Objektorientierte Programmierung
(Winter 2008/2009)

Kapitel 14: Exceptions (Ausnahmebehandlung)

- Motivation
- Throw und Catch

Motivation (1)

- Funktionen werden manchmal mit Eingabewerten oder in Situationen aufgerufen, bei denen sie ihre Aufgabe nicht erfüllen können.
- Z.B. `pop()` bei leerem Stack (siehe Kap. 13):

```
template<class T, int Max> T Stack<T,Max>::pop()
{
    if(num_elems > 0) // Stack ist nicht leer
        return elems[--num_elems];
    else {
        ???
    }
};
```


Motivation (2)

- Oben wurden bereits zwei typische (früher übliche) Behandlungsmöglichkeiten gezeigt:
  - ◇ Es wird Fehlerwert zurückgeliefert (`0`).
  - ◇ Das Programm wird mit einer Fehlermeldung beendet.
- Beide Möglichkeiten haben wesentliche Nachteile.
- Deswegen haben moderne Sprachen üblicherweise einen Exception-Mechanismus.
- Die wesentliche Idee ist dabei, die Fehlererkennung und die Fehlerbehandlung zu trennen.

Motivation (3)

- Probleme der Lösung mit Fehlerwert:
  - ◇ Templates benötigen einen zusätzlichen Parameter für den Fehlerwert (`0` geht nicht immer).
  - ◇ Eventuell möchte man die Zahl `0` auch als normalen Wert auf dem Stack speichern.
  - ◇ Diese Art der Fehlerbehandlung ist mühsam, weil bei jedem Aufruf abgefragt werden muß, ob er erfolgreich war (Programm wird deutlich länger).
  - ◇ Das kann leicht vergessen werden, dann wird mit einem falschen Wert weitergerechnet.

Motivation (4)

- Probleme bei Beendigung des Programms:
  - ◇ Die Bibliotheksfunktion weiß nicht, ob der Aufrufer noch etwas wichtiges zu tun hat.
  - ◇ Z.B. bei einem Editor / Spiel: Noch einmal abspeichern.

    Eventuell in eine temporäre Datei, weil die Daten im Puffer ja eventuell durch den Fehler beschädigt sein könnten, und man den letzten sicheren Stand nicht überschreiben will. Aber wenn der Benutzer lange mit dem Programm gearbeitet/gespielt hat, möchte er möglichst vermeiden, daß durch einen Programmfehler alles umsonst war.
  - ◇ Bei einem Kernkraftwerk: Anlage in einen sicheren Betriebszustand herunterfahren.

Lösung: Exceptions (1)

- Zunächst definiert man Klassen für die Exceptions (für jede Fehlerart eine eigene Klasse).

  Auf diese Weise können verschiedene Arten von Fehlern unterschieden werden. Natürlich gehen auch Strukturen (der Unterschied ist nur, daß der Inhalt `public` ist). Tatsächlich auch beliebige Typen (s.u.).
- In der Klasse können bei Bedarf weitere Informationen über den Fehler abgelegt werden

  Z.B. Eingabewerte des fehlerhaften Funktionsaufrufs.
- Es ist aber nicht verlangt, daß Exceptions mit Klassen identifiziert werden: Auch Zahlen oder Strings lassen sich verwenden.

Lösung: Exceptions (2)

- Im Beispiel gibt es keine zusätzlichen Angaben zur Beschreibung der Fehlersituation, daher definiert man eine leere Klasse (ohne Attribute):

```
class pop_empty_stack { };
```

- Durch die Lösung mit Fehlerklassen kann das Typsystem von C++ zur Unterscheidung der Fehler benutzt werden, inklusive auch der Vererbung (Klassifizierung von Fehlern).

  Man braucht auf diese Art keine neuen Schlüsselworte zur Definition verschiedener Exceptions.

Lösung: Exceptions (3)

- Wenn der Fehler erkannt ist, löst man mit `throw` eine Exeption aus.
- Z.B. `pop()` bei leerem Stack (siehe Kap. 13):

```
template<class T, int Max> T Stack<T,Max>::pop()
{
    if(num_elems > 0) // Stack ist nicht leer
        return elems[--num_elems];
    else {
        throw pop_empty_stack();
        return elems[0]; // unnötig.
    }
};
```

Lösung: Exceptions (4)

- In einem übergeordneten Programmteil behandelt man die Exception mit einem `try-catch`-Block:

```
int main()
{
    try {
        ... Normale Arbeit,
        ... ruft direkt oder indirekt pop() auf
    }
    catch(pop_empty_stack) {
        ... z.B. noch einmal abspeichern
    }
}
```


Lösung: Exceptions (5)

- `throw` führt einen nicht-lokalen Sprung aus:
  - ◇ `goto` ist nur innerhalb einer Prozedur möglich.
  - ◇ `throw` beendet alle Prozeduren bis zur in der Aufrufschachtelung nächsten Prozedur, bei der sich der Aufruf innerhalb eines `try`-Blockes mit passender `catch`-Klausel befunden hat.

    Falls die `throw`-Anweisung dagegen direkt innerhalb eines solchen `try`-Blockes steht, wird natürlich keine Prozedur beendet: Dann ist es ein normaler Sprung.

    Eine mögliche Implementierung dieses Konzeptes ist mit einem zusätzlichem Stack für Exceptions: Bei jedem betreten eines `try`-Blockes wird ein Eintrag auf den Stack gelegt, der auch den aktuellen Stackpointer des Prozeduraufruf-Stacks enthält.

Lösung: Exceptions (6)

- Nach einem `try`-Block kann man mehrere `catch`-Klauseln für verschiede Exceptions angeben:

```
try { ... }
catch(Exeption1) { ... }
catch(Exeption2) { ... }
```

- Am Ende eines `catch`-Blocks wird die Ausführung nach der Liste von `catch`-Blöcken fortgesetzt.

  Sofern der `catch`-Block nicht wieder `throw` ausführt, weil er die Exception nur partiell behandelt hat ("re-throwing an exception"). Natürlich kann man den `catch`-Block auch über `return` oder `exit()` etc. verlassen. Falls man all das aber nicht tut, landet man nicht wie beim `switch` im nächsten Fall, sondern nach allen `catch`-Blöcken.

Lösung: Exceptions (7)

- Wenn eine Exception nicht (mit `catch`) abgefangen wird, wird das Programm beendet.

  Mit einer für den Benutzer eventuell nicht besonders verständlichen Fehlermeldung. Man sollte diesen Fall besser vermeiden.

- Man kann aber mit `catch(...)` jede (nicht vorher durch eine andere `catch`-Klausel behandelte) Exception abfangen.
- Falls das Exeption-Objekt Daten enthält, kann man dafür eine Variable einführen:

```
catch(Exception x)
```

Noch einige Feinheiten (1)

- `throw E` ist eine Expression vom Typ `void`. Sie kann damit insbesondere in bedingten Ausdrücken `?:` eingesetzt werden.

  Es ist beim bedingten Ausdruck ok, wenn nur einer der beiden Fälle ein `throw` ist, dann ist der Gesamttyp des Ausdrucks der des anderen Falls. Man kann damit eine Exception auch tief im Innern eines Ausdrucks auslösen.

- Eine Exception zählt als behandelt, sobald der passende Exception-Handler `catch {...}` betreten wird.

  Es kann in C++ deswegen nie gleichzeitig mehr als eine unbehandelte Exception geben.

Noch einige Feinheiten (2)

- Im Exception-Handler kann man auch `throw;` (ohne Argument) verwenden, um die gleiche Exception weiterzugeben.

  Das temporäre Objekt, das die Exception beschreibt, wird in diesem Fall nicht freigegeben.

- Wenn die Prozeduren verlassen werden, die für eine Exception kein passendes `catch` enthalten, werden dort die Destruktoren für alle Objekte in lokalen Variablen ausgeführt (werden ja gelöscht).

  Falls einer der Destruktoren eine Exception auslöst, wird das Programm beendet (mit `terminate()`).

Achtung!

- Die klassischen Hardware-Alarme, wie sie z.B. durch eine Division durch 0 ausgelöst werden, haben mit dem C++ Exception-Mechanismus nichts zu tun.

  Obwohl man sie in der Informatik auch "Exceptions" nennt.

- Man kann daher eine Division durch 0 (und ähnliche Fehler) nicht einmal mit `catch(...)` abfangen.

  Obwohl auf ... sonst jede Exception passen würde. C++ Exceptions sind nur für "synchrone Ausnahmen". Eine asynchrone Unterbrechung, wie etwa, daß das nächste Zeichen von der Tastatur bereit steht, muß ganz anders behandelt werden. Bei modernen CPUs mit Pipelined Architektur soll auch die Division durch 0 asynchron gemeldet werden. Natürlich möchte man aber solche Fehler auch abfangen. Bei UNIX schaue man im Handbuch unter `signal()`.