Kapitel 16 - Ausnahmebehandlung

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

- Bisher: Auftreten von Fehlern war nebensächlich
  - Im ersten Teil der Vorlesung: `errno`
  - Außerdem: Bestimmte Rückgabewerte von Funktionen deuten auf Fehler hin
  - Beispiel: `open()` (Posix-Funktion)
- Aber: Fehlerbehandlung ist aber essentiell wichtig!
- Ein gutes Programm …
  1. verarbeitet unerwartete Ereignisse („Fehlerfall“) angemessen.
  2. verursacht im Katastrophenfall einen möglichst geringen Schaden.
- Beachtet
  - Berücksichtigen der möglichen Fehlerfälle ist die hohe Kunst!
  - Nehmt aus Faulheit **keine** Abkürzungen

Rückblick – Fehlerbehandlung in C

- Aus C sind bereits einige Arten der Fehlerbehandlung bekannt
- Die einfachste Methode: Programm beenden
  - Radikal: Zum Beispiel durch Aufruf `exit()` (`stdlib.h`) oder vorzeitiges `return` in `main()`

```
1  bool func_call() { /*...*/ }
2
3  int main() {
4      bool success = func_call();
5      if (!success) {
6          exit(1);
7      } else {
8          exit(0);
9      }
10 }
```

- Funktioniert prinzipiell einwandfrei, **aber**:
  - Nur für sehr kleine Programme sinnvoll
  - Sehr schlecht geeignet für den Dauerbetrieb (z.B. Webserver)
  - **Nicht akzeptabel** in sicherheitsrelevanten Anwendungen

Rückblick – Fehlerbehandlung in C

- Besserer Ansatz: Kodierung über Rückgabewert, z.B. `int`
- Unmittelbare Vorteile
  - Kein Abbruch mehr notwendig
  - Fehlergrund kann an Ort und Stelle erkannt und behandelt werden

```c
// Beispiel aus Posix
#include <fcntl.h>

int main() {
    // - 1, falls Fehler auftritt
    int ret = open("file.txt", O_RDONLY);
    if (ret < 0) {
        // Fallunterscheidung für Fehlertyp
    }
}
```

- Unmittelbarer Nachteil: Code wird komplexer
  - Fehlerfälle müssen nun einzeln kodiert bzw. dekodiert und behandelt werden
  - Das ist leider der Tradeoff 😕
- Ein weiteres Problem: Es ist nur ein Rückgabetyp möglich

Rückblick - Fehlerbehandlung in C

- Zusätzlich zum Rückgabewert: `errno`
  - Kodiert den Fehlergrund
  - Globale Variable
  - Gilt für Funktionen der C-Standardbibliothek

```c
#include <fcntl.h>
#include <errno.h>

int main() {
    int ret = open("file.txt", O_RDONLY);
    if (ret < 0) {
        if (errno == ENOENT) { /*...*/ }
    }
}
```

- `errno` hat eine Reihe von Problemen
  - **Globale** Variable aus `errno.h`
  - Kodiert nur den jeweils letzten Fehlergrund (automatische Überschreibung)

    → Prüfung wird sehr leicht vergessen, Grund ist dann ggf. bei Folgefehlern nicht mehr nachvollziehbar
  - Wertekodierung ist abhängig von der Zielplattform

    → andere Fehlercodes unter Linux/UNIX als unter Windows

Ausnahmebehandlung in C++

- `errno` und Rückgabewerte existieren in C++ natürlich weiterhin
- Gerade bei Systemfunktionen führt oft kein Weg daran vorbei, wie `open()`

- Es gibt allerdings noch weitere Wege
  - Fehlerbehandlung mithilfe von **Ausnahmen** *(Exceptions)*
  - Kombinierter Rückgabewert (Union von Wert + Fehler) mithilfe von `std::expected` – nicht Thema in diesem Kapitel

Exceptions

- Ausnahmebehandlung ist tief in C++ verankert
  - Beispiel: `new` und `delete` können Ausnahmen werfen
- Konzept hinter Exceptions ist relativ simpel
  - Bei Auftritt eines Fehlers, z.B. *Datei existiert nicht*, wird dieser zunächst nur erkannt
  - Behandlung des Fehlers erfolgt **nicht** unmittelbar, keine lokale Fehlerbehandlung wie bisher
- Stattdessen: Signalisierung eines Fehlerfalls an aufrufende Funktion

→ „*Heißes Eisen wird weggeschoben und ist das Problem des Vorgesetzten*"

Terminologie

- Bei der Signalisierung wird die Ausnahme **geworfen** (*throw an exception*)
- Aufrufende Funktion **fängt die Ausnahme** (*catch an exception*)
- Umsetzung
  - `try`: Beginn eines Code-Blocks, in dem eine Ausnahme ausgelöst werden könnte
  - `throw`: Werfen einer Ausnahme
  - `catch`: Code-Block, der potenziell die Ausnahme behandeln kann

```
1  try {
2      bool error = may_error();
3      if (error) {
4          throw MyException;
5      }
6  }
7  catch (MyException ex) {
8      // Behandlung der Ausnahme
9      // Zum Beispiel: Ausgabe
10     cout << ex.what();
11 }
```

Fangen einer Ausnahme

- Betreten des `catch`-Blocks, wenn die Signatur der Ausnahme passt
- Ansonsten: An den Nächsten weiterwerfen

  → *rethrow*
- Achtung: Beim Werfen wird die bisherige Funktion verlassen
  - Nach der Behandlung der Ausnahme erfolgt **keine** Rückkehr an die ursprüngliche Stelle des Kontrollflusses
  - Daraus resultierende Folgen werden gleich beleuchtet

```
1  try {
2      if (error) {
3          throw MyException;
4      }
5  }
6  catch (MyException ex) {
7      // Behandlung der Ausnahme
8      // Zum Beispiel: Ausgabe
9      cout << ex.what();
10 }
11 // ...
12 catch (OtherException ex) {
13     // ...
14 }
15 // ...
16 catch (int i) {
17     // ...
18 }
19 // ... und so weiter
```

Ablauf einer Ausnahmebehandlung

1. Der Reihe nach `catch`-Handler ablaufen
2. Falls ein Ausnahmetyp auf einen Handler passt, wird er verwendet

   → Nachfolgende Handler werden ignoriert
3. Kein passender Handler?

   → Aufwärtstraversierung der Aufrufkette zu darüberliegenden Funktionen
4. Falls auf der Ebene ein `try`-Block existiert → Schritt 1, sonst Schritt 3

- Ende der Ausnahmekette erreicht? → Aufruf von `std::terminate()`
  - Sofortiger Abbruch des Programms **ohne Rückkehr** zur `main()`
  - Gleiches geschieht übrigens, wenn eine **neue** Ausnahme während der Ausnahmebehandlung entsteht

Das Wurfobjekt

- Beliebiger Datentyp mittels `throw` geworfen werden
- Es sollte ein entsprechendes `catch` vorhanden sein
  - Kann aber leicht vergessen werden

```
throw 1;
throw "Error"; // const char *
throw SpecialErrorClass("Error");
```

```
try {
    throw int i;
}
catch (MyException ex) {}
catch (const char * msg) {}
catch (...) { // Catch-All
    cout << "Caught unhandled exception";
}
```

- Alternative: Die sogenannte Ellipse (`...`)
  - Eigentliche Verwendung: Akzeptieren beliebiger Parameter bei Funktionsaufruf (siehe `printf`)
  - Bei Ausnahmen: Akzeptieren **beliebiger Ausnahmetypen** (*Catch All*)

Beispiel – Exceptions

cpp Run ▶

```cpp
#include <iostream>
#include <string>

using std::cout, std::endl;

class Error {
    std::string msg;
public:
    Error(std::string s) : msg(s) {}
    const char* message() { return msg.c_str(); }
};

void throws_char_ptr() {
    try {
        throw "pointer error";
    }
    catch(int i) {  // Kein Matching gegen geworfenen Datentyp -> weiterreichen
        cout << i << endl;
```

Eigenschaften von Ausnahmeklassen

- Wie bei anderen Klassen auch
  - Erstellung wie bisher gewohnt
  - Dynamische Allokierung mit `new` und `delete` ebenfalls erlaubt (problematisch)
  - Referenzen ebenfalls erlaubt
  - Klassen dürfen Vererbung und Polymorphie verwenden
- Achtung: Fallstricke und Untiefen! 😱
- Beispiel: Aufräumen in **jedem** `catch`-Block erforderlich

```
1  try {
2      throw new MyException;
3  }
4  catch(MyException * me) {
5      delete me; // Ok...
6  }
7  // ...
8  catch(...) {} // ...und hier?
```

Throw by value, catch by reference

- Allgemeine Herangehensweise bei Ausnahmen:

  Throw by value, catch by reference

- Der Hintergrund ist simpel
  - *Throw by value* stellt sicher, dass keine Speicherlecks entstehen
  - *catch by reference* erlaubt Zugriff auf Methoden der Kinder einer gefangenen Klasse

    (Polymorphie statt versehentlicher Typkonvertierung)

```
1  class Base {};
2  class MyException :
3          public Base {}
4  try {
5      throw MyException();
6  }
7  catch(Base& e) {
8      // So ist es korrekt
9  }
10 // ...
11 catch(...) {}
```

Verwendung von `std::exception`

- Beliebige Datentypen als Ausnahme erlaubt
- Besser: verarbeitbare Datentypen verwenden
- Dafür bietet C++ eine eigene Klasse: `std::exception`
  - Hat eine virtuelle Methode `what()`
  - Beschreibt den Fehler, muss entsprechend in Kindklasse überschrieben werden
  - `const char* what() const noexcept override;`

```cpp
#include <exception>

using std::exception;

class MyException: public exception {
public:
    const char* what()
        const noexcept override {
        return "MyException :)";
    }
};

class IntError: public exception {
public:
    const char* what()
        const noexcept override {
        return "IntError";
    }
};
```

Beispiel – `std::exception`

cpp Run ▶

```cpp
#include <iostream>
#include <string>
#include <exception>

using std::cout, std::endl;
using std::exception;

class MyException: publ
public:
    const char* what()
        return "MyExcep
    }
};

class IntError: public exception {
public:
    const char* what() const noexcept override {
        return "IntError";
```

Ermöglicht deutlich bessere und übersichtlichere `catch` -Blöcke als generische Ellipse

Rethrowing

- Eine besondere Eigenschaft ist das sogenannte *Rethrowing*
- Nach dem Fangen und erneuten Abarbeiten kann eine Ausnahme erneut mit `throw` geworfen werden
- Grund: Andere Teile des Systems könnten auch am Auftreten der Ausnahme interessiert sein

cpp Run ▶

```cpp
#include <iostream>
#include <string>
#include <exception>

using std::cout, std::endl;
using std::exception;

class MyException: public exception {
public:
    const char* what() const noexcept override {
        return "MyException :)";
    }
};

void foo() {
    try {
        throw MyException();
    } catch (MyException& me) {
```

Reihenfolge der Exceptions

- Empfehlenswert: erst Spezialfälle behandlen
- Je weiter unten die Ausnahme behandelt wird, desto generischer sollte der gefangene Typ sein

```
int divide(int dividend, int divisor) {
    try {
        if (divisor == 0) {
            throw DivByZeroException();
        }
    }
}

catch (DivByZeroException e) { /*... */ }
// ..
catch (MathException e) { /*... */ }
// ..
catch (std::exception e) { /*... */ }
// ..
catch (...) { /*... */ }
```

Einschub: `finally` in anderen Programmiersprachen

- Ausnahmebehandlung in vielen populären Programmiersprachen ebenfalls üblich

  z.B. Java, Python oder Javascript

- Unterschied: **Nach** den `catch`-Blöcken ein Block mit Schlüsselwort `finally`

```
1  // Datenbank-Interaktion
2  try {
3      db_connection.open();
4  }
5  catch(Exception e) {
6      // Print
7  }
8  finally {
9      db_connection.close();
10 }
```

- Funktional wie ein Destruktor für die Ausnahmebehandlung

  → Wird **in jedem Fall** ausgeführt, unabhängig vom `catch`-Block

- Existiert in C++ nicht, wird aber garantiert einmal an anderer Stelle auftauchen

Ausnahmen im Konstruktor

- Exceptions auch in Konstruktoren / Destruktoren möglich
- 🚨 Gefährlich: Aufnahmebehandlung sorgt **eigentlich** für Aufruf der Destruktoren
  - Bei einem Konstruktoraufruf geschieht dies nicht
- Objekt gilt erst nach Konstruktoraufruf als vollständig angelegt
  - Möglichst keine Nebeneffekte im Konstruktor, wenn `throw` auftreten könnte

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std:: endl;
class Member {
public:
    Member() { cout << "MC" << endl; }
    ~Member() { cout << "MD" << endl; }
};

class Class {
    Member member;
public:
    Class() {
        cout << "CC" << endl;
        throw 4711;
    }
    ~Class() { cout << "CD" << endl; }
};
```

Ausnahmen im Destruktor

- Kritisch wird es erst bei den Destruktoren
- `try` -Block auch hier erlaubt, aber…
- Es gilt: Destruktoren werden am Ende eines Scopes aufgerufen
- Tritt hier eine Ausn

  → C++-Runtime beendet das Programm ohne zu zögern
- Konsequenz: **Niemals** `throw` im Destruktor propagieren
  - `catch` **muss** an Ort und Stelle geschehen

Exceptions im Destruktor am besten vermeiden

cpp Run ▶

```cpp
#include <iostream>
using std::cout, std:: endl;
class Member {
public:
    Member() { cout << "MC" << endl; }
    ~Member() { cout << "MD" << endl; }
};

class Class {
    Member m;
public:
    Class() { cout << "CC" << endl; }
    ~Class() {
        cout << "CD" << endl;
        try { throw 4711; }
        catch (int i) {} // So ok
    }
};
```

Exkurs: `noexcept`

- Es ist möglich, einzelne Funktionen als `noexcept` zu markieren
  - Beispiel: `void foo() noexcept { /* ... */ };`
  - Versprechen an Compiler, dass hier keine Exception erzeugt wird
- Warum ist das wichtig?
  - Manche Teile der STL haben sogenannte *Strong Exception Guarantee*
  - Dürfen unter keinen Umständen Ausnahmen erzeugen
- Betrifft auch STL-Container (z.B. `std::vector`):
  - Bei interner Reallokierung (→ mehr Kapazität) wird aus Effizienzgründen Move-Konstruktor verwendet
  - Ohne `noexcept` → bei Move-Konstruktor: **Rückfall auf Copy-Konstruktor** 😕 (deutlich langsamer)
  - Grund: Wenn bei Move eine Exception auftritt, dann könnte theoretisch der alte Vector-Speicher noch nicht auf `nullptr` gesetzt sein (→ ggf. doppeltes free(), darf *niemals* passieren)

Beispiel - noexcept

cpp Run ▶

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

using std::cout, std::endl;

class Foo {
    int id_;
public:
    Foo(int id) : id_(i
    Foo(const Foo & oth
        cout << "Copy I                _
    }
    Foo(Foo && other) noexcept : id_(other.id_) {
        cout << "Move Foo" << id_ << endl;
    }
};
```

Long story short: Annotiert *ungefährliche* Move-Konstruktoren mit noexcept

Probleme mit Exceptions

- Sind leider nicht völlig unumstritten
- In der Praxis teilweise sogar per Coding-Guideline verboten, z.B. bei Google
- Ein Teil der Probleme wurde bereits dargelegt
  - Beispiel: Probleme mit Zeigern und `delete`
- Auch im Bereich *Embedded* problematisch
  - Nehmen relativ viel Speicher ein
  - Bei 512 Kilobyte Gesamtspeicher können nicht mehrere hundert Kilobytes für Exceptions verwendet werden
  - Für Interessierte hier ein interessanter Vortrag zu Problemen und Lösungen (Youtube, Englisch)

Fazit zu Ausnahmen

- Sind Exceptions deswegen schlecht? Nein
- Ausnahmen sind nützlich, **wenn** etwas Performanceverlust im Tausch für schnellere Entwicklungszeit akzeptabel ist
- Klassische Beispiele
  - Web-Programmierung
  - GUI-Programmierung, z.B. Qt-Framework
- Aber: Seid Euch der Schwächen bewusst und berücksichtigt dies
- Außerdem: Andere Programmiersprachen verfügen auch über Ausnahmen
  - Beispiele: Javascript, Python, Java, C#
  - Problembereiche nicht so präsent, weil diese sich eher an die Desktopprogrammierung richten

Zusammenfassung

- Wiederholung: Klassische Fehlerbehandlung in C – inkl. der Schwächen
- Jetzt: Ausnamen in C++
- Drei neue Schlüsselwörter
  - `try`
  - `catch`
  - `throw`
- Beliebige Datentypen dürfen geworfen werden
- Reihenfolge der `catch` -Anweisungen ist wichtig
- 🚨 **keine** Ausnahmen im Destruktor werfen

Kapitel 16 - Ausnahmebehandlung