Prof. Dr. Carsten Meyer
Fachbereich Informatik & Elektrotechnik
Fachhochschule Kiel

Programmieren in C++ – Sommersemester 2015

Aufgabenblatt 3

Abgabe: Donnerstag, 04.06.2015, 11:30

Aufgabe 1 (Definition von Klassen, Konstruktor, Destruktor, statische Klassenelemente):

Schreiben Sie eine Klasse **`Auto`** mit (privaten) Instanzvariablen Leistung (vom Typ **`int`**) und Fabrikat (vom Typ **`char*`**) sowie (privater) Klassenvariabler **`anzahl`**. Die Klasse soll einen Standardkonstruktor und einen überladenen Konstruktor haben, der die Leistung und das Fabrikat über entsprechende Parameter setzt; außerdem soll ein geeigneter Destruktor definiert werden. Geben Sie bei jedem Konstruktor-/Destruktoraufruf die Anzahl der vorhandenen Autos auf dem Bildschirm aus. Neben den üblichen get-/set-Methoden soll die Klasse eine Instanzmethode **`print()`** haben, die die Leistung und das Fabrikat auf dem Bildschirm ausgibt, sowie eine Klassenmethode **`getAnzahl()`**, die die Anzahl der vorhandenen Autos zurückgibt.

Im Hauptprogramm soll der Benutzer gefragt werden, wieviele Autos er erzeugen will. Für jedes zu erzeugende Auto soll der Benutzer Leistung und Fabrikat eingeben; diese Werte werden für die Instanzvariablen des entsprechenden Autos verwendet. Welcher Konstruktor kommt dabei für jedes Auto zum Einsatz? Nach erfolgter Eingabe soll zur Kontrolle die **`print()`**-Funktion für alle Autos aufgerufen werden.

Legen Sie weiterhin ein Feld von 3 Autos über eine Initialisierungsliste an. Welcher Konstruktor wird nun verwendet? Rufen Sie zur Kontrolle wiederum die **`print()`**-Funktion für alle Autos auf. Setzen Sie danach das Attribut „**`Fabrikat`**“ für alle Autos auf „Volkswagen“ und kontrollieren Sie das Ergebnis.

Die Lösung für diese Aufgabe soll die Implementierung und die Bildschirmausgabe des Programms umfassen.

(frei nach Karsten Robel, FH Lausitz)

Aufgabe 2 (Definition von Klassen, Implementierungs- und Headerdatei):

a) Schreiben Sie ein Programm, das zwei Klassen definiert: **`Point`** und **`Circle`**. Um bei einer möglichen späteren Erweiterung des Programms den Überblick zu behalten, nutzen Sie bitte für jede Klasse eine eigene Headerdatei, die die Klassendeklaration enthält, und jeweils eine Quelldatei mit der Klassenimplementierung. Hier, wie immer, wenn nicht anders gefordert, sollen die Elementvariablen gekapselt, d.h. privat, sein.

Die Klasse **`Point`** repräsentiert das geometrische Objekt Punkt und soll die Elementvariablen x und y vom Typ **`double`** haben. Diese beiden Elementvariablen repräsentieren die x- und y-Koordinaten eines Punktes. Die beiden Elementvariablen sollen mittels eines geeigneten Konstruktors aus zwei Parametern **`a`** und **`b`** vom Typ **`double`** gesetzt werden können; darüber hinaus soll es einen Standardkonstruktor geben. Weiterhin soll es eine Elementfunktion **`move(double dx, double dy)`** geben, die den Punkt um die Werte dx und dy in x- bzw. y-Richtung verschiebt. Zusätzlich soll es eine Elementfunktion **`print()`** geben, die einen String mit den Koordinaten des Punktes zurückgibt (siehe Hinweis unten). Die Methode **`print()`** benötigt einen Parameter vom Typ **`bool`**, der steuert, ob nach der Ausgabe ein Zeilenvorschub (**`endl`**) durchgeführt wird oder nicht. Defaultwert für diesen Parameter soll **`true`** sein.

Die Klasse **`Circle`** repräsentiert einen Kreis und soll als Elementvariablen **`centre`** vom Typ **`Point`** und **`radius`** vom Typ **`double`** haben. Die Variable **`centre`** steht also für den Mittelpunkt des Kreises. Auch hier sollen die Elementvariablen durch einen geeigneten Konstruktor gesetzt werden können (beachten Sie auch hier die Standardkonstruktor-Funktionalität!). Zusätzlich gibt es (ebenso wie bei der Klasse **`Point`**) eine Funktion **`move(…)`**, die hier den Kreis verschiebt, indem der Mittelpunkt des Kreises verschoben wird. Schließlich soll es auch hier eine Elementfunktion **`print()`** geben, die einen String mit den Koordinaten des Mittelpunktes und den Radius zurückgibt (siehe Hinweis unten). Die Methode **`print()`** soll wieder einen Parameter vom Typ **`bool`** haben, der steuert, ob nach der Ausgabe ein Zeilenvorschub (**`endl`**) durchgeführt wird oder nicht. Defaultwert für diesen Parameter soll **`true`** sein.

Die Ausgabe des unten gegebenen Hauptprogramms soll wie folgt aussehen:

```
(1.3, 1.4)
<(1.3, 1.4), 0.5>
<(3, 3), 0.5>
```

```
// Hauptprogramm
int main (int argc, char* argv[] )
{
   Point p(1.3, 1.4);
   cout << p.print();
   Circle c(p, 0.5);
   cout << c.print();
   c.move(1.7, 1.6);
   cout << c.print();

   return 0;
}
```

Die Lösung für diese Aufgabe soll die Implementierung und die Bildschirmausgabe des Programms umfassen.

(nach HS Esslingen)

Hinweis:

Die Umwandlung einer Zahl vom Typ **double** in einen String kann z.B. über die folgende Funktion erfolgen (wegen **ostringstream** muß die **sstream**-Bibliothek inkludiert werden):

```
string toString( double a )
{
      ostringstream ss;
      ss << a;
      return ss.str();
}
```

b) Gegeben seien die Klassendefinitionen **Point** und **Circle** aus der vorherigen Aufgabe. Schauen Sie sich das folgende Hauptprogramm an. Welche Zeilen sind nicht korrekt und warum nicht? Verifizieren Sie Ihre Aussagen mit einem Kompilerversuch. Korrigieren Sie das Programm sinnvoll.

   Die Lösung für diese Aufgabe soll die Implementierung und die Bildschirmausgabe des Programms umfassen.

```
 0 // Hauptprogramm
 1 int main (int argc, char* argv[] )
 2 {
 3   Point p(4, 5);
 4   cout << p.x << endl;
 5   cout << p.print;
 6   move(3,4);
 7   cout << p.print();
 8
 9   Point * ptr = new Point(7,9);
10   double *fpt = ptr.x
11   cout << ptr->x << endl;
12   cout << ptr.print();
13   *ptr->move(3,11);
14   ptr->move(3,11);
15   cout << ptr->print();
16
17   return 0;
18 }
```

(Quelle: HS Esslingen)

Aufgabe 3 (Definition von Klassen, Elementfunktionen, const, Implementierungs- und Headerdatei):

Schreiben Sie eine Klasse **`IntMenge`**, bestehend aus den zwei Dateien **`IntMenge.h`** und **`IntMenge.cpp`**. Diese Klasse soll die mathematische Menge der ganzen Zahlen nachbilden. Es sollen die folgenden einfachen (Element)Funktionen möglich sein:

- **`void hinzufuegen(int el):`** Element el hinzufügen, falls es noch nicht in der Menge enthalten ist. Ansonsten nichts tun. Siehe Bemerkung unten.
- **`void entfernen(int el):`** Element el aus der Menge entfernen, falls es vorhanden ist. Ansonsten nichts tun. Siehe Bemerkung unten.
- **`bool istMitglied(int el):`** Rückgabewert „true", falls Element el in der Menge enthalten ist. Sonst Rückgabewert „false".
- **`int size():`** Gibt die Anzahl der Elemente in der Menge zurück.
- **`void anzeigen():`** Gibt die Elemente der Menge (ungeordnet) aus.
- **`void loeschen():`** Löscht alle Elemente der Menge. Siehe Bemerkung unten.
- **`int getMax():`** Gibt das größte Element der Menge zurück.
- **`int getMin():`** Gibt das kleinste Element der Menge zurück.

Private Hilfsfunktion:

- **`int finden(int el):`** Gibt die Position des Elements el in der Menge zurück oder –1, falls es nicht vorhanden ist.

Zum Speichern der Daten wird intern in der Klasse das Vektor-Objekt `vec` verwendet:

```
std::vector<int> vec;
```

Ein `vector` ist ein Klassentemplate aus der "Standard Template Library" (STL) für einen Sequenzcontainer, dessen Größe geändert werden kann; um die **`vector`** – Klasse zu verwenden, muß der Header **`<vector>`** eingebunden werden. In der Klasse **`vector`** sind zahlreiche Methoden für die Manipulation von **`vector`** – Objekten implementiert; z.B. ist der Indexoperator überladen, so daß Sie einfach die von den Feldern bekannte Syntax verwenden können, um auf ein Datenelement `i` zuzugreifen:

```
vec[i] = 2; // Beispiel für Zuweisung an das i-te Element des Vektors
```

Die (hier) wichtigsten Methoden, die für ein **`vector`**–Objekt **`vec`** zur Verfügung stehen, sind:

- **`vec.size():`** Rückgabe: Anzahl der im Vektor gespeicherten Elemente als Ganzzahl.
- **`vec.push_back(el):`** Allokiert Speicher für ein neues Element und fügt es am Ende in den Vektor ein.
- **`vec.resize (12):`** Setzt die ursprünglich reservierte Größe des Vektors auf 12.
- **`vec[5]:`** Liefert das 6. Vektorelement als Leftvalue. Keine Fehlermeldung bei Bereichsüberschreitung.
- **`vec.at (5):`** Liefert das 6. Vektorelement als Leftvalue. Wirft out_of_range-Exception bei Bereichsüberschreitung.

Neben der Klasse **`vector`** gibt es in der STL noch weitere Klassen (z.B. **`list`**, **`deque`**) und Algorithmen (z.B. **`find`**, **`sort`**), siehe hierzu weiterführende Literatur oder z.B. http://www.willemer.de/informatik/cpp/stl.htm.

Für die Lösung der Aufgabe verwenden Sie – soweit möglich – *konstante* Elementfunktionen. Testen Sie Ihre Klasse mit dem unten angegebenen Hauptprogramm.

Bemerkungen:

- Die Menge darf jedes Element höchstens einmal enthalten.
- Die Elemente der Menge werden in der Reihenfolge der Eingabe gespeichert.
- Beim Entfernen des letzten Elements müssen Sie den Vektor der Einfachheit halber nicht entsprechend verkleinern, sondern sich einfach die Position des neuen letzten Elements merken (und somit das im Vektor tatsächlich an letzter Stelle vorkommende einfach ignorieren). Soll ein Element gelöscht werden, das nicht an letzter Stelle steht, so können Sie es einfach mit dem letzten tauschen und anschließend das letzte Element wie oben beschrieben löschen.
- Bei Einfügen eines neuen Elements werden dann zunächst die evt. schon vorhandenen leeren Vektorstellen am Ende gefüllt, bevor mit **`push_back()`** neue Elemente angelegt werden.

.

Hauptprogramm:

```
#include <iostream>
#include <vector>
#include "IntMenge.h"

using namespace std;

int main(int argc, char *argv[])
{
    IntMenge menge;
    cout << "fuege Elemente von 5 bis -2 hinzu" << endl;
    for ( int i=5; i > -3; i-- )
        menge.hinzufuegen(i);
    menge.anzeigen();
    cout << endl;

    cout << "fuege Elemente von 3 bis 7 hinzu" << endl;

    for ( int i=3; i < 8; i++ )
        menge.hinzufuegen(i);
    menge.anzeigen();
    cout << "menge.istMitglied(3) " << menge.istMitglied(3) << endl;
    cout << "menge.istMitglied(-11) " << menge.istMitglied(-11) <<
endl;
    cout << endl;

    cout << "fuege Element 11 hinzu" << endl;
    menge.hinzufuegen(-11);
    menge.anzeigen();
    cout << "menge.istMitglied(-11) " << menge.istMitglied(-11) <<
endl;
    cout << endl;
```

```
    cout << "fuege Element 2 hinzu; entferne Elemente 99 und -9" <<
endl;
    menge.hinzufuegen(2);   // keine Wirkung, 2 gibt es schon
    menge.entfernen(99);    // keine  Wirkung, nicht vorhanden
    menge.entfernen(-9);    // keine Wirkung, nicht vorhanden
    menge.anzeigen();
    cout << endl;

    cout << "fuege Quadratzahlen hinzu" << endl;
    for (int i = 11; i < 16; i++)
    {
        menge.hinzufuegen(i*i);
    }
    menge.anzeigen();
    cout << endl;
    cout << "Anzahl = " << menge.size() << " Minimum = " <<
menge.getMin() << endl;
    cout << "menge.getMin() = " << menge.getMin() << endl;
    cout << "menge.getMax() = " << menge.getMax() << endl;
    cout << endl;

    cout << "nacheinander Elemente entfernen..." << endl;
    int k = menge.getMin();
    while ( menge.size() > 0 )
    {
      int vorher = menge.size();
      menge.entfernen(k++);
      if ( vorher != menge.size() )
      {
        menge.anzeigen();
        cout << endl;
      }
    }
    menge.loeschen();
    return 0;
}
```

(Quelle: Breymann, Hochschule Bremen)