Dirk Praetorius,
Marcus Page,
Thomas Führer,
Josef Kemetmüller

Sommersemester 2013
18.06.2013

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 13

Aufgabe 13.1. Implementieren Sie eine Klasse `Person`, welche die Datenfelder `name` und `adresse` enthält. Schreiben Sie auch die zugehörigen Zugriffsfunktionen. Leiten Sie von dieser Klasse eine Klasse `Student` ab, welche die zusätzlichen Datenfelder `matrikelnummer` und `studium` enthält. Leiten Sie von der Klasse `Person` auch eine Klasse `Arbeiter` ab. Erweitern Sie diese Klasse um die Datenfelder `gehalt` und `arbeit`. Überlegen Sie sich auch, welche Zugriffsspezifizierer für welche Datenelemente verwendet werden und begründen Sie dies.

Aufgabe 13.2. Erstellen Sie für die Basisklasse `Person` aus Aufgabe 13.1 eine Methode `void print()`, welche den Namen und die Adresse einer Person am Bildschirm ausgibt. Redefinieren Sie diese Funktion dann jeweils für die Klassen `Student` und `Arbeiter` (Es sollen die zusätzlich definierten Datenelemente auch ausgegeben werden). Schreiben Sie dann noch ein Hauptprogramm, in welchem die `print`-Funktionen der verschiedenen Klassen getestet werden sollen.

Aufgabe 13.3. Gegeben seien die Klassen `CRectangle` und `CTriangle` die wie folgt von `CPolygon` abgeleitet sind:

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void setValues (int a, int b)
      { width=a; height=b; }
  };

class CRectangle: public CPolygon {
  public:
    double getArea()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    double getArea()
      { return (width * height / 2.); }
  };
```

Schreiben Sie ein aufrufendes Hauptprogramm in dem Sie ein Array von 5 Polygonen anlegen. Drei der Polygone sollen hierbei Dreiecke, und zwei Rechtecke sein. Befüllen Sie Ihre Polygone mittels `setValues` mit Daten.

Aufgabe 13.4. Ihr Array aus Aufgabe 13.3 soll nun in einer Schleife durchlaufen werden. Hierbei soll die Fläche jedes einzelnen Polygons ausgegeben werden. Wie können Sie das bewerkstelligen?

Aufgabe 13.5. Die Ausgabe aus der vorangegangenen Aufgabe ist noch immer etwas armselig. Zusätzlich zur Gesamtfläche des Polygons, soll jedes Polygon nun auch seinen Typ mitteilen. Durchlaufen Sie ihr Array wieder in einer Schleife und stellen Sie sicher, dass jede einzelne Ausgabe etwa wie folgt lautet:

```
Ich bin ein Rechteck und mein Flächeninhalt ist 24.
```

Erweitern Sie alle nötigen Klassen um diese Funktionalität sicherzustellen.

Aufgabe 13.6. *Achtung:* Nicht schrecken! Folgendes Beispiel ist in sehr wenigen Zeilen programmierbar, wenn Sie die Methoden der Klasse `list` verwenden. (Wirklich!)
Wir wollen nun eine Klasse schreiben, welche Funktionen f von $\mathbb{R}$ nach $\mathbb{R}$ repräsentiert. Damit wir in weiterer Folge verschiedenartige Funktionen als Elemente der Menge $\mathcal{F} := \{f : \mathbb{R} \to \mathbb{R}\}$ verwenden können, benötigen wir eine polymorphe Basisklasse. Schreiben Sie also eine abstrakte polymorphe Basisklasse `RealFunc`, welche den `operator()` mittels `virtual double operator()(double)` deklariert. Leiten Sie nun von dieser Klasse eine Klasse `SinusTypeFunc` ab, deren Objekte die Menge der Funktionen $\mathcal{S} := \{f \in \mathcal{F} : f(x) = \alpha \sin(\beta x), \alpha, \beta \in \mathbb{R}\}$ darstellt. Die Objekte der Klasse sollen also lediglich die Werte `alpha` und `beta` als Member-Variable enthalten. Entsprechend ist ein Konstruktor `SinusTypeFunc(double alpha, double beta)` zu programmieren. Die Funktion `operator()` soll so redefiniert werden, dass sie für die enthaltenen `alpha` und `beta` des Objektes und ein übergebenenes `x` den Funktionswert $\alpha \sin \beta x$ mithilfe der Sinusfunktion aus der Mathematikbibliothek zurückliefert.
Nun wollen wir damit eine Klasse für die Menge der ungeraden trigonometrischen Polynome erstellen. Das sind genau Summen von `SinusTypeFunc`-Funktionen. Schreiben Sie dazu eine Klasse `SinusSum`, welche öffentlich von `RealFunc` abgeleitet wird. Diese soll eine `list<...>`e von `SinusTypeFunc`-Objekten speichern. Schreiben Sie also die Funktionalitäten der nachfolgenden Klasse.

```
class SinusSum : public RealFunc{
public:
   SinusSum(SinusTypeFunc f);
   double operator()(double x);
   friend SinusSum operator+(const SinusSum& f, const SinusSum& g);
protected:
   std::list<SinusTypeFunc> summands;
};
```

Der Konstruktor `SinusSum(SinusTypeFunc f)` soll dabei lediglich `f` in die leere Liste einfügen. Der `operator+` soll ein neues `SinusSum`-Objekt zurückliefern in das die Summandenlisten von `f` und `g` aneinandergereiht werden.
Hier noch ein Beispielaufruf der Funktionen.

```
SinusTypeFunc s (1,2);
SinusSum F = s+s+s+s+s;
SinusTypeFunc s2 (5,2);
std::cout << F(3.14) << std::endl;
std::cout << s2(3.14) << std::endl;
```

Aufgabe 13.7. In dieser Aufgabe beschäftigen wir uns mit der Integration stetiger Funktionen. Zu einer gegebenen Funktion f und Integrationsgrenzen a und b möchten wir also das Integral

$$I(f) := \int_a^b f(x)dx$$

möglichst genau annähern. Zu diesem Zweck gibt es in der numerischen Mathematik diverse Möglichkeiten (Quadraturformeln). Die bekannteste ist sicherlich die Ein-Punkt-Quadratur

$$I_{M,h}(f) := (b-a)f\big(\frac{a+b}{2}\big) \approx \int_a^b f(x)dx = I(f)$$

Eine weitere Möglichkeit ergibt sich durch die sogenannte *Simpsonregel*

$$I_{S,h} := \frac{b-a}{6}\big(f(a) + 4f(\frac{a+b}{2}) + f(b)\big) \approx \int_a^b f(x)dx = I(f).$$

Letztere benötigt zwar mehr Funktionsauswertungen, zeichnet sich allerdings durch eine höhere Genauigkeit aus.
Die numerische Quadratur soll nun als Programm umgesetzt werden. Schreiben Sie hierzu eine abstrakte Klasse `Integrator`, welche die beiden Integrationsgrenzen a und b als Member abspeichert und eine (abstrakte) Methode `double integrate (RealFunc& f)` bereitstellt. Dieser sollen später Referenzen auf Funktionen vom Typ `RealFunc` aus der vorherigen Aufgabe übergeben werden. Leiten Sie von `Integrator` ferner die beiden Klassen `MidpointRule` und `SimpsonRule` ab. In diesen beiden Klassen sollen die entsprechenden Integrationsroutinen, wie oben angegeben, implementiert werden.

Aufgabe 13.8. Manchmal kann es sinnvoll sein das Integrationsgebiet in mehrere kleine Intervalle T_i aufzuteilen und einzeln zu integrieren, d.h.

$$I_{M,h}(f) = \sum_{T_i} I_{M,h}(f|_{T_i})$$

Schreiben Sie hierzu eine Klasse `IteratedIntegrator`, die Sie von `Integrator` ableiten. Diese soll, zusätzlich zu den Integrationsgrenzen auch eine Liste (`list<Integrator*>`) von Pointern auf Integratoren speichern, welche die jeweiligen Intervalle bearbeiten. Außerdem verfügt die Klasse über eine Methode `refine`, mit der sich das Intervall weiter aufteilen lässt. Ein Objekt von Typ `IteratedIntegrator` bekommt nun bei der Erstellung einen Pointer auf einen Integrator übergeben. Wird `refine` aufgerufen, so werden alle Teilgebiete T_i halbiert, und entsprechende Integratoren für die neuen Intervalle werden in die Liste mit aufgenommen. Die `integrate`-Methode führt schließlich alle Integrationen auf den Teilintervallen aus und summiert das Ergebnis.
Berechnen Sie die folgenden Integrale

$$I_1 = \int_{-1}^{1} 13\sin(2x)$$
$$I_2 = \int_{1}^{2} 3\sin(2x) + \sin(3x) - 0.5\sin(4x)$$

mithilfe der Quadraturformeln aus der vorherigen Aufgabe. Vergleichen Sie das Ergebnis mit dem exakten Wert des Integrals. Was beobachten Sie, wenn Sie den `IteratedIntegrator` immer weiter verfeinern.
Hinweis: Da sie beim Verfeinern eine dynamische Kopie eines Objektes benötigen von dem Sie nur wissen, dass es von `Integrator` abgeleitet ist benötigen Sie eine polymorphe Kopierfunktion. Diese Sei hier unter `Clone()` angegeben. In der Funktion `refine()` können Sie diese dann verwenden um die Integratoren zu kopieren.

```
class Integrator{
public:
   virtual Integrator* Clone() = 0;
...
};
class MidPointRule : public Integrator{
public:
   Integrator* Clone() { return new MidPointRule(*this); }
....
};
class SimpsonRule : public Integrator{
public:
   Integrator* Clone() { return new SimpsonRule(*this); }
....
};
class IteratedIntegrator : public Integrator{
public:
   Integrator* Clone() { return new IteratedIntegrator(*this); }
...
};
```