Übungen zur Vorlesung

"Objektorientiertes Programmieren im Wissenschaftlichen Rechnen"

Dr. Olaf Ippisch, Ole Klein

Abgabe am 20. 05. 2014 in der Vorlesung

Übung 1 Leere-Basisklassen-Optimierung

In dieser Aufgabe wollen wir untersuchen, wieviel Platz leere C++-Klassen und deren abgeleitete Klassen belegen.

Entwerfen Sie hierzu eine Klasse `Empty`, die keine Attribute besitzt. Leiten Sie hiervon eine Klasse `EmptyDerived` ab, die ebenfalls keine Attribute hat. Entwerfen sie auch eine Klasse `NonEmpty`, die von `Empty` abgeleitet ist, aber einen `char` als Attribute besitzt.

Welche Größe hat ein Objekt der Klassen `Empty`, `EmptyDerived` und `NonEmpty` laut dem **`sizeof`** Operator? (BTW dieses Verhalten wird Leere-Basisklasse-Optimierung genannt.)

Um zu verstehen, warum die Größe von leeren Klassen (laut Standard) so ist wie beobachtet, betrachten Sie folgende Klasse

```
struct Composite{
  Empty a;
  int  b;
};
```

Was würde passieren, wenn `Empty` keinen Speicherplatz beanspruchen würde? Sei `c` ein Objekt der Klasse `Composite`, welche Adressen hätten in diesem Fall die Attribute `c.a` und `c.b`?

Bestimmen Sie die Größe von Objekten des Klassentyps `Composite`! Wie ändert sich diese, wenn Sie den Typ des Attributes `b` in `char` umwandeln? (Ausprobieren!) *6 Punkte*

Übung 2 Vererbung und Komposition

Sie sollen ein Klasse `NumVector` schreiben, die einen numerischen Vektor repräsentiert. Die Größe des Vektors muss unveränderbar sein. Auf einzelne Einträge des Vektors soll mittels des eckige-Klammer-Operators Lese- und Schreibzugriff gewährt werden. Zusätzlich soll die Klasse die Methode **`double`** `norm()` besitzen, die die euklidische Norm des Vektors berechnet. Weitere Methoden sollen nicht sichtbar sein. Insbesondere darf ein `NumVector` kein `std::vector<double>` sein!

Entwerfen Sie zwei Implementierungen dieser Klasse, die beide die Werte in einem `std::vector<double>` speichern. Die erste Implementierung soll von `std::vector<double>` abgeleitet sein und die zweite ohne Vererbung auskommen.

Testen Sie Ihre Implementierung wie folgt. Die Klassen sollen mit dem folgendem **`main`**-Programm korrekt funktionieren.

```
#include<vector>
#include<iostream>
#include<cmath>
int main(){
  NumVector v(3);
  v[0]=1; v[1]=3, v[2]=4;
  const NumVector& w=v;
```

```
  std::cout<<"norm is "<<w.norm()<<std::endl;
  std::cout<<"vector is ["<<w[0]<<", "<<w[1]<<", "<<w[2]<<"]"<<std::endl;
}
```

Das nachfolgende Programm darf allerdings nicht kompilieren!

```
#include<vector>
#include<iostream>
#include<cmath>

void vectorTest(std::vector<double>& v){}

int main(){
  NumVector v(3);
  v.resize(2); // Darf wie andere std::vector Funktionen nicht
               //  sichtbar sein!
  vectorTest(v); // Hier muss auch ein Compiler Fehler auftreten!
}
```

10 Punkte

ÜBUNG 3 VERERBUNG UND FUNKTIONEN

Finden Sie die Fehler in nachfolgendem Program. Welche Ausgaben erzeugt es nach Entfernung der fehlerhaften Zeilen und warum?

```
1  #include<iostream>
2
3  class A{
4  public:
5    void foo() const { std::cout<<"A::foo"<<std::endl; }
6  };
7
8  class B : public A{
9  public:
10   void foo() { std::cout<<"B::foo"<<std::endl; }
11 };
12
13 class C : private B{
14 public:
15   void bar() { foo();}
16 };
17
18 void test(const A& a){ a.foo(); }
19
20 int main(){
21   A a; B b; C c;
22   a.foo();
23   b.foo();
24   test(b);
25   c.bar();
26   test(c);
27 }
```

4 Punkte