Programmieren:

Exceptions

Inhaltsverzeichnis:

1. Was sind Exceptions?

- Wenn irgendwo eine Datei geöffnet wird, so ist damit zu rechnen, dass die Datei nicht vorhanden ist (falscher Name, falsches Directory). Das sollte eigentlich nicht vorkommen, hoffentlich selten sein, aber es ist damit zu rechnen: eine Ausnahme, Exception.
- Inhalt ist falsch (beim Lesen einer Datei
- Dateisystem ist voll (beim Schreiben in einer Datei)
- Gerät ist defekt, liefert ungültige Werte, liefert (vorübergehend) keine Daten.

Exceptions definieren Ausnahmefälle. Mit anderen Worten, man sagt dem Programm wie es sich in einer gewissen Fehlersituation zu verhalten hat.

2. Struktur des Exception Handling

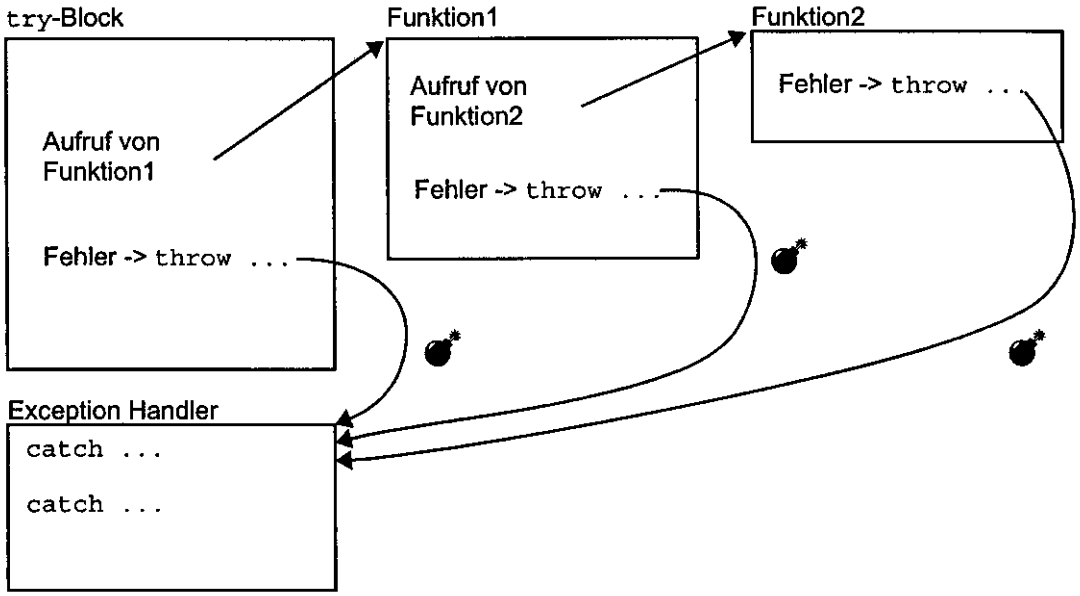
Die Ausnahmebehandlung hat folgende Struktur:

- alle Ausnahmen und Fehler eines ganzen Blocks von Code werden an einer Stelle, am Ende dieses Blocks, gemeinsam behandelt
- in dieser Fehlerbehandlung landen alle Fehler aus dem Code-Block, auch die Fehler aus aufgerufenen Funktionen einschliesslich Konstruktoren und überladenen Operatoren
- die Funktionsaufrufe können dabei beliebig tief geschachtelt sein
- bei der ersten Ausnahme wird zur Fehlerbehandlung gesprungen, der Rest des Code-Blocks wird nicht mehr abgearbeitet
- alle in dem Block einschliesslich der aufgerufenen Funktionen erzeugten Objekte werden vor dem Sprung zur Fehlerbehandlung durch Destruktoraufrufe vernichtet
- diese Blöcke können geschachtelt werden
- für gemeldete, aber vom Programm selbst nicht behandelte Fehlrer wird vom Compiler eine Standardbehandlung bereitgestellt

Folgende Begriffe und neue Sprachelemente werden verwendet:

- der Code-Block, dessen Ausnahmen gemeinsam behandelt werden, heisst `try`-Block, denn er wird „ausprobiert“
- die Fehler werden der Ausnahmebehandlung gemeldet, dieser „zugeworfen“ oder in diese „hineingeworfen“, die entsprechende Anweisung heisst daher `throw`
- die Ausnahmebehandlung geschieht durch Funktionen, die man Eception Handler (auch kurz Handler) nennt
- diese „fangen“ die „geworfenen“ Fehler, daher das Schlüsselwort catch

Die folgende Abbildung zeigt die Struktur der Ausnahmebehandlung:



3. Try-Block

Ein try-Block hat die Form:
`try`
Zusammengesetzte Anweisung (Block)
Handler-Liste

Also:

```
try {
     ...
}
catch(...) {...}    //1. Handler
catch(...) {...}    //2. Handler
```

In der zusammengesetzten Anweisung, also in dem von einer geschweiften Klammer eingeschlossenen Block, werden die Fehler entdeckt und geworfen. Beim Werfen des Fehlers wird ein Argument mitgegeben, dies kann ein Argument eines eingebauten Datentyps oder eines selbstdefinierten Datentyps (Objekt einer Klasse) sein.

Sobald ein Fehler geworfen wird, wird die weitere Abarbeitung der Anweisungen des `try`-Blocks (einschliesslich der Anweisungen der aufgerufenen Funktionen) abgebrochen. Alle lokalen Objekte werden vernichtet, sowohl die lokalen Objekte der aufgerufenen Funktionen als auch des `try`-Blocks selbst, und es wird zur Handler-Liste gesprungen (`catch`).

Es wird nur ein passender Handler aufgerufen, d.h. ein Handler, der einen passenden Datentyp fängt. Dabei sucht er in dieser Reihenfolge einen passenden Handler:

1. Handler mit exakt passendem Datentyp (ein const fängt dabei auch nicht-const)
2. Handler, dessen Datentyp eine Oberklasse des geworfenen Datentyps ist
3. bei Zeigern ein Zeiger auf den gleichen Datentyp oder den einer Oberklasse

Nicht passende Handler sind

- andere eingebaute Datentypen (d.h. int wird nicht als double gefangen)
- andere Datentypen in die konvertiert werden könnte (d.h. es wird keine Typwandlung gemacht)

Nach dem Aufruf des Handlers wird im Programm hinter dem try-Block, also hinter der Handlerliste, fortgefahren (natürlich nur sofern der Handler das Programm nicht abgebrochen hat).

Wie bereits erwähnt können try-Blöcke geschachtelt werden. Beim Aufruf einer Funktion aus einem try-Block kann die Funktion selbst wieder einen try-Block enthalten. Beim Werfen von Fehlern werden die try-Blöcke entlang dem Aufrufweg rückwärts verfolgt und der erste (also innnerste) passende Handler aufgerufen.

4. Exception Handler – catch

Ein Exception Handler hat die Form:

Catch(Formalparameter) Block

d.h. ein catch entspricht einer Funktionsdefinition mit einem Argument ohne Rückgabetyp. Der Parameter kann wie bei einer Funktion im Block verwendet werden.

Für das Fangen von Objekten empfiehlt sich wie bis anhin, das Objekt als Referenz zu verwenden, um ein Kopieren zu verhindern.

5. Default Exception Handler

Will man in einem try-Block alle Fehler fangen, aber nicht für jeden möglichen Typ von Formalparameter einen eigenen Exception Handler schreiben, so kann man einen Default Exception Handler definieren. Dieser fängt dann alle Fehler, für die kein anderer Exception Handler vorhanden ist.

Der Default Exception Hndler hat die Form.
`catch( ... )` Block

Beispiel:

```
catch(...) {
     cout << „Irgendein Fehler trat auf!“;
     exit(1); }
```

6. throw

Throw kommt in zwei Formen vor.

1. `throw` *Ausdruck*`;`
   Sie dient zum Werfen eines Fehlers. Ausdruck gibt den zu werfenden Wert. Dies kann ein Wert eines eingebauten Datentyps sein, es kann aber auch ein Objekt einer Klasse sein.
   throw string(„Speicherfehler“);
   Hier wird der Klasse string aufgerufen, damit wird ein Objekt dieser Klasse erzeugt und geworfen.
2. `throw;`
   Sie dient dazu, den Fehler aus einem Exception Handler in einen anderen weiterzuwerfen, d.h. in einen Exception Handler eines `try`-Blocks, der den aktuelllen try-Block umschliesst. Sie kann daher nur in einem Exception Handler verwendet werden.

Im folgenden Beispiel vom „`throw`-Typ 2“ wird das gefangene Objekt weitergeworfen. Dabei wird **keine** Kopie gemacht.

```
Try {
     ...
}
catch( Error& e) {
     ...
     throw;              //werfe den Fehler weiter
}
```

7. Exception Spezifikation

Die effektiven Exceptions sind nur im Quellcode geschrieben. Damit sie aber auch von aussen sichtbar werden kann man sie auch im Header-File bei der entsprechenden Funktion angeben.

Funktionsdeklaration mit Exception Speifikation:

Rückgabetyp Funktionsname`(` *Parameterliste* `) throw (` *Typliste* `);`

Bsp:

```
Vektor operator+( const Vektor& v) const throw (Error);
```

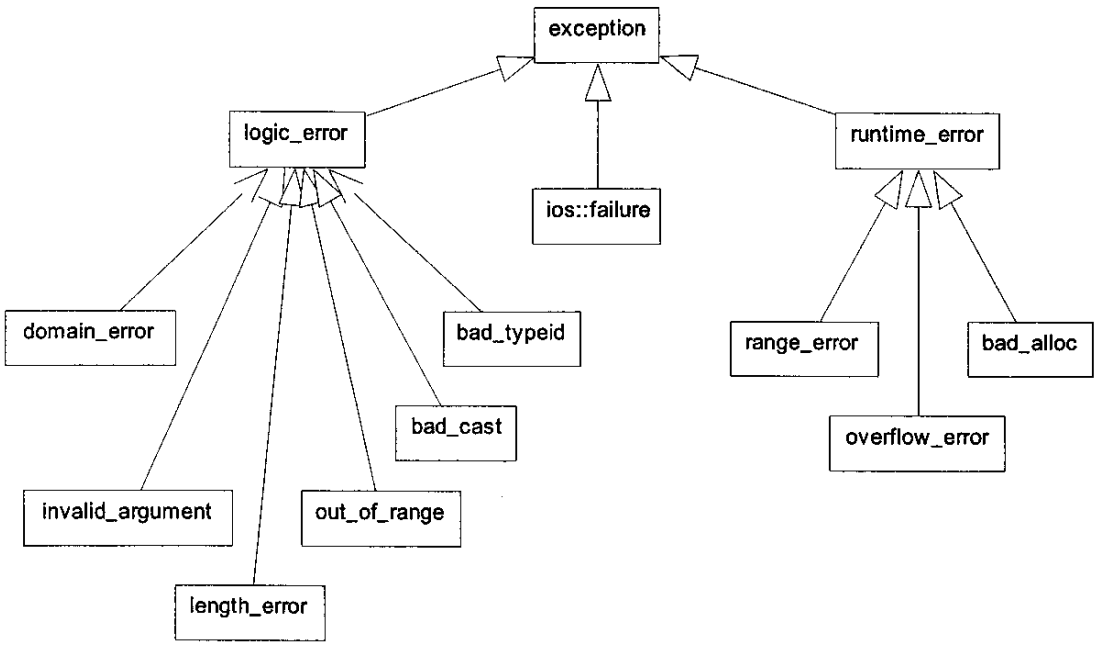
Typliste enthält die Datentypen, die throw geworfen werden. Wird **keine Typliste** angegeben, darf die Funktion alle Typen werfen. Wird eine **leere Typliste** angegeben, so darf die Funktion keine Fehler werfen.
Eine Verletzung der Exception Spezifikation wird nicht beim Übersetzen geprüft, sondern kann erst zur Laufzeit des Programms entdeckt werden. Eine solche Verletzung führt zum Aufruf des Handlers `unexpected()`.

Wenn fremde Klassen verwendet werden, sind nicht immer alle Exceptions bekannt, die aus diesen Klassen geworfen werden können. Damit besteht immer das Risiko in den Handler unexpected() hineinzulaufen.

8. terminate(), unexpected()

- **`terminate()`**
  `terminate()` fängt alle Fehler, für die kein Handler definiert wurde. Das Programm wird mit `abort()` abgebrochen. Es ist auch möglich einen eigenen Handler an Stelle des eingebauten `terminate()` ins System zu hängen (mit `set_terminate()`).
- **`Unexpected()`**
  `unexpected()` wird aufgerufen, wenn eine Funktion einen Fehler wirft, der nicht in ihrer Exeptionen eigenen Handler an Stelle des eingebauten `unexpected()` ins System zu hängen (mit `set_unexpected()`)

9. Exception Klassen



Fragen zu Exceptions:

Befehl exit(1); geht nicht. Wo ist er drin???????????

Error-Klasse muss zuerst erstellt werden??

Programm EXIT

Unterschied catch-Block - Funktionsimplementierung
Catch-Block:

- kein Rückgabewert
- nur 1 Parameter
- wird automatisch aufgerufen infolge throw

Funktionsimplementierung:

- wird explizit aufgerufen

Klasse Error z.B. Seite 3-222ff Definition S. 3-225

Bei throw; Sie kann nur in einem Exception Handler verwendet werden, heisst das, dass es nur einen catch-Block geben darf????

Exceptions in C++3.0 Seite 3-225 ist nicht so wichtig, oder??

Exception Klassen S. 3-226 wird auch nicht kommen...???????????