Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 3. Tag

Aufgabe 1 Vererbung und Polymorphie

Aufgabe 1.1 Klasse `Person`

Implementiere eine Klasse `Person`, die eine Person mit einem Namen darstellt. Füge allen Konstruktoren und Destrukoren eine Ausgabe auf die Konsole hinzu, um später den Lebenszyklus der Objekte besser nachvollziehen zu können.

```
class Person {
public:
   Person(const std::string& name);    // initialize the name of the person
   ~Person();                          // destructor
   std::string getInfo() const;        // get the name of the person
protected:
   std::string name;                   // the name of the person
};
```

Hinweise

- Verwende `#include <string>` um `std::string` zu verwenden.
- Um ein String-Literal an eine `std::string` Variable anzuhängen, musst du aus dem String-Literal zuerst ein `std::string`-Objekt machen: `std::string text = std::string("Name: ") + name;`.

Aufgabe 1.2 Klasse `Student`

Implementiere eine Klasse `Student`, die von `Person` erbt (`public`) und einen Studenten mit einer Matrikelnummer (ebenfalls `std::string`) modelliert. Rufe in der Initialisierungsliste den entsprechenden Konstruktor der Elternklasse `Person` mittels `Person(name)` auf. Füge allen Konstruktoren und Destrukoren eine Ausgabe auf die Konsole hinzu, um später den Lebenszyklus der Objekte besser nachvollziehen zu können.

```
class Student: public Person {   // public inheritance
public:
   Student(const std::string& name, const std::string& studentID); // init name and ID
   ~Student();                    // destructor
   std::string getInfo() const;  // override Person::getInfo() - get name and studentID
private:
   std::string studentID;        // the student ID of the student
};
```

Hinweise

- Erst ab C++11 gibt es die Möglichkeit mit dem Schlüsselwort `override` zu deklarieren, dass eine Funktion eine andere (virtuelle) überschreibt (vergleichbar mit der Annotation `@Override` in Java). Trotzdem zeigt dir Eclipse mit einem kleinen aufwärts zeigenden Dreieck links neben den Zeilennummern an, ob du eine Methode überschreibst oder nicht.
- Du kannst bei Bedarf die `getInfo()`-Implementation der Elternklasse `Person` von `Student` aus mittels `Person::getInfo()` aufrufen.

Übung zum C/C++-Praktikum - Tag 3

Aufgabe 1.3 Test

Erstelle nun in `main()` je eine Person und einen Studenten und gib deren Daten auf der Konsole aus. Vergewissere dich, dass bei `Student` auch die Matrikelnummer ausgegeben wird. Schau dir auch die Ausgaben der Konstruktoren und Destruktoren an, und versuche, diese nachzuvollziehen.
Implementiere dann folgende Funktion und teste deine Implementation erneut, indem du `printPersonInfo()` mit beiden Personentypen aufrufst.

```
void printPersonInfo(const Person *person);      // print person information on console
```

Hinweise

- Dadurch dass `Person` als `const` Zeiger übergeben wird, können auch Unterklassen von `Person`, wie z.B. `Student`, übergeben werden.

Aufgabe 1.4 Dynamic Dispatch bei `printPersonInfo`

Du merkst, dass `printPersonInfo()` unabhängig von übergebenem Personentyp immer nur den Namen der Person ausgibt, aber nicht die Matrikelnummer. Der Grund dafür ist, dass `getInfo()` nicht als `virtual` deklariert wurde und deshalb auch kein dynamischer Dispatch der Methode stattfindet. Deklariere daher `getInfo()` in beiden Klassen als `virtual`.
Teste deine Implementation erneut und vergewissere dich, dass nun immer die richtige Methode aufgerufen wird.

Hinweise

- Möchte man Methoden einer Basisklasse überschreiben, **muss** `virtual` in der Basisklasse gesetzt werden. In den abgeleiteten Klassen kann `virtual` weggelassen werden, es wird dann vom Compiler ergänzt.

Aufgabe 1.5 Virtueller Destruktor

Lege einen Studenten mit `new` dynamisch auf dem Heap an und speichere die Adresse in einem Zeiger auf eine `Person`. Lösche die Person anschließend mit `delete`.

```
Person *pTim = new Student("Tim", "321654");
delete pTim;
```

Analysiere die Konsolenausgabe. Es wird nur der Destruktor von `Person` aufgerufen, obwohl es sich um ein Objekt vom Typ `Student` handelt. Auch hier liegt es daran, dass kein dynamischer Dispatch bei der Zerstörung erfolgt. Deklariere deshalb in beiden Klassen den Destruktor als **virtual** und teste die Korrektheit der Destruktoraufrufe.

Hinweise

- Faustregel: Besitzt eine Klasse mindestens eine virtuelle Funktion, so sollte auch der Destruktor virtuell sein.

Übung zum C/C++-Praktikum - Tag 3

Aufgabe 2 Pure Virtual

In dieser Aufgabe wollen wir Vererbung und Polymorphie dazu nutzen, um mathematische Ausdrücke als Bäume von Primitivoperationen zu modellieren. Dazu werden wir eine abstrakte Oberklasse `Expression` mit der abstrakten Methode `compute()` erstellen. Einzelne Knotentypen wie Addition und Subtraktion werden von `Expression` abgeleitet und implementieren `compute()`, um die jeweilige Operation zu realisieren.

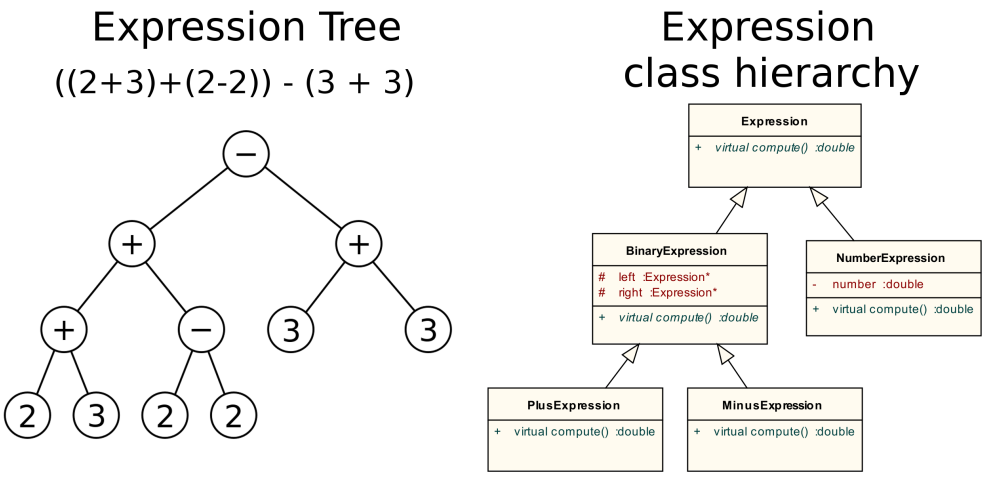

Abbildung 1: Abbildung: Beispielausdruck mit Ausdrucksbaum und Klassenhierarchie

a) **Klasse `Expression`** Schreibe die abstrakte Klasse `Expression`. Diese soll als Basisklasse für alle Ausdrücke dienen. Implementiere einen parameterlosen Konstruktor und einen virtuellen Destruktor, die je eine Meldung auf der Konsole ausgeben, sodass es bei der Ausführung ersichtlich wird, wann eine `Expression` erzeugt und wann zerstört wird. Deklariere außerdem eine abstrakte (pure `virtual`) Methode `virtual double compute() = 0;`, die das Ergebnis des Ausdrucks berechnen und zurückgeben soll.

b) **Klasse `NumberExpression`** Schreibe die Klasse `NumberExpression`, die ein (Baum-)Blatt mit einer Zahl darstellt. Dementsprechend soll `NumberExpression` von `Expression` erben und ein Attribut zum Speichern einer Zahl besitzen, das im Konstruktor initialisiert wird. Implementiere den Konstruktor und virtuellen Destruktor und versehe auch diese mit einer Konsolenausgabe. Die Methode `compute()` gibt die gespeicherte Zahl zurück.

c) **Klasse `BinaryExpression`** Schreibe die abstrakte Klasse `BinaryExpression` mit den `protected` Attributen `Expression *left, *right`. Implementiere den Konstruktor und virtuellen Destruktor mit entsprechender Ausgabe. Vergiss nicht, im Destruktor die beiden Zweige zu löschen.

d) **Klassen `Plus-` und `Minusexpression`** Schreibe die Klassen `PlusExpression` und `MinusExpression`, die von `BinaryExpression` erben und eine Addition bzw. Subtraktion realisieren. Implementiere die Kon- und Destruktoren sowie die `compute()` Methode.

e) **Test** Teste deine Implementation. Ein gutes Beispiel findest du in Abbildung weiter oben. Schaue dir die Ausgabe genau an und versuche anhand der gegebenen Klassenhierarchie die Reihenfolge der Erzeugung und Zerstörung von Objekten nachzuvollziehen.

Übung zum C/C++-Praktikum - Tag 3

Aufgabe 3 Fortsetzung Aufzugsimulator

Unser bisheriger Aufzugsimulator hat eine feste Strategie, nach der die einzelnen Stockwerke abgefahren werden. Mithilfe von Polymorphie können wir den Simulator so erweitern, dass die Strategie austauschbar wird.

Aufgabe 3.1 Vorbereitung

Lagere die bereits existierende Simulation des Aufzugs aus der `main`-Funktion in eine eigene Funktion `runSimulation()` aus. Die Funktion sollte das volle Gebäude als Parameter entgegennehmen und eine Liste (`std::list<int>`) der angefahrenen Stockwerke zurückgeben. Überlege dir, auf welche Art das Gebäude idealerweise übergeben werden sollte. Teste deine Implementation.

Aufgabe 3.2 Klasse `ElevatorStrategy`

Implementiere die Klasse `ElevatorStrategy`. Diese soll die Basisklasse für verschiedene Aufzugstrategien sein. Damit die Strategie das Gebäude nicht selbst modifizieren kann, wird `Building` per `const` Pointer übergeben.

```
// Elevator strategy class: Determines to which floor the elevator should move next.
class ElevatorStrategy {
public:
   virtual ~ElevatorStrategy();
   virtual void createPlan(const Building*);    // create a plan for the simulation - the default
    implementation does nothing but saving the building pointer
   virtual int nextFloor() = 0;  // get the next floor to visit
protected:
   const Building *building;     // pointer to current building, set by createPlan()
};
```

Aufgabe 3.3 Eine einfache Aufzusstrategie

Implementiere eine einfache Aufzugstrategie, indem du eine neue Klasse erzeugst die von `ElevatorStrategy` erbt. Diese soll folgendermaßen vorgehen: Falls der Aufzug momentan leer ist, soll zum tiefsten Stockwerk gefahren werden, wo sich noch Personen befinden. Falls der Aufzug nicht leer ist, wird das Zielstockwerk einer der Personen im Aufzug ausgewählt.

Aufgabe 3.4 Implementation von `runSimulation`

Ändere nun `runSimulation()` entsprechend um, sodass die Simulation anhand der gegebenen Strategie durchgeführt wird. Folgender Pseudocode kann dir als Denkhilfe dienen:

while *People in Building or Elevator* **do**
  Calculate next floor;
  Move Elevator to next floor;
  Let all arrived people off;
  Let all people on floor into Elevator;
end

Teste die einfache Aufzugstrategie

Aufgabe 3.5 Neue Aufzugstrategien (optional)

Entwickle eine eigene Aufzugstrategien, indem du erneut eine neue Klasse erzeugst die von `ElevatorStrategy` erbt. Versuche, verschiedene Größen zu optimieren, wie z.B. die Anzahl der Stopps oder die verbrauchte Energie. Hierfür könnte Backtracking verwenden[1], eine einfache Methode, um eine optimale Lösungen durch Ausprobieren zu finden. Beachte, dass der Aufzug auch kopiert werden kann, um verschiedene Strategien zu testen.

[1] Siehe `http://de.wikipedia.org/wiki/Backtracking`