Übungen zur Vorlesung

"Objektorientiertes Programmieren im Wissenschaftlichen Rechnen"

Dr. O. Ippisch, Dr. C. Engwer

Abgabe am 25. 05. 2010 in der Vorlesung

ÜBUNG 1 EXCEPTIONS UND DESTRUKTOREN

Was passiert, wenn in einem Destruktor eine Exception geworfen wird? Betrachten wir nebenstehendes Programm.

```
1  #include <iostream>
2  #include <string>
3  #include <exception>
4
5  // class Foo throws in the destructor
6  class Foo {
7  public:
8      ~Foo () {
9          throw my_exception("Foo exception");
10     }
11 };
12
13 // class Bar throws in the constructor
14 class Bar {
15 public:
16     Bar () {
17         throw my_exception("Bar exception");
18     }
19 };
20
21 int main()
22 try {
23     Foo f;
24     Bar b;
25 }
26 catch (const std::exception & e){
27     std::cout << "ERROR:" << e.what() << std::endl;
28 }
```

- Ergänzen Sie das nachfolgende Beispiel um die Definition der Klasse `my_exception`, welche von `std::exception` abgeleitet ist. Die Klasse `my_exception` soll in ihrem Konstruktor eine Fehlermeldung als `std::string` übergeben bekommen. Von der virtuellen Methode `what()` soll diese Fehlermeldung dann zurückgegeben werden.

- Was beobachten Sie beim Ausführen des Programms?

- Versuchen Sie, das Verhalten zu erklären.

- Der C++-Standard schreibt in Sektion 15.2, Punkt 3:

  3 The process of calling destructors for automatic objects constructed on the path from a try block to a throw-expression is called "stack *unwinding*." [Note: If a destructor called during stack unwinding exits with an exception, terminate is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor. —end note]

  Warum ist das unter *Note* beschriebene Verhalten sinnvoll?

Weiterlesen: Eine schöne Zusammenstellung der verschiedenen Optionen im Zusammenspiel von Destruktoren und Exceptions finden Sie unter `http://www.kolpackov.net/projects/c++/eh/dtor-1.xhtml`

6 Punkte

In der Vorlesung wurde das Konzept der Interfaces anhand des Sprachmittels der virtuellen Methoden eingeführt. Als Beispiel wurde die Numerische Integration vorgestellt. Dieses Beispiel wollen wir nun aufgreifen und erweitern.

Ziel ist es, eine Code-Bibliothek zu schreiben, die es ermöglicht, das Integral einer beliebigen Funktion $f(t)$ auf einem gegebenen Intervall $t \in [A, B]$ ($A, B \in \mathbb{R}$) durch numerische Integration näherungsweise zu bestimmen. Das Intervall $[A, B]$ wird hierzu in N gleich große Teilintervalle der Länge Δt zerlegt.

$$\Delta t = \frac{B - A}{N}, \quad t_i = A + \Delta t \cdot i, \quad i = 0, \ldots, N$$

Auf jedem der Teilintervalle kann dann das Integral durch eine geeignete Quadraturregel abgeschätzt werden:

$$\int_A^B f(t)dt = \sum_{i=0}^{N-1} \left(Q_{t_i}^{t_{i+1}}(f) + E_{t_i}^{t_{i+1}}(f) \right)$$

wobei Q_a^b das Ergebnis der Quadraturregel auf dem Intervall $[a, b]$ bezeichnet und E_a^b den Fehler. In der Regel liegen den Quadraturregeln Verfahren der Polynominterpolation zugrunde. Für jede Quadraturregel lässt sich daher angeben, welche Polynome noch exakt integriert werden und von welcher Ordnung der Fehler ist, falls die Funktion nicht exakt integriert werden kann.

In dieser Übung wollen wir uns auf zwei Quadraturregeln beschränken:

- Trapezregel: ${Q_{\text{Trapez}}}_a^b(f) = \frac{b-a}{2}(f(a) + f(b))$
  integriert Polynome 1. Ordnung exakt, der Fehler ist $O(\Delta t^2)$.
- Simpsonregel: ${Q_{\text{Simpson}}}_a^b(f) = \frac{b-a}{6} \cdot \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$
  integriert Polynome 2. Ordnung exakt, der Fehler ist $O(\Delta t^4)$.

Ein guter Test für eine solche numerische Integration ist die Untersuchung der Konvergenzordung. Für ein Problem mit bekannter Lösung berechnen Sie den Fehler E_N für N Teilintervalle und für $2N$ Teilintervalle. Bei der Verdoppelung der Teilintervalle halbiert sich die Intervallbreite Δt. Dadurch verringert sich der Fehler. Der Quotient

$$EOC = \frac{log(E_N / E_{2N})}{log(2)}$$

ermittelt experimentell die Konvergenzordnung der Fehlers (für die Trapezregel sollte man hier Ordnung 2 beobachten).

1. Entwerfen Sie Interfaces für die verschiedenen Konzepte. Hierzu gehen wir wie folgt vor:
   - Zunächsteinmal identifizieren wir die abstrakten Konzepte in unserer Problemstellung.
     - Wir wollen Integrale verschiedener Funktionen auswerten; es gibt also erstmal die Funktion als abstraktes Konzept.
     - Ein Integral wird als Summe über Integrale von Teilintervalle dargestellt.
     - Auf jedem Teilintervall wird eine geeignete Quadraturformel angewandt.
     - Die Quadraturformel wertet die zu integrierende Funktion an gewissen Punkten aus um die Näherungslösung bestimmen.
   - Wie lassen sich diese abstrakten Konzepte auf Interfaces abbilden? Üblicherweise führt man für jedes Konzept eine Basisklasse, als Interfacebeschreibung ein.
     - Was ist ein geeignetes Interface für eine Funktion? Vielleicht kennen Sie ja bereits eines aus der Vorlesung.
     - Eine Quadraturregel wertet eine Funktion auf einem gegebenen Intervall aus, wie sähe dafür ein geeignete virtuelle Methode aus?

- Zu jeder Quadraturregel lässt sich außerdem der Grad der exakt integrierbaren Polynome, sowie die Konvergenzordnung des Integrationsfehlers angeben.
- Die eigentliche Integration wird mit einer Quadraturregel und einer Funktion parametrisiert. Sie wertet das Integral der Funktion auf einem zu spezifizierenden Intervall aus, indem sie dieses in N Teilintervalle unterteilt. Außerdem muss die zu verwendende Quadraturformel angegeben werden.

2. Beschreiben Sie Ihre Interfaces und erklären Sie die Designentscheidungen. Stellen Sie die Interfaces als UML Diagramm dar.

3. Implementieren Sie die Interfaces mit dynamischem Polymorphismus. Schreiben Sie Implementierungen für alle oben aufgeführten Varianten der abstrakten Konzepte.

4. Testen Sie ihre Implementierung mit der Integration von

   - $2t^2 + 5$ auf dem Intervall $[-3, 13]$
   - $\frac{t}{\pi}\sin(t)$ auf dem Intervall $[0, 2\pi]$

   Jede Testfunktion soll zwei weitere Methoden haben:

   - `void integrationInterval(double & l, double & r) const;` um die Grenzen des Intervalls auszugeben.
   - `double exactIntegral() const;` um die exakte Lösung auszugeben.

   Führen Sie hierzu eine Basisklasse Testfunktion als Erweiterung des Funktionsinterfaces ein. Schreiben Sie eine freie Funktion, die für ein gegebenes Testproblem und eine gegebene Quadraturregel die Konvergenzordnung bestimmt. In einer verbesserten Variante kann diese freie Funktion noch die erwartete Konvergenzordnung der Quadraturregel mit der beobachteten Konvergenzordnung vergleichen. Beachten Sie, dass die angegebene Konvergenzordnung nur asymptotisch gilt.

12 Punkte

ÜBUNG 3 *Bonusaufgabe*: ADAPTIVE INTEGRATION

Eine Verbesserung gegenüber der Integration mittels äquidistanter Intervalle ist eine adaptive Integration. Die Idee hierbei ist es, nur dort kleine Intervalle Δt zu verwenden, wo der Fehler groß ist.

Wem jetzt noch langweilig ist `;-)` kann als Ergänzung zu der Integratorklasse aus der vorangegangenen Übung eine Klasse zur adaptiven Integration schreiben. Diese neue Variante soll die gleichen Interfaces für Funktionen und Quadraturregeln verwenden.

Den Fehler in einem Teilintervall kann man abschätzen, indem man das Ergebnis der Quadratur mit dem Ergebnis eines einmal verfeinerten Intervalls vergleicht, d.h. man schätzt den lokalen Fehler durch hierarchisch Verfeinerung ab. Das Ziel ist es in allen Teilintervallen den Fehler ungefähr gleich groß zu haben. Die Teilintervalle mit großem Fehler werden erneut unterteilt. Man startet mit einem einzelnen Teilintervall und wiederholt die Fehlerschätzung und Verfeinerung, bis man die Anzahl der vorgesehenen Teilintervalle erreicht hat.

Um nicht ständig die Integrale der einzelnen Teilintervalle neu auswerten zu müssen, ist es hilfreich die Integrale und die Fehler in einer gesonderten Datenstruktur zu speichern. Hierzu bietet sich z.B. ein doppelt verkettete Liste an, diese ist in der `C++` Standardbibliothek als `deque` verfügbar.

8 Punkte

Wie immer gilt: Kommentieren Sie Ihr Programm. Erklären Sie was Sie tuen.