double x)
  {
    return x*x;
  }
```

```
};

int main()
{
  Quadrat f;
  double y = f(3.0); // an dieser Stelle wird Quadrat::operator()(3.0) ausgewertet!
}
```

In der numerischen Mathematik nähert man Integrale mittels sogenannter Quadraturformeln. Dabei wird das Integrationsintervall $[a, b]$ in n gleich große Intervalle unterteilt und die Funktion auf jedem dieser Intervalle durch ein Polynom angenähert. Für $n \to \infty$ konvergiert dieser Wert gegen das Integral. Man erhält ein Näherungsverfahren, indem man n gerade so groß wählt, dass der Fehler im Integral klein genug wird. Verwendet man lineare Polynome ergibt sich die sogenannte "summierte Trapezregel" (dabei ist die Intervalllänge $h = (b - a)/n$):

$$I_n(f; a, b) = \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(a + ih) + f(b)\right)$$



a) Schreiben Sie eine Funktion `trapezregel`, welche obige Formel implementiert. Neben der Anzahl an Intervallen `n` und den Integrationsgrenzen `a` und `b` muss dieser Funktion auch die zu integrierende Funktion übergeben werden. Da man die Funktion `trapezregel` für beliebige Funktionen verwenden will, muss man hierfür eine Technik des Polymorphismus verwenden. Im Falle des dynamischen Polymorphismus würde man eine abstrakte Basisklasse `Funktion` einführen und der Funktion `trapezregel` eine Referenz auf ein Objekt dieser Basisklasse übergeben. Wir wollen das Problem allerdings mittels statischem Polymorphismus lösen. Dazu realisieren wir unsere Funktion als Funktor und übergeben diesen an die Funktion. Der exakte Typ des Funktors ist dabei ein Template-Parameter der Methode `trapezregel`. Testen Sie Ihre Funktion mit einem Beispielfunktor. [*6 Punkte*]

b) Nennen Sie Vor- und Nachteile der Verwendung von statischem Polymorphismus gegenüber dynamischem Polymorphismus. [*4 Punkte*]

10 Punkte