Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 3. Tag

Aufgabe 1 Vererbung und Polymorphie

In dieser Aufgabe sollst du Konzepte der Vererbung und Polymorphie unter Verwendung abstraker Funktionen erlernen.

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. Es ist aber hilfreich, auch dort der Lesbarkeit halber das Schlüsselwort zu verwenden.

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.

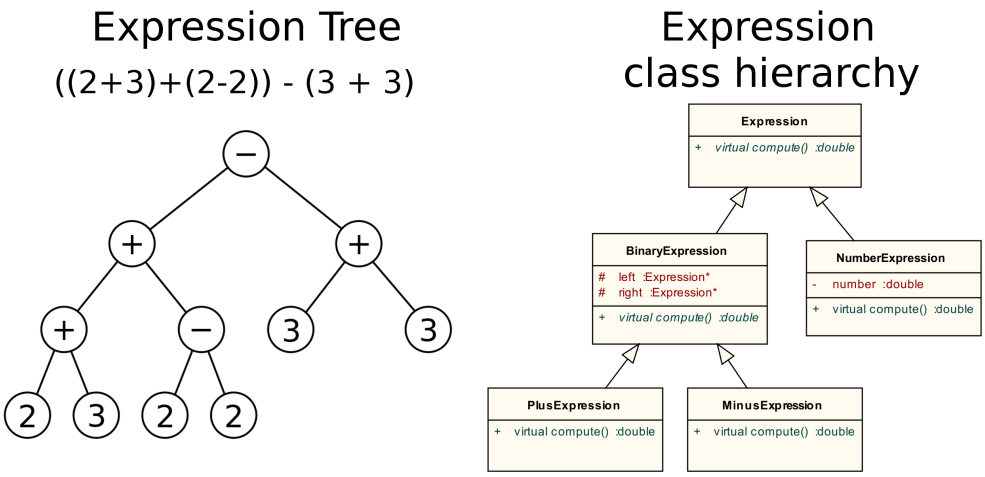

Abbildung 1: Abbildung: Beispielausdruck mit Ausdrucksbaum und Klassenhierarchie

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 Mehrfachvererbung

Verwende den Code der Aufgabe Aufgabe 1 als Basis.

Aufgabe 3.1 Klasse `Employee`

Schreibe die Klasse `Employee`, die einen Mitarbeiter darstellt. `Employee` soll von `Person` erben und den Namen seines Vorgesetzten als Attribut beinhalten. Erweitere auch entsprechend die Methode `getInfo()`.

Aufgabe 3.2 Klasse `StudentAssistant`

Schreibe nun eine Klasse `StudentAssistant`, die eine wissenschaftliche Hilfskraft modelliert. Eine wissenschaftliche Hilfskraft ist ein Student und gleichzeitig auch ein Mitarbeiter. Dementsprechend soll `StudentAssistant` sowohl von `Student` als auch von `Employee` erben. Das heißt es werden je ein Student- und ein Employee-Objekt im Konstruktor initialisiert. Weitere Attribute sind nicht nötig. Überschreibe `getInfo()`, um alle Daten auszugeben. Ändere dazu die Sichtbarkeit der Attribute sowohl von `Student` als auch von `Employee` von `private` auf `protected`.
Du wirst feststellen, dass sich die Klasse nicht kompilieren lässt, falls du das Attribut `name` direkt verwendest, da in einer `StudentAssistant`-Instanz zwei Instanzen von `Person` vorhanden sind - je eine von jeder Elternklasse. Deshalb müsse mittels dem Scope-Operator :: angeben, welche Basis du genau meinst.

```
Employee::name
// or
Student::name
```

Teste deine Implementation, indem du das Ergebnis von `getInfo()` direkt in der `main` ausgibst.

Aufgabe 3.3 Virtuelle Vererbung

Versuche nun, `printPersonInfo()` mit einer Instanz von `StudentAssistant` aufzurufen. Auch hier wird der Compiler mit einer Fehlermeldung abbrechen, da er nicht weiß, welche der beiden Basisklassen er nehmen soll. Diesmal ist es in C++ allerdings nicht mehr möglich, die Basisklasse zu spezifizieren, weshalb wir anders vorgehen werden. Wir sorgen mittels virtueller Vererbung dafür, dass `Person` nur ein Mal in `StudentAssistant` vorhanden ist.
Lasse dazu `Student` und `Employee` virtuell von `Person` erben. Noch lässt sich das Programm nicht kompilieren, denn sowohl `Student` als auch `Employee` versuchen, einen Konstruktor von `Person` aufzurufen. Da `Person` aber nur ein einziges mal in `StudentAssistant` vorhanden ist, müsste der Konstruktor demnach zwei mal aufgerufen werden – einmal von `Student` und einmal von `Employee`. Dies würde jedoch grob gegen die Sprachprinzipien verstoßen. Deshalb wird der Konstruktor von `Person` weder von `Student` noch von `Employee` aufgerufen! Stattdessen müssen wir in der Initialisierungsliste von `StudentAssistant` angeben, welcher Konstruktor von `Person` aufgerufen werden soll. Die Konstruktoraufrufe innerhalb von `Student` und `Employee` laufen stattdessen ins Leere, auch wenn sie syntaktisch vorhanden sind! Füge deshalb ein `Person(name)` in die Initialisierungsliste von `StudentAssistant` hinzu.
Teste deine Implementation. Versuche auch Folgendes: Ändere die Namen in den Konstruktoraufrufen von `Student` und `Employee` in der Initialisierungsliste von `StudentAssistant` und beobachte die Ausgabe. Mache dir dadurch klar, welche Probleme Mehrfachvererbung von implementierten Klassen verursachen kann!

Aufgabe 3.4 Erklärung

Eine Alternative zur Implementationsvererbung stellt **Schnittstellenvererbung** dar, wie es in Java üblich ist. Dabei werden Schnittstellen (Klassen mit ausschließlich abstrakten Methoden und ohne Attribute) definiert und nur diese vererbt. Zusätzlich gibt es Implementationen von diesen Schnittstellen. Man würde also `Person`, `Student`, `Employee` und `StudentAssistant` in jeweils zwei Klassen aufteilen, eine Schnittstelle und eine Implementation. Die Schnittstellen würden voneinander erben, z.B. `StudentBase` von `PersonBase`, und entsprechende pur virtuelle/abstrakte Methoden wie `virtual std::string StudentBase::GetStudentID() = 0` bereitstellen. Die Implementation würde ausschließlich von der jeweiligen Schnittstelle erben (`Student` von `StudentBase`). Diese Variante erscheint zwar aufwändiger als Implementationsvererbung, vermeidet aber viele der dabei entstehenden Probleme. Schnittstellenvererbung kann in Java eingesetzt werden, um Mehrfachvererbung zu realisieren.

Übung zum C/C++-Praktikum - Tag 3

Aufgabe 4 Exceptions

Ähnlich wie in Java können Fehler in C++ mittels Exceptions signalisiert werden.

```
try {
   ...
   throw <Type>;
} catch(<Type1> <param name>) {
   ...
} catch(<Type2> <param name>) {
   ...
}
...
```

Es gibt jedoch einige Unterschiede zur Fehlerbehandlung in Java. Das aus Java bekannte `finally`-Konstrukt existiert in C++ nicht. Außerdem kann jede Art von Wert geworfen werden – sowohl Objekte als auch primitive Werte wie z.B. `int`. In der Praxis wird es jedoch empfohlen, den geworfenen Wert von `std::exception` abzuleiten oder eine der existierenden Klassen aus der Standardbibliothek zu nutzen.
Im Gegensatz zu Java kann man Objekte nicht nur *by-Reference* sondern auch *by-Value* werfen und fangen. In diesem Fall wird das geworfene Objekt nach der Behandlung im `catch`-Block automatisch zerstört. Wenn es *by-Value* gefangen wird, wird das geworfene Objekt kopiert, ähnlich wie bei einem Funktionsaufruf. Beispiel:

```
// 1. Catch by value
try {
   throw C();     // create new object of class C and throw it
} catch(C c) {    // catch c by value => a copy of c is created when catching
   ...
}

// 2. Catch by reference
try {
   throw C();           // create new object of class C and throw it
} catch(const C &c) {   // catch c by reference, no copy is created
   ...
}
```

In der Praxis hat es sich durchgesetzt, *by-Value* zu werfen und *by-const-Reference* zu fangen.

Aufgabe 4.1

Erstelle eine Klasse `C` und implementiere einen Konstruktor, einen Copy-Konstruktor und einen Destruktor. Versehe diese mit Ausgaben auf der Konsole, so dass der Lebenszyklus während der Ausführung ersichtlich wird.

Aufgabe 4.2

Experimentiere mit Exceptions. Probiere insbesondere die beiden o.g. Fälle aus und beobachte die Ausgabe. Wann wird ein Objekt erstellt/kopiert/gelöscht? Teste auch, was passiert, wenn du mehrere `catch`-Blöcke erstellst und sich diese nur in der Übergabe unterscheiden (Wert/Referenz). Welcher von ihnen wird aufgerufen? Spielt die Reihenfolge eine Rolle?

Aufgabe 4.3

Füge der Klasse `List` vom Vortag Bereichsprüfungen hinzu. Schreibe die Methoden `insertElementAt()`, `getNthElement()` und `deleteAt()` so um, dass eine Exception geworfen wird, falls der angegebene Index die Größe der Liste überschreitet. Nimm dafür die Klasse `std::out_of_range` aus dem `stdexcept` Header.

Aufgabe 4.4

Teste deine Implementation. Provoziere eine Exception, indem du falsche Indices angibst, und fange die Exception als `const` Referenz ab. Du kannst die Methode `what()` benutzen, um an den Nachrichtentext der Exception zu gelangen.

Übung zum C/C++-Praktikum - Tag 3

Aufgabe 5 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 5.1 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 5.2 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 5.3 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 5.4 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 5.5 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`