Exceptions

Exceptions (*Ausnahmen*) und ihre Behandlung ermöglichen es die weitere Auswertung von Programmteilen abzubrechen und stattdessen Informationen über den aufgetretenen Fehler an eine andere Stelle im Programm zu kommunizieren wo dann geeignet reagiert werden kann.

⟨ThrowAusd⟩ → "`throw`" [⟨Wert⟩] "`;`"

`throw`-Ausdrücke sind als Ausdruck vom Typ `void`. Wird ein `throw` Ausdruck ausgeführt, wird die Ausführung des Programms an dieser Stelle abgebrochen. Man sagt der an `throw` übergebene Wert wird als exception *geworfen*.

⟨TryBlock⟩ → "`try`" "`{`" {⟨Anweisung⟩} "`}`" {⟨Handler⟩}
⟨Handler⟩ → "`catch`" "`(`" ⟨ExceptionDekl⟩ "`)`" "`{`" {⟨Anweisung⟩} "`}`"
⟨ExceptionDekl⟩ → "`...`" | ⟨Typ⟩ [⟨Deklarator⟩]

Wird eine exception geworfen, so wird nach dem *nächsten* `try`-Block gesucht der assoziiert ist einem *passendem* `catch`-Block. Hierfür wird zunächst lokal gesucht (ob sich der `throw`-Ausdruck also direkt in einem `try`-Block mit passendem `catch`-Block befindet) und danach etwaige Funktionsaufrufe „rückabgewickelt". Es wird also betrachtet von wo die aktuelle Funktion aufgerufen wurde und ob der Aufruf dort in einem `try`-Block mit passendem `catch`-Block liegt. Falls nicht wird auf die gleiche Weise weiter gesucht bis, als erster Einstiegspunkt in das Programm, in der Funktion `main` gesucht wird.

Für eine exception vom Typ T sind jene `catch`-Blöcke *passend* für die der in den runden Klammern angegebene Typ T ist (oder Varianten davon wie `const` T oder T`&`). Zusätzlich ist `catch (...)` immer passend. Bei mehreren passenden `catch`-Blöcken die mit dem selben `try`-Block assoziiert sind, ist die Reihenfolge der `catch`-Blöcke ausschlaggebend.

Der Ausdruck `throw` ohne Wert ist nur innerhalb eines `catch`-Blocks zulässig und wirft den ursprünglichen Wert der vom `catch`-Block gefangen wurde erneut.

Wird eine exception zwar geworfen aber nie gefangen, so wird die spezielle Funktion `std::terminate()` aufgerufen. Voreinstellung ist, dass `std::terminate()` die Ausführung des Programms abbricht.

Exceptions in der Standardlibrary

Alle Typen deren Werte als exceptions von Funktionen der Standardlibrary geworfen werden, implementieren eine Methode `.what()` die eine `\0`-terminierten C-Zeichenkette (vom Typ `const char*`) zurückgibt, die den aufgetretenen Fehler beschreibt.

Exceptions bei Speicherreservierung

Die Header-Datei `<new>` enthält einen exception-Typ `std::bad_alloc`. Werte vom Typ `bad_alloc` werden als exception geworfen, insb. wenn bei Aufruf des `new`-Operators nicht genug Speicher reserviert werden kann.

Beispiel. Wir lesen von der Standardeingabe eine natürliche Zahl ein und versuchen ein C-Array von `double` dieser Länge zu reservieren. Wenn die Reservierung eine exception vom Typ `bad_alloc` wirft, wird diese in

einem `catch`-Block gefangen und eine Fehlermeldung ausgegeben.

dyn_bad_alloc.cpp

```cpp
#include <iostream>
#include <new>
#include <cstddef>

using namespace std;

int main() {
  size_t n;
  cout << "n: "; cin >> n;

  double* a;
  try {
    a = new double[n];

    a[0] = 3.14;
    cout << a[0] << endl;
  } catch (const bad_alloc& e) {
    cerr << "Zu wenig Speicherplatz fuer a vorhanden: "
         << e.what() << endl;
    return 1;
  }

  return 0;
}
```

Wir rufen das Programm auf und übergeben eine Länge für n, die definitiv zu groß ist.

```
n: 100000000000
Zu wenig Speicherplatz fuer a vorhanden: std::bad_alloc
```

Bereichsüberprüfter Elementen-Zugriff

Die Klasse `vector<T>` implementiert eine Methode `.at(n)` die eine Referenz auf das Element mit Index n zurückgibt. Falls n kein valider Index für den Vektor ist, wird stattdessen eine exception vom Typ `std::out_of_range` (aus `<stdexcept>`) geworfen.

Beispiel. Wir lesen von der Standardeingabe eine natürliche Zahl n ein und versuchen einen C++-Vektor von `double` dieser Länge zu konstruieren. Wenn die Reservierung eine exception vom Typ `bad_alloc` wirft, wird diese in einem `catch`-Block gefangen und eine Fehlermeldung ausgegeben. Nach Konstruktion des C++-Vektor wird versucht auf die Komponente mit Index n zuzugreifen. Dies sollte eine exception vom Typ `out_of_range` werfen.

vector_exc.cpp

```cpp
#include <iostream>
#include <new>
#include <vector>
```

```cpp
using namespace std;

int main() {
  vector<double>::size_type n;
  cout << "n: "; cin >> n;

  cout << "Beginn" << endl;
  try {
    cout << "Vor Vereinbarung" << endl;
    vector<double> a(n);
    cout << "Nach Vereinbarung" << endl;
    a.at(n) = 1;
    cout << "Nach Zugriff" << endl;
  } catch (const bad_alloc& e) {
    cerr << "Zu wenig Speicherplatz fuer a vorhanden: " << endl
         << "  " << e.what() << endl;
  } catch (const out_of_range& e) {
    cerr << "Zugriff ausserhalb von Grenzen:" << endl
         << "  " << e.what() << endl;
  } catch (...) {
    cerr << "Andere exception" << endl;
    throw;
  }
  cout << "Ende" << endl;

  return 0;
}
```

Wir rufen das Programm auf und übergeben eine Länge für n, die definitiv zu groß ist.

```
n: 100000000000
Beginn
Vor Vereinbarung
Zu wenig Speicherplatz fuer a vorhanden:
  std::bad_alloc
Ende
```

```
n: 10
Beginn
Vor Vereinbarung
Nach Vereinbarung
Zugriff ausserhalb von Grenzen:
  vector::_M_range_check: __n (which is 10) >= this->size() (which is 10)
Ende
```

Exceptions bei Ein-/Ausgabe

Die Stromklassen (`istream`, `ostream`, `fstream`, ...) implementieren eine Methode `.exceptions(`m`)`. Der Parameter m ist vom Typ `ios::iostate`. Aufrufen der Methode `.exceptions(`m`)` setzt eine interne Bitmaske des Stromobjekts, auf dem sie aufgerufen wurde, auf den Wert vom m. Sobald im internen Zustand des Stromobjekts ein bit (`ios::badbit`, `ios::eofbit`, `ios::failbit`) gesetzt wird, das auch in m gesetzt ist, so wird eine exception vom Typ `ios::failure` geworfen.

Beispiel. Wir implementieren ein Hauptprogramm, dass eine ganze Zahl von der Standardeingabe ein- und dann auf der Standardausgabe ausgibt. Wir aktivieren vorher das Werfen von exceptions wenn im Stromzustand von `cin` die bits `ios::failbit` oder `ios::badbit` gesetzt werden.

`iostream_exc.cpp`

```cpp
#include <iostream>

using namespace std;

int main() {
  cin.exceptions(ios::failbit|ios::badbit);

  try {
    int n;
    cout << "n: "; cin >> n;
    cout << "n: " << n << endl;
  } catch (const ios::failure& e) {
    cerr << "I/O-exception: "
         << e.what() << endl;
  }

  return 0;
}
```

Wir führen das Programm aus und übergeben auf der Standardeingabe eine Zeichenkette, die nicht erfolgreich als `int` interpretiert werden kann. Es wird eine exception vom Typ `ios::failure` geworfen, da im internen Zustand von `cin` das bit `ios::failbit` gesetzt wird, wenn das formattierte Einlesen eines Werts fehlschlägt.

```
n: abc
I/O-exception: basic_ios::clear: iostream error
```

Funktions-`try`-Blöcke

Es ist zulässig den Körper einer Funktion oder Methode durch einen `try`-Block (und folgende `catch`-Blöcke) zu ersetzen.

Im Falle eines Konstruktors muss das Schlüsselwort `try` *vor* der Initialisierungsliste stehen.

Beispiel. Wir vereinbaren eine Klasse `Rational` zur modellieren von rationalen Zahlen. Wir vereinbaren die `main`-Funktion mit Funktions-`try`-Block um im Programmverlauf etwaige auftrende exceptions zu verarbeiten. Wir stellen sicher, dass an Stellen, an denen Fehler auftreten können, i.d.R. exceptions geworfen werden, im Fehlerfall.

`rational_exc.cpp`

```cpp
#include <iostream>
#include <stdexcept>
#include <sstream>
```

```
using namespace std;

class Rational {
  private:
    long p, q;

    void normalisieren() {
      long teiler = ggt(abs(p), abs(q));

      if (teiler > 1) {
        p /= teiler;
        q /= teiler;
      }
      if (q < 0) {
        p = -p;
        q = -q;
      }
      if (p == 0)
        q = 1;
    }

    static long ggt(long a, long b) {
      if (a == 0 && b == 0)
        throw domain_error("ggt(0, 0)");

      long r;
      do {
        r = a % b;
        a = b;
        b = r;
      } while (r != 0);
      return r;
    }

  public:
    Rational(long p_ = 0, long q_ = 1): p(p_), q(q_) {
      if (q == 0) {
        ostringstream os;
        os << "Unerlaubter Aufruf Rational(" << p << ", " << q << ")";
        throw domain_error(os.str());
      }
      normalisieren();
    }

    friend ostream& operator<<(ostream& stream, const Rational& r) {
      if (r.q == 1)
        return stream << r.p;

      return stream << r.p << '/' << r.q;
    }
    friend istream& operator>>(istream& stream, Rational& r) {
      char c;
      stream >> r.p >> c >> r.q;
      if (c != '/' || r.q == 0) {
        stream.setstate(ios::failbit);
        return stream;
      }
```

```
    r.normalisieren();
    return stream;
  }

  friend Rational operator/(const Rational& s, const Rational& t) {
    if (t.p == 0) {
      ostringstream os;
      os << "Unerlaubte Operation: " << s << " / " << t;
      throw domain_error(os.str());
    }
    return Rational(s.p*t.q, s.q*t.p);
  }
};

int main() try {
  cin.exceptions(ios::failbit|ios::badbit);

  Rational r, s;
  cout << "p1/q1 p2/q2: ";
  cin >> r >> s;
  cout << "r/s = " << r/s << endl;
  return 0;
} catch (const domain_error& e) {
  cerr << "Bereichsfehler: " << e.what() << endl;
} catch (const ios::failure& e) {
  cerr << "I/O-Fehler: " << e.what() << endl;
} catch (...) {
  cerr << "Sonstige exception" << endl;
}
```

```
p1/q1 p2/q2: 1/2 3/4
r/s = 4/6
```

```
p1/q1 p2/q2: 1/0 3/4
I/O-Fehler: basic_ios::clear: iostream error
```

```
p1/q1 p2/q2: 1/2 0/4
r/s = Bereichsfehler: Unerlaubte Operation: 1/2 / 0
```