Übungen zur Vorlesung

"Objektorientiertes Programmieren im Wissenschaftlichen Rechnen"

Dr. Olaf Ippisch, Ole Klein

Abgabe am 27. 05. 2014 in der Vorlesung

Übung 1 Exceptions und Destruktoren

Was passiert, wenn in einem Destruktor eine Exception geworfen wird? Betrachten wir nebenstehendes Programm.

```
1  #include <iostream>
2  #include <string>
3  #include <exception>
4
5  // class Foo throws in the destructor
6  class Foo {
7  public:
8      ~Foo () {
9          throw my_exception("Foo exception");
10     }
11 };
12
13 // class Bar throws in the constructor
14 class Bar {
15 public:
16     Bar () {
17         throw my_exception("Bar exception");
18     }
19 };
20
21 int main()
22 try {
23     Foo f;
24     Bar b;
25 }
26 catch (const std::exception & e){
27     std::cout << "ERROR:" << e.what() << std::endl;
28 }
```

- Ergänzen Sie das nachfolgende Beispiel um die Definition der Klasse `my_exception`, welche von `std::exception` abgeleitet ist. Die Klasse `my_exception` soll in ihrem Konstruktor eine Fehlermeldung als `std::string` übergeben bekommen. Von der virtuellen Methode `what()` soll diese Fehlermeldung dann zurückgegeben werden.

- Was beobachten Sie beim Ausführen des Programms?

- Versuchen Sie, das Verhalten zu erklären.

- Der C++-Standard schreibt in Sektion 15.2, Punkt 3:

  > 3 The process of calling destructors for automatic objects constructed on the path from a try block to a throw-expression is called "stack *unwinding*." [Note: If a destructor called during stack unwinding exits with an exception, terminate is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor. —end note]

  Warum ist das unter *Note* beschriebene Verhalten sinnvoll?

Weiterlesen: Eine schöne Zusammenstellung der verschiedenen Optionen im Zusammenspiel von Destruktoren und Exceptions finden Sie unter `http://www.kolpackov.net/projects/c++/eh/dtor-1.xhtml`

8 Punkte

ÜBUNG 2 EXCEPTIONS UND SANITY CHECKS

Die auf dem letzten Blatt behandelte Klasse `NumVector` kam bisher ohne Bounds Checking aus, das heisst man konnte auf Indizes kleiner Null oder jenseits des größten Index zugreifen. Modifizieren Sie die Methode `operator[]` so, dass sie für fehlerhafte Zugriffe eine Exception wirft (vergleichbar mit der Methode `std::vector<T>::at` statt `std::vector<T>::operator[]`).

Implementieren Sie außerdem eine Methode `operator*`, die das Skalarprodukt zweier Vektoren berechnet. Dabei potentiell aufgrund inkompatibler Längen auftretende Exceptions sollten Sie abfangen und stattdessen eine neue, aussagekräftigere Exception werfen.

Die Exceptions sollen dabei jeweils von Ihnen erstellte Klassen sein, die als Token an die aufrufende Funktion übergeben werden. Schreiben Sie ein Programm, dass beide Exceptions testet und bei Fehlern eine informative Meldung ausgibt, ohne das Programm zu beenden. *8 Punkte*

ÜBUNG 3 FATHER AND SON

Was wird Ihrer Meinung nach der nebenstehende JAVA-Code tun? Übersetzen Sie ihn in äquivalenten C++-Code (`extends` entspricht dabei `public`-Vererbung und `Throwable` einer Exception).

```
CLASS BALL EXTENDS THROWABLE {}
CLASS P{
  P TARGET;
  P(P TARGET) {
    THIS.TARGET = TARGET;
  }
  VOID AIM(BALL BALL) {
    TRY {
      THROW BALL;
    }
    CATCH (BALL B){
      TARGET.AIM(B);
    }
  }
  PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
    P PARENT = NEW P(NULL);
    P CHILD = NEW P(PARENT);
    PARENT.TARGET = CHILD;
    PARENT.AIM(NEW BALL());
  }
}
```

Kommentieren Sie Ihr Programm und geben Sie an, in welcher Reihenfolge die Zeilen ausgeführt werden. Ergibt sich ein valides C++-Programm? Falls ja, was tut es?

Quelle: Randall Munroe (`xkcd.com`)

4 Punkte