Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 3. Tag

Aufgabe 1 Vererbung und Polymorphie

a) Schreiben Sie eine Klasse *Person*. *Person* soll ein *protected* Attribut *name* vom Typ *std::string* haben (**#include <string>**), welches den Namen speichert. Initialisieren Sie den Namen im Konstruktor von *Person* und schreiben Sie auch einen Destruktor. Implementieren Sie außerdem die Methode

```
std::string getInfo() const;
```

um Informationen über die Person, in unserem Fall den Namen, abzurufen. Hinweis: Um ein String-Literal an eine *std::string* Variable anzuhängen, müssen Sie aus dem String-Literal zuerst ein *std::string*-Objekt machen. Beispiel:

```
std::string text = std::string("Name: ") + name;
```

b) Schreiben Sie eine Klasse *Student*, die von *Person* erbt und eine Person mit einer Matrikelnummer (ebenfalls *std::string*) modelliert. Rufen Sie in der Initialisierungsliste den entsprechenden Konstruktor der Elternklasse *Person* mittels **Person(name)** auf. Implementieren Sie auch einen Destruktor.

Überschreiben Sie die Methode *getInfo()*, sodass zusätzlich zum Namen auch die Matrikelnummer zurückgegeben wird. Sie können bei Bedarf die *getInfo()*-Implementierung der Elternklasse *Person* von *Student* aus mittels **Person::getInfo()** aufrufen.

c) Erstellen Sie nun in der *main()* eine je eine Person und einen Studenten und geben Sie deren Daten auf der Konsole aus. Vergewissern Sie sich, dass bei *Student* auch die Matrikelnummer ausgegeben wird. Schauen Sie sich auch die Ausgaben der Konstruktoren und Destruktoren an, und versuchen Sie, diese nachzuvollziehen.

d) Implementieren Sie nun die Funktion

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

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

Testen Sie Ihre Implementierung. Rufen Sie dazu *printPersonInfo()* sowohl mit beiden Personentypen auf.

e) Sie werden merken, 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. Deklarieren Sie daher *getInfo()* als **virtual**.

Testen Sie Ihre Implementierung erneut und vergewissern Sie sich, dass nun immer die richtige Methode aufgerufen wird.

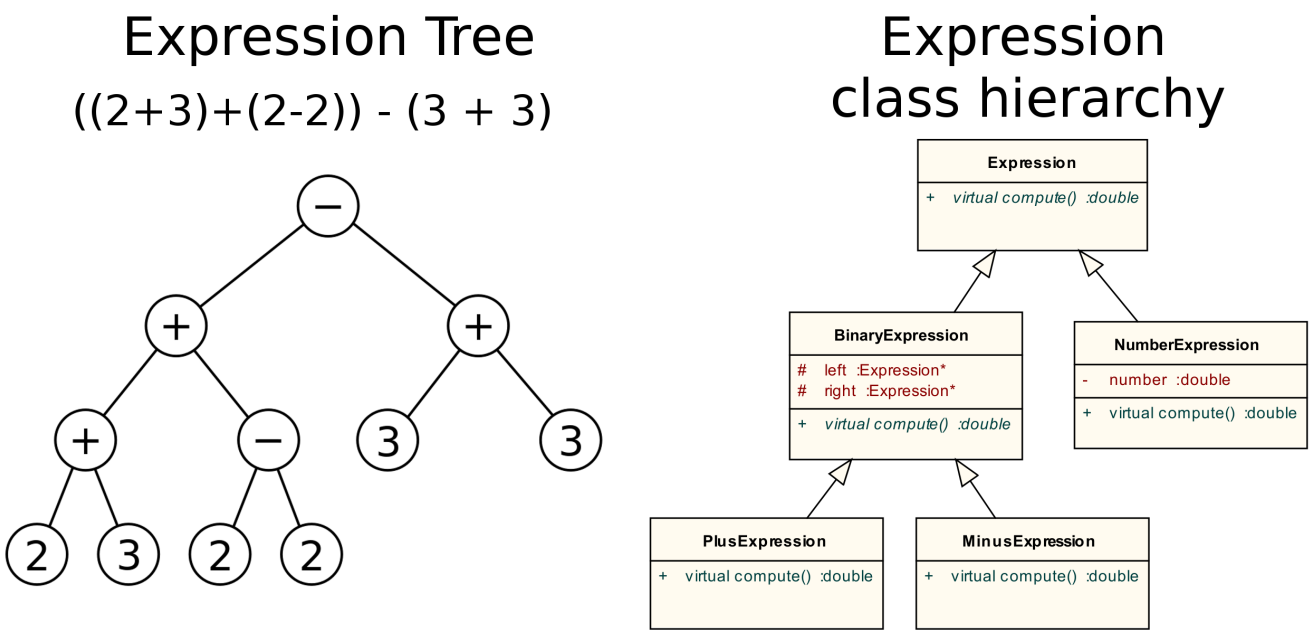
f) Legen Sie einen Studenten dynamisch auf dem Heap an, speichern Sie die Adresse jedoch in einem Zeiger auf eine *Person*. Löschen Sie die Person anschließend.

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

Analysieren Sie die Konsolenausgabe. Wie Sie sehen, 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. Deklarieren Sie deshalb in beiden Klassen den Destruktor als **virtual** und testen Sie die Korrektheit der Destruktoraufrufe.

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.



a) Schreiben Sie die abstrakte Klasse *Expression*. Diese soll als Basisklasse für alle Ausdrücke dienen. Implementieren Sie 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. Deklarieren Sie außerdem eine abstrakte (pure virtual) Methode **double compute()**, die das Ergebnis des Ausdrucks berechnen und zurückgeben soll.

b) Schreiben Sie die Klasse *NumberExpression*, die ein Blatt mit einer Zahl darstellt. Dementsprechend soll *NumberExpression* von *Expression* erben und ein Attribut zum Speichern einer Zahl besitzen, der im Konstruktor initialisiert wird. Implementieren Sie den Konstruktor und virtuellen Destruktor und versehen Sie auch diese mit einer Konsolenausgabe. Implementieren Sie die Methode *compute()*, die die gespeicherte Zahl zurückgibt.

c) Schreiben Sie die abstrakte Klasse *BinaryExpression* mit den protected Attributen *Expression* left, *right*. Implementieren Sie den Konstruktor und virtuellen Destruktor mit entsprechender Ausgabe. Vergessen Sie nicht, im Destruktor die beiden Zweige zu löschen.

d) Schreiben Sie die Klassen *PlusExpression* und *MinusExpression*, die von *BinaryExpression* erben und eine Addition bzw. Subtraktion realisieren. Implementieren Sie die Kon- und Destruktoren sowie die *compute()* Methode.

e) Testen Sie Ihre Implementierung. Schauen Sie sich die Ausgabe genau an und versuchen Sie 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.

a) Lagern Sie 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. Überlegen Sie sich, auf welche Art das Gebäude idealerweise übergeben werden sollte. Testen Sie Ihre Implementierung.

b) Implementieren Sie die Klasse *ElevatorStrategy*. Diese soll die Basisklasse für verschiedene Aufzugstrategien sein.

```
/**
 * Elevator strategy base class. Used to determine at which floor the elevator should move next.
 */
class ElevatorStrategy {
public:
    virtual ~ElevatorStrategy();

    /**
     * Creates a plan for the simulation.
     * Default implementation does nothing but saving the building pointer.
     */
    virtual void createPlan(const Building*);

    /**
     * get next floor to visit.
     */
    virtual int nextFloor() = 0;

protected:
    /** Pointer to current building, set by createPlan() */
    const Building* building;
};
```

Damit die Strategie das Gebäude nicht selbst modifizieren kann, wird *Building* per const Pointer übergeben.

c) Implementieren Sie eine einfache Aufzugstrategie. 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 eines der Personen im Aufzug ausgewählt.

d) Ändern Sie nun *runSimulation()* entsprechend um, sodass die Simulation anhand der gegebenen Strategie durchführt wird. Folgender Pseudocode kann Ihnen 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
```

e) Testen Sie Ihre Implementierung mit der bisher erstellten, einfachen Strategie.

f) Entwickeln Sie eigene Aufzugstrategien. Versuchen Sie, verschiedene Größen zu Optimieren, wie z.B. die Anzahl der Stopps oder die verbrauchte Energie. Hierfür können Sie Backtracking verwenden (siehe Wikipedia), eine einfache Methode, um optimale Lösungen durch Ausprobieren zu finden. Beachten Sie, dass der Aufzug auch kopiert werden kann, um verschiedene Strategien zu testen.