Katastrophenschutz mit try und catch

Willemers Informatik-Ecke

Es ist unmöglich, idiotensichere Programme zu schreiben, da Idioten so erfinderisch sind.

Tatsächlich ist es nicht leicht, gegen das Gesetz von Murphy (»If anything can go wrong it will.«) anzukämpfen. Der Programmierer muss nicht nur den Algorithmus im Blick behalten, sondern alle Fehleingaben, Systemfehler und Unverträglichkeiten vorhersehen und ausschalten.

try und catch

C++ bietet einen Mechanismus, der es ermöglicht, einen Block von Anweisungen gegen Abstürze zu sichern. Alle darin auftretenden Ausnahmefehler werden einem Behandlungsblock zugeleitet. Der gesicherte Anweisungsblock wird durch das Schlüsselwort `try` (engl. versuchen) eingeleitet. Der Fehlerbehandlungsblock beginnt mit `catch` (engl. fangen). Im Anweisungsblock wird also versucht, das Programm fehlerfrei abzuarbeiten. Der Fehlerbehandlungsblock fängt die Abstürze ab wie ein Netz.

[Fehlerfangen]

```
try
{
    int divisor=0;
    int dividend=1;
    int quotient = dividend/divisor; // hier knallt es!
}
catch (...)
{

    cout << "Problem erkannt..." << endl;
}
```

Die Division durch 0 im Beispiel würde normalerweise zu einem Programmabsturz führen. Man nennt einen solchen Fehler eine >>Exception<< oder auf Deutsch >>Ausnahme<<. Tritt eine solche Ausnahme in einem `try`-Block auf, dann wird die Verarbeitung in dem behandelnden `catch`-Block fortgesetzt. [1]

Vorteile

Natürlich kann das Problem verhindert werden, indem vor jeder Division der Divisor geprüft wird. Aber mit Hilfe der `try`-Anweisung wird die Fehlerbehandlung bewusst aus dem normalen Programmablauf herausgenommen. Diese Trennung von Code und Fehlerbehandlung führt zu Programmen, die leichter lesbar und damit auch leichter zu warten sind. Ein weiterer Vorteil liegt darin, dass nicht jeder Teilschritt einzeln auf jede erdenkliche Fehlersituation geprüft werden muss. Ein abschreckendes Beispiel für eine solche Überfrachtung durch Fehlerbehandlungen sehen Sie hier.

Drei Punkte

Die drei Punkte hinter dem `catch`-Befehl bedeuten nicht, dass dem Autor nichts mehr eingefallen wäre, sondern bezeichnen den generellen oder allgemeinen

Ausnahmebehandlungsfall. In der Klammer der `catch`-Anweisung wird der Typ der Exception benannt, die von diesem Block abgefangen werden soll. Es ist möglich, mehrere `catch`-Blöcke für unterschiedliche Typen hintereinander zu setzen. Die drei Punkte sind ein Platzhalter für alle Typen, so dass hier alle noch nicht behandelten Ausnahmen gefangen werden sollen. Dieser allgemeine `catch` muss immer als letzter Fehlerbehandlungsblock stehen.

Wirkungskreis

Eine Ausnahmebehandlung wirkt nicht nur auf den Block selbst, sondern auch auf alle innerhalb des Blocks aufgerufenen Funktionen. Dadurch kann die Fehlerbehandlung an einer zentralen Stelle durchgeführt werden. Außerdem kann zusätzlich an jeder anderen Stelle des Programms eine Ausnahmebehandlung stattfinden. Sobald es zu einem Fehler kommt, wird die nächste `catch`-Anweisung verwendet, die auf den Ausnahmetyp passt. Um die nächste Ausnahmebehandlung zu finden, wird die Aufrufreihenfolge rückwärts durchlaufen. Dadurch wird immer die Ausnahmebehandlung verwendet, die der Fehlerstelle am nächsten ist.

Eigene Ausnahmen erzeugen

throw

Nicht alle Fehler führen immer gleich zu einer Ausnahmesituation. Aber nur Ausnahmen können durch `catch` abgefangen werden. Darum ist es besonders interessant, eigene Ausnahmesituationen definieren zu können. Der Befehl `throw` erzeugt eine solche Ausnahme. Er bricht die Verarbeitung sofort ab und springt direkt in die passende Ausnahmebehandlung.

[Ausnahme mit throw generieren]

```
#include <iostream>
using namespace std;

void Tuwas(int Problem)
{
    if (Problem>0)
    {
        throw 0;
    }
}

int main()
{
    try
    {
        Tuwas(1);
    }
    catch(...)
    {
        cout << "Da gab es ein Problem!" << endl;
    }
}
```

Der Befehl `throw` in der Funktion `Tuwas()` löst eine Ausnahme aus, wenn der Parameter **Problem** größer als 0 ist. Die Verarbeitung wird sofort im `catch`-Block fortgesetzt.

Parameterübergabe

Manchmal reicht es aber nicht, dass die Behandlungsroutine einfach aufgerufen wird. Um einen Fehler behandeln zu können, braucht sie Informationen wie beispielsweise den Namen der Datei, die die Probleme verursacht oder eine Fehlernummer. Diese Informationen können mit dem Argument der `throw`-Anweisung an den `catch`-Block übermittelt werden. Als erstes Beispiel soll eine ganze Zahl übermittelt werden. `catch` bekommt nun einen Parameter wie eine Funktion, in diesem Fall vom Typ `int`.

[throw übermittelt eine Fehlernummer]

```
#include <iostream>
using namespace std;

void Tuwas(int Problem)
{
    if (Problem>0)
    {
        throw 5;
    }
}

int main()
{
    try
    {
        Tuwas(1);
    }
    catch(int a)
    {
        cout << "Ausnahme: " << a << endl;
    }
}
```

Wenn der `catch`-Block `int` als Parameter hat, bearbeitet er nur Ausnahmen, die durch einen `throw`-Befehl mit einer Zahl als Argument ausgelöst wurden. Um andere Parameter zu bearbeiten, wird einfach ein weiterer `catch`-Block mit einem anderen Parametertyp angehängt. Um alle übrigen Ausnahmen zu behandeln, kann der allgemeine `catch`-Befehl mit den drei Punkten als Parameter ganz zum Schluss auch noch angehängt werden. Ganz ähnlich wie bei überladenen Funktionen werden die passenden `catch`-Blöcke nach ihren Parametern ausgewählt. Allerdings hat ein `catch` immer nur einen Parameter. Eine automatische Typanpassung wie bei Funktionen wird hier nicht durchgeführt. Wenn Sie also beispielsweise `throw` mit dem Argument 1.2 verwenden, muss der passende `catch` als Parameter `double` und nicht `float` haben, weil Fließkommakonstanten vom Compiler standardmäßig als `double` behandelt werden. Das folgende Beispiel zeigt mehrere `catch`-Blöcke für verschiedene Typen.

[Mehrere catch-Blöcke (throwtyp.cpp)]

```
// Programm zur Demonstration verschiedener catch-Bloecke
#include <iostream>
using namespace std;

// Tuwas wird unterschiedliche Typen werfen, je nach dem
// Wert des Parameters Problem.
void Tuwas(int Problem)
```

```
{
    switch (Problem)
    {
    case 0: throw 5; break;                   // wirft int
    case 1: throw (char *)"test.dat"; break; // wirft char *
    case 2: throw 2.1; break;                 // wirft double
    case 3: throw 'c'; break;                 // wirft char
    }
}

// Testprogramm
int main()
{
    // Problem-Nummer eingeben
    int Auswahl;
    cout << "Zahl zwischen 0 und 3 eingeben:" << endl;
    cin >> Auswahl;
    // Der try-Block fängt die Exception in Tuwas
    try
    {
        Tuwas(Auswahl);
    }
    catch(int i) // Behandler für int
    {
        cout << "Integer " << i << endl;
    }
    catch(char *s) // Behandler für char*
    {
        cout << "Zeichenkette " << s << endl;
    }
    catch(double f) // Behandler für double
    {
        cout << "Fließkomma " << f << endl;
    }
    catch(...) // fängt alle anderen Exception-Typen
    {
        cout << "Allgemeinfall" << endl;
    }
}
```

Wenn Sie das Programm starten, können Sie durch die Zahlen 0 bis 3 auswählen, welche Ausnahme ausgelöst wird. Bei 1 wird eine Zeichenkette >>geworfen<<, die dann im `catch`-Block als Parameter **s** auch weiterverarbeitet werden kann. Bei 3 wird ein `char` >>geworfen<<, für den kein `catch`-Block vorgesehen ist. Also wird diese Ausnahme vom allgemeinen `catch`-Block gefangen.

Erstellen von Fehlerklassen

Sie können auch eigene Klassen erstellen, die als Parameter für `catch` verwendet werden können. Selbst leere Fehlerklassen können allein durch einen gut gewählten Namen bereits sehr nützlich sein.

[Eigener Fehlertyp]

```
#include <iostream>
using namespace std;

class KeineDatenMehr
```

```
{
};

void Tuwas(int Problem)
{
    ...
        throw KeineDatenMehr();
    ...
}

int main()
{
    try
    {
        Tuwas(0);
    }
    catch(KeineDatenMehr& )
    {
        cout << "Keine Daten mehr vorhanden" << endl;
    }
}
```

Fehleridentifizierung

Die Klasse **KeineDatenMehr** enthält gar nichts. Sie trägt also nichts zur Fehlerbehandlung bei, außer ihrem Namen. Aus diesem lässt sich aber bereits schließen, dass hier eine Fehlersituation vorliegt, in der der Datenstrom versiegt ist. Diese Situation wird nun separat von allen anderen Fehlern behandelt. Durch den sprechenden Namen ist jedem Leser klar, was den `throw` auslöst und welche Ausnahme `catch` empfängt. Sollte in dem Programm nur eine Datenquelle verwendet werden, kann der Klassenname bereits völlig ausreichend sein. Die Behandlungsroutine braucht dann keine näheren Informationen.

Fehlerklasse erweitern

Darauf aufbauend ist es einfach, in der Fehlerklasse auch Informationen über die Fehlerursache zu hinterlegen. Diese Hinweise kann dann der Behandlungsblock auswerten. Es ist auch möglich, in der Fehlerklasse Funktionen zur Fehlerbehandlung einzubauen, die dann im `catch`-Block aufgerufen werden. Im folgenden Beispiel wird ein ganzzahliger Wert im Fehlerobjekt hinterlegt und dann durch die Funktion `MeldeFehler()` angezeigt.

[Fehlerklasse wird aktiv]

```
#include <iostream>
using namespace std;

class KeineDatenMehr
{
public:
    KeineDatenMehr(int a) { nr = a; }
    void MeldeFehler() { cout << nr << endl; }
private:
    int nr;
};

void Tuwas(int Problem)
{
    if (Problem==0)
    {
```

```
        throw KeineDatenMehr(8);
    }
}

int main()
{
    try
    {
        Tuwas(0);
    }
    catch(KeineDatenMehr& fehler)
    {
        fehler.MeldeFehler();
    }
}
```

Polymorphie

Um verschiedene Fehlersituationen zu behandeln, bietet es sich an, eine Basisklasse für alle Fehlerklassen zu schreiben. Darin implementieren Sie, was allen Fehlersituationen gemeinsam ist. Die Möglichkeiten der Vererbung lassen sich auf diese Weise praktisch nutzen. Es besteht darüber hinaus auch keine Notwendigkeit, für jede Fehlersituation einen eigenen `catch`-Block zu schreiben, wenn die Polymorphie genutzt wird, um je nach Fehlersituation die Behandlungsfunktion der jeweiligen Klasse aufzurufen.

[Polymorphe Fehlerklasse (tryclass.cpp)]

```
#include <iostream>
using namespace std;

class meaCulpa // Basisklasse aller meiner Fehler
{
public:
    virtual void MeldeFehler() = 0;
};

class KeineDatenMehr : public meaCulpa
{
public:
    KeineDatenMehr(int a) { nr = a; }
    void MeldeFehler() { cout << nr << endl; }
private:
    int nr;
};

class QuelleFehlt : public meaCulpa
{
public:
    QuelleFehlt() {}
    void MeldeFehler() { cout << "Quelle fehlt" << endl; }
};

void Tuwas(int Problem) throw (KeineDatenMehr, QuelleFehlt)
{
    if (Problem==0)
    {
        throw KeineDatenMehr(8);
    }
    if (Problem==1)
    {
```

```
        throw QuelleFehlt();
    }
}

int main()
{
    try
    {
        Tuwas(0);
    }
    catch(meaCulpa& fehler)
    {
        fehler.MeldeFehler();
    }
}
```

Wie immer bei der Polymorphie wird ein Zeiger auf das übergebene Objekt verwendet, um die virtuelle Funktion aufzurufen. Das Objekt kennt sich selbst und ruft über die eigene VTable die zugehörige Funktion `MeldeFehler()` auf. Einen ähnlichen Ansatz bieten die Standardbibliotheken. Sie verwenden Fehlerklassen, die auf der Klasse **exception** basieren. Wie Sie Ihre Fehlerklasse von **exception** ableiten können, wird auf den nächsten Seiten beschrieben.

throw-Deklaration

In Listing sehen Sie, dass der Funktion `Tuwas()` eine Deklaration hinzugefügt wurde, die anzeigt, welche Ausnahmen sie auslöst. Diese Informationen sollen darauf hinweisen, dass die Funktion `throw`-Befehle enthält. In der Klammer wird aufgezählt, welche Ausnahmen ausgelöst werden. Der Compiler prüft nicht, ob diese wirklich enthalten sind. Nicht einmal das Fehlen jeglicher `throw`-Befehle würde ihm auffallen. Allerdings macht der Compiler mächtig Ärger, wenn die Funktion versucht, eine Ausnahme auszulösen, die in der Deklaration nicht angekündigt war. Daraus folgt, dass eine Funktion, deren `throw`-Deklaration keinen Typ zwischen den Klammern enthält, sich verpflichtet, keinen `throw` auszuführen.

Die Ausnahmen der Standardbibliotheken

Die Klasse exception

Die Standardbibliotheken von C++ implementieren Ausnahmebehandlungen. Dabei werden ebenfalls Fehlerklassen definiert, die auf einer Basisklasse beruhen. Diese Basisklasse heißt **exception**. Damit lässt sich jede Ausnahme der Standardbibliotheken durch ein `catch` mit dem Parameter `exception&` abfangen.

Eigene Ableitung von exception

Es ist durchaus sinnvoll, auch eigene Ausnahmen von **exception** abzuleiten. Dazu muss die Funktion `what()` implementiert werden, da diese virtuell ist. Sie liefert einen Zeiger auf einen C-String, der Informationen über den Fehler im Klartext anzeigt. Hier sehen Sie die Definition der Klasse **exception**.

```
class exception
{
public:
    exception() throw();
```

```
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char * what() const throw();
};
```

Um eine solche Ableitung zu definieren, muss zunächst die Header-Datei **exception** eingebunden werden, da hier die Klasse **exception** definiert wird. Das folgende Beispiel zeigt die Klasse **meaCulpa**, die nun von **exception** abgeleitet wird. Als spezielle Ausnahmen werden wieder **KeineDatenMehr** und **QuelleFehlt** davon abgeleitet.

[Fehlerklassen von exception ableiten (exception.cpp)]

```
#include <iostream>
#include <exception>
#include <string>
#include <sstream>
using namespace std;

// Meine eigene Basisklasse, abgeleitet von exception
class meaCulpa : public exception
{
public:
    meaCulpa(string s) {this->s = s;}
    virtual ~meaCulpa() throw() {}
    virtual const char * what() const throw()
        {return s.c_str();}
private:
    string s;
};

// Besonderer Fehler, wenn keine Daten mehr vorliegen
class KeineDatenMehr : public meaCulpa
{
public:
    KeineDatenMehr(int a) : meaCulpa(" ") {nr = a;}
    virtual ~KeineDatenMehr() throw() {}
    virtual const char * what() const throw();
private:
    int nr;
    string s;
};

// what() wird für eigene Fehlermeldung überschrieben
const char * KeineDatenMehr::what() const throw()
{
  ostringstream getNr;
  getNr << "Keine Daten mehr. Fehlernr.: " << nr;
  return getNr.str().c_str();
}

// Eine weitere Fehlerart wird von meaCulpa abgeleitet.
class QuelleFehlt : public meaCulpa
{
public:
    QuelleFehlt() : meaCulpa("Quelle fehlt") {}
};

// Tuwas simuliert die beiden Fehlerarten in Abhängigkeit vom
// Parameter
void Tuwas(int Problem) throw (KeineDatenMehr, QuelleFehlt)
```

```
{
    if (Problem==0)
    {
        throw KeineDatenMehr(8);
    }
    if (Problem==1)
    {
        throw QuelleFehlt();
    }
}

int main()
{
    // Problem-Nummer eingeben
    int Auswahl;
    cout << "Zahl zwischen 0 und 3 eingeben:" << endl;
    cin >> Auswahl;
    // Der try-Block fängt die Exception in Tuwas
    try
    {
        Tuwas(Auswahl);
    }
    // Fängt nur die eigenen Fehler
    catch(meaCulpa& fehler)
    {
        cout << fehler.what() << endl;
    }
}
```

Die Fehlerklasse **KeineDatenMehr** erwartet im Konstruktor eine ganze Zahl. Der Konstruktor ruft für **meaCulpa** den Konstruktor mit dem String als Parameter auf. Die Funktion `what()` wird von **KeineDatenMehr** neu implementiert. Die Fehlernummer wird, begleitet von ein wenig Text, in eine Zeichenkette konvertiert und als C-String zurückgegeben. Die Fehlerklasse **QuelleFehlt** gibt einfach ihre Meldung an den Konstruktor von **meaCulpa** durch und braucht darum `what()` nicht neu zu implementieren.

Übersicht über die Standardfehlerklassen

Die Standardbibliotheken von C++ haben folgende Ausnahmeklassen definiert, die sich alle von der Klasse **exception** ableiten:

runtime_error
: Laufzeitfehler. Davon abgeleitet sind folgende Fehlerklassen:
ios_base::failure
Fehlerklasse der Stream-Klassen. Sie ist nicht bei allen Compilern verfügbar.
range_error
Bereichsüberschreitung. Wird durch die `at()`-Funktion ausgelöst.
overflow_error
Überlauf bei Berechnungen.
underflow_error
Unterlauf bei Berechnungen.

logic_error
: Logische Fehler. Davon abgeleitet sind folgende Fehlerklassen:

domain_error
: Bereichsfehler.

invalid_argument
: Unzulässiges Argument.

length_error
: Beim Anlegen wurde die maximal mögliche Größe überschritten.

out_of_range
: Zugriff mit unzulässigem Index.

Die Sprache C++ hat darüber hinaus noch einige Standardausnahmen:

bad_alloc
: Entsteht, wenn `new` keinen Speicher anfordern kann.

bad_cast
: Im Zusammenhang mit **dynamic_cast**.

bad_exception
: Wenn die Ausnahmebehandlung selbst Probleme bekommt.

bad_typeid
: Im Zusammenhang mit **typeid**.

Die Standardfehlerklassen können natürlich auch in den eigenen Funktionen als Argument für `throw` verwendet werden. Dann ist es sinnvoll, dem Konstruktor eine eigene Fehlermeldung zu übergeben:

```
#include <stdexcept>

int main()
{
  try
  {
    throw range_error("Oha, ist das eng hier!");
  }
  catch(range_error& e)
  {
     cout << e.what() << endl;
  }
}
```

Ausnahmen der Standardklasse fstream

Gerade bei Dateizugriffen ist die Fehlerbehandlung wichtig, da es unzählige Gründe gibt, warum ein Zugriff nicht funktioniert. Aus diesem Grund ist die Klasse **fstream** auch bereits auf die Ausnahmebehandlung vorbereitet.

Aktivieren

Das Auslösen der Ausnahmebehandlung muss zunächst für jedes Datenstromobjekt freigeschaltet werden. Dazu wird die Funktion `exceptions()` aufgerufen. Als Parameter können Sie eine oder mehrere der Konstanten `ios::failbit`, `ios::badbit` oder `ios::eofbit` übergeben. Sollen mehrere gültig sein, werden sie durch einen senkrechten Strich voneinander getrennt. Der Parameter für `catch` ist `ios::failure&`. Ein typischer Programmausschnitt sieht also wie folgt aus:

[fstream mit Ausnahmen (fstreamexception.cpp)]

```
fstream f;
try
{
    f.exceptions(ios::failbit|ios::badbit);
    f.open("test.dat", ios::in);
    f << "Dieser Text geht in die Datei" << endl;
    f.close();
}
```

```
catch(ios::failure&)
{
    if (f.fail()) cout << "fail" << endl;
    if (f.bad())  cout << "bad" << endl;
}
```

Leider kennen der GNU-Compiler und der Borland C++-Builder bis zur Version 4 `ios::failure` noch nicht. Wie Sie am Listing sehen können, ist das nicht so dramatisch, da die Klasse `ios::failure` keine Informationen über die Datei enthält. So muss das **fstream**-Objekt dem `catch`-Block sowieso per Direktzugriff zur Verfügung stehen. So kann gleichwertig stattdessen auch **exception** verwendet werden. Sie können aber davon ausgehen, dass im Laufe der Zeit alle Compiler diese Ausnahme unterstützen werden.

---

[1] Bei vielen Versionen des GNU-Compilers (auf jeden Fall bei gcc-2.9.2 und bei gcc-2.95.3, vermutlich auch bei neueren) werden die Divisionen durch 0 und die Fließkomma-Ausnahmen nicht vom generellen catch(...) abgefangen.
