h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK
O. SKROCH

Lesen Sie die Angaben in Ruhe, bearbeiten Sie die Aufgaben vollständig, machen Sie für die Testate aber nicht mehr. Legen Sie Wert auf Qualität.

Aufgabe 1

a) Bringen Sie die `Myvector` Containerklasse aus der Vorlesung zum Laufen. Erstellen Sie eine Headerdatei `myvector.h` mit allen Deklarationen und eine Datei `myvector.cpp` mit allen Definitionen. Passen Sie sämtliche Methoden wie erforderlich an, so dass die dynamische Reallokation des gekapselten `double`-Datenfelds und damit auch `Myvector::push_back()` korrekt funktioniert. Probieren Sie Ihren Typ in `main()` gründlich aus. Legen Sie `Myvector` Objekte an, testen Sie *alle* verfügbaren Methoden aus, speziell auch alle Konstruktoren.

b) Bringen Sie die `SiLL` Stapelklasse aus der Vorlesung in der Version mit dem `SillNode` Typ zum Laufen. Erstellen Sie eine Headerdatei `sill.h` mit allen Deklarationen und eine Datei `sill.cpp` mit allen Definitionen. Programmieren Sie ein einfaches Menü zur interaktiven Auswahl von "push" (ablegen), "pop" (entfernen) und "print" (alles informativ ausgeben). Probieren Sie alles gründlich aus.
Hinweis: Sie finden im vorlesungsbegleitenden Quellcode eine Version noch ohne den `SillNode` Typ, auf der Sie aufbauen können…

Aufgabe 2

Implementieren Sie vollständig die in der Vorlesung noch nicht definierten Methoden der Klassen `Kunde` und `Auftrag` unter Berücksichtigung der Assoziation zwischen den beiden Klassen. Ihre Implementierung soll Codezeilen wie z.B. die folgenden lauffähig machen:

```
// Anlegen: zwei Kunden und drei Auftraege
Kunde* k1 { new Kunde { "Sepplhuber-Finsterwalder" } };
Kunde k2 { "Kurz" };
Auftrag* a1 { new Auftrag { "Decke streichen" } };
Auftrag* a2 { new Auftrag { "Wand tapezieren" } };
Auftrag a3 { "Jalousie montieren" };
Auftrag a4 { "Laminat verlegen" };
// Erste Assoziationsrichtung:
// aus einem Objekt vom Typ Kunde
// zu Objekten vom Typ Auftrag
k1->add_auftrag( a1 );
k1->add_auftrag( a2 );
k1->add_auftrag( &a3 );

// Zweite Assoziationsrichtung:
// aus einem Objekt vom Typ Auftrag
// zu einem Objekt vom Typ Kunde
a4.setKunde( &k2 );
// Alles ausgeben:
k1->print();
k2.print();
a1->print();
a2->print();
```

```
a3.print();
a4.print();
// Speicher zurueckgeben:
delete k1; k1 = nullptr;
delete a1; a1 = nullptr;
delete a2; a2 = nullptr;
```

Berücksichtigen Sie, dass `add_auftrag()` auch den Kunden im Auftragsobjekt setzen muss, und dass `set_kunde()` auch den Auftrag ins Kundenobjekt eintragen muss.

Schreiben Sie eine `run_test()` Testfunktion mit mindestens zehn Testfällen, so dass die von Ihnen definierten Methoden an kritischen Stellen des Programms (überlegen Sie: welche sind das?) überprüft werden.

Aufgabe 3

Implementieren Sie, auf Basis der in der Vorlesung vorgestellten Quellcodefragmente, die einfache Generalisierung mit `Employee` als Basisklasse und `Manager` als Klasse, die direkt und public aus `Employee` abgeleitet ist.

Der `Employee` Typ kapselt mindestens `string name`, `int department`, `double salary` und bietet als Operationen mindestens den Standardkonstruktor, einen weiteren Konstruktor und eine virtuelle `print()` Methode zur Ausgabe.

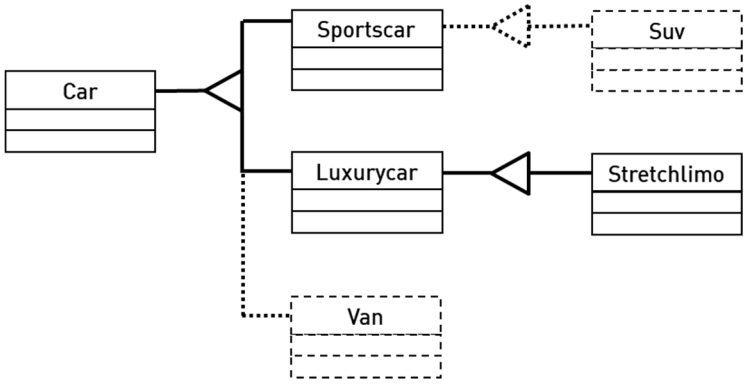
Der `Manager` Typ kapselt mindestens `int level`, `Employee* deputy` und bietet als Operationen mindestens den Standardkonstruktor, einen weiteren Konstruktor und eine `print()` Methode zur Ausgabe.

Stellen Sie sicher, dass eine einfach verkettete Liste aufgebaut wird, die alle erzeugten Objekte (vom Typ `Employee` und vom Typ `Manager`) enthält.

Probieren Sie Ihre Klassen über einen `main()` Treiber aus: legen Sie einige `Employee` Objekte und `Manager` Objekte an, geben Sie in einer Schleife über einen `Employee*` alle Objektdetails aus.

Aufgabe 4

Im vorlesungsbegleitenden Quellcode finden Sie eine rudimentäre Implementierung für die nicht-gestrichelt dargestellten Teile der folgenden Klassenhierarchie:



a) Bringen Sie den Quellcode zum Laufen, verwenden Sie dafür genau drei eigene Dateien: `car.h` mit allen Deklarationen, `car.cpp` mit den Definitionen aus dem Header, und `car_main.cpp` mit der `main()` Funktion zum Ausprobieren.

b) Erweitern Sie das Programm wie oben dargestellt um

```
class Van : public Car
class Suv : public Sportscar.
```

Denken Sie sich je zwei Attribute für die neuen Typen aus, und sorgen Sie dafür, dass v.a. `Van::print()` und `Suv::print()` komplett sind und alle Informationen ausgeben.

c) Probieren Sie alles gründlich aus, experimentieren Sie vor allem mit dem Mechanismus für virtuelle Methodenaufrufe. Bauen Sie auch auf dem vorlesungsbegleitenden Quellcode und den teilwiese auskommentierten Codezeilen auf.

Aufgabe 5 (optional)

a) Ändern und erweitern Sie die `SiLL`-Klasse aus der Vorlesung zu einer `DoLL`-Klasse (doubly-linked list) so, dass die Implementierung eine *doppelt* verkettete dynamische Liste von `int`-Werten ist. In einer doppelt verketteten Liste enthält jeder Knoten zusätzlich zu dem Zeiger auf seinen direkten Nachfolger auch einen Zeiger auf seinen direkten Vorgänger. Der direkte Vorgänger des obersten Knotens ist, genau wie der direkte Nachfolger des untersten Knotens, der Nullzeiger. Für eine solche Datenstruktur können effizient Methoden programmiert werden, die an jeder Position einfach ein Knoten einfügen oder entfernen, da sich ein „Operationszeiger" in jedem Knoten eines `DoLL`-Objekts effizient in beide Richtungen „um eins weiter bewegen" kann.

b) Implementieren Sie an der Schnittstelle der Klasse zumindest die Methoden print() zur Ausgabe der Liste und `insert()` / `remove()` zum Einfügen / Entfernen an einer beliebigen Stelle. Durch Einsatz von `insert()` / `remove()` lassen sich dann `push_back()` / `pop_back()` zum Einfügen oder Entfernen am Ende (nach dem letzten Element), sowie `push_front()` / `pop_front()` zum Einfügen oder Entfernen am Anfang (vor dem ersten Element) programmieren. Dann kann man die Elemente neben dem LIFO-Prinzip (last in first out, Stapel) auch nach dem FIFO-Prinzip (first in first out, Warteschlange) verwalten.

c) Implementieren Sie ein einfaches, interaktives „Menü" in `main()`, über welches der Benutzer per Tastaureingaben in der Liste zufällig erzeugte `int`-Werte am Anfang oder am Ende einfügen und entfernen kann und sich die komplette Liste ausgeben lassen kann.