

Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Objektorientierung in C++

Aufgabe 11 [0] Vererbung und Polymorphie

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/inheritance_polymorphism`. In dieser Aufgabe sollst du Konzepte der Vererbung und Polymorphie unter Verwendung abstrakter Funktionen erlernen.

Aufgabe 11.1 Klasse `Person`

Implementiere eine Klasse `Person`, die eine Person mit einem Namen darstellt. Füge allen Konstruktoren und Destruktoren 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 11.2 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 Destruktoren 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; // Person::getInfo() - get name and studentID  
private:  
    std::string studentID; // the student ID of the student  
};
```

Aufgaben zur Objektorientierung in C++

Hinweise

- Du kannst bei Bedarf die `getInfo()`-Implementation der Elternklasse `Person` von `Student` aus mittels `Person::getInfo()` aufrufen.

Aufgabe 11.3 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 Zeiger übergeben wird, können auch Unterklassen von `Person`, wie z.B. `Student`, übergeben werden.

Aufgabe 11.4 Dynamic Dispatch bei `printPersonInfo`

Du merkst, dass `printPersonInfo()` unabhängig von übergebenem Typ einer `Person` 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. Dekлариere 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.
- 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).

Aufgabe 11.5 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. Dekлариere 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.

Aufgabe 12 [0] Pure-Virtual-Methoden

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/expression_tree`. 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.

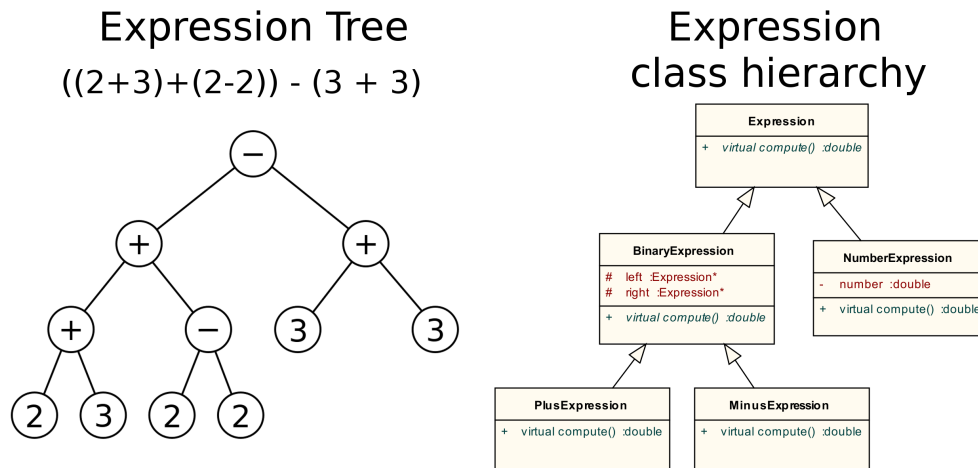


Abbildung 1: Beispielausdruck mit Ausdrucksbaum und Klassenhierarchie

- 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. Dekлариere außerdem eine abstrakte (pure `virtual`) Methode `virtual double compute() = 0;`, die das Ergebnis des Ausdrucks berechnen und zurückgeben soll.

Hinweise

- Anders als in Java muss die Klasse nicht explizit als `abstract` gekennzeichnet werden - es reicht, wenn sie mindestens eine *pure virtual* Methode enthält.
- 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.
- d) **Klassen `PlusExpression` 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) **Testlauf:** 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.

Aufgaben zur Objektorientierung in C++

Aufgabe 13 [0] Mehrfachvererbung

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/multiple_inheritance`. Verwende den Code der Aufgabe 11 als Basis.

Aufgabe 13.1 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 13.2 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 musst du mittels dem Scope-Operator `::` angeben, welche Klasse du genau meinst.

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

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

Aufgabe 13.3 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 13.4 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

Aufgaben zur Objektorientierung in C++

StudentAssistant in jeweils zwei Klassen aufteilen, eine Schnittstelle und eine Implementation. Die Schnittstellen würden voneinander erben, z.B. StudentBase von PersonBase, und entsprechende pure 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.



Dieses Werk ist unter einer Creative Commons Lizenz vom Typ Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International zugänglich. Um eine Kopie dieser Lizenz einzusehen, konsultieren Sie <http://creativecommons.org/licenses/by-nc-nd/4.0/> oder wenden Sie sich brieflich an Creative Commons, Postfach 1866, Mountain View, California, 94042 USA.