Praxisorientierte Einführung in C++
Lektion: "Exceptions"

Christof Elbrechter

Neuroinformatics Group, CITEC

June 18, 2014

Table of Contents

Motivation

- Motivation:
  - Funktionen/Methoden brauchen Möglichkeit, Fehler anzuzeigen
- Bisher:
  - Rückgabewert einer Funktion/Methode
  - Übergabe eines Objektes per Referenz, welches den Fehler kodiert
  - Globale Variable

Motivation

```
// Fehler wird durch negativen Rueckgabewert angezeigt
int f(int i);

// Fehler wird durch setzen von Eigenschaften von error
// angezeigt
int f(int i, Error &error);

// Fehler wird durch setzen von globaler Variable
// angezeigt
int f(int i);
```

- Globale Variablen von C-Standard-bibliothek benutzt
- Header: <cerrno>
  - Nicht immer! Kann auch Makro sein

Rückgabewerte - Diskussion

```
// Fehler wird durch negativen Rueckgabewert angezeigt
int f(int i);
```

- Nachteile:
  - Mögliche Rückgabewerte werden eingeschränkt
  - Implizit (Funktionsdeklaration nicht selbst-erklärend)
  - Nicht möglich für Konstruktoren/Destruktoren
  - Fehlerzustand muss für jeden Funktionsaufruf einzeln überpüft werden
- Vorteil:
  - Kein zusätzlicher Speicher- oder Rechenzeit-Overhead

Fehlerargument - Diskussion

```
// Fehler wird durch setzen von error auf true
// angezeigt
int f(int i, Error &error);
```

- Nachteil:
  - Fehlerzustand muss für jeden Funktionsaufruf einzeln überpüft werden
- Vorteil:
  - Relativ explizit
  - Rückgabewertebereich wird nicht eingeschränkt

Fehlerargument - Mit Zeigern

- Oder mit Zeigern

```
// Fehler wird durch setzen von error auf true
// angezeigt
int f(int i, Error *error = 0);
```

- Wenn `error == 0` wird kein Fehlercode geschrieben
- Bei `inline`-Funktionen: Optimierungen möglich, wenn mit konstantem Ausdruck 0 aufgerufen
- Oft wird Typ `bool*` benutzt

Globale Variable - Diskussion

```
// Fehler wird durch setzen von globaler Variable
// angezeigt
int f(int i);
```

- Nachteil:
  - Fehlerzustand muss für jeden Funktionsaufruf einzeln überpüft werden
  - Thread-Safety?
  - Implizit, nicht explizit
- Vorteil:
  - Rückgabewertebereich wird nicht eingeschränkt

Exceptions - Überblick

- Exceptions:
  - Explizit (falls gewünscht)
  - Schlüsselwort `throw`
  - Rückgabewertebereich wird nicht eingeschränkt
  - Explizite Fehlerbehandlung nicht nur für einzelne Funktions/Methoden-aufrufe möglich, sondern auch für ganze Blöcke

Exceptions - Überblick

- Exceptions werden "geworfen" oder "geschmissen"
- Mit `throw` kann beliebiger Typ geworfen werden
- Funktion/Methode wird bei `throw` beendet

```
// Fehler wird durch "werfen" von Ausnahme angezeigt
int f(int i) {
  // do something
  // ...
  // oh we need to signal an error:
  throw Error("something went wrong!");
  // ...
}
```

- `Error` ist hier benutzerdefinierter Typ
- Man kann auch Werte eingebauter Typen werfen (`int`, `float`, `double`, `bool`, ...) und auch Zeiger...

Exceptions

- Exceptions sind "alternativer Rückgabewert"
  - Aber der Vergleich hinkt
- Besser: Exceptions stellen Möglichkeit dar, Informationen aus einem Kontext in einen anderen Kontext zu befördern
- Der andere Kontext hat evtl. die nötigen Informationen, die Exception richtig zu behandeln:
- Schlüsselwort `catch`

Exceptions - Beispiel

- Beispiel: Klasse Image hat statische Methode

```
Image *load(const char *fileName);
```

- Es können verschiedene Fehler auftreten
  - Datei kann nicht geöffnet werden
  - Fehlerhafte Daten in Datei (falsche Reihenfolge, fehlende Felder, usw..)
  - Fehler beim Erstellen des Ziel-Image-Objekts (zuwenig Speicher zur Verfügung, usw..)
  - ...

Exceptions Beispiel

- Typ des Fehlerobjekts sollte Fehler anzeigen (damit kann man dann Funktionsüberladung ausnutzen (catch))
- Über Vererbungshierarchie ist dann auch "Default"-Behandlung möglich

```
struct Error { };
struct FileOpenError    : public Error { };
struct FileFormatError  : public Error { };
struct OutOfMemoryError : public Error { };

Image *load(const char *fileName) {
  // ... File could not be opened:
  throw FileOpenError();

  // ...Oh we are out of memory:
  throw OutOfMemoryError();

  // ...File format incorrect:
  throw FileFormatError();
}
```

Exceptions und `auto`-Variablen

- Wir haben damit Möglichkeit, Funktionen/Methoden frühzeitig zu beenden und ein Objekt beliebigen Typs (was will man mehr?) zu "werfen"
- Nützlich: C++ garantiert, dass alle angelegten auto-Variablen "aufgeräumt" werden
- Aber wer fängt die Exception auf und verwertet sie?
  - Schlüsselworte `try` und `catch`...
- Und was ist mit dynamisch angelegten Objekten?

try und catch

- Schlüsselwort try kennzeichnet Anfang von *Block*, der getestet werden soll:

```
try {
  Image *i = Image::load("test.pgm");
}
```

- Der Block kann dabei auch mehrere Anweisungen enthalten
- Auf try folgt immer ein Block
- Aber: Noch kein korrektes C++

try und catch

- Mit Schlüsselwort `catch` wird Exceptionhandler angezeigt

```
try {
  Image *i = Image::load("test.pgm");
}
catch (const Error &e) {
  std::cout << "Oh Oh, something's up!" << std::endl;
}
```

- Alle drei Fehler, die `load` werfen kann, sind abgeleitet von `Error`, daher "passt" der `catch`-Block

try und catch

- Detailliertere Fehlerbehandlung:

```
try {
  Image *i = Image::load("test.pgm");
}
catch (const FileOpenError &e) {
  std::cout << "Failed to open File" << std::endl;
}
catch (const Error &e) {
  std::cout << "Oh Oh, something's up!" << std::endl;
}
```

- Der erste catch-Block der "passt" wird angesprungen
- Exception gilt danach als behandelt

Exceptions - Typen

- Man kann *alle* Typen als Exceptions werfen und fangen
- Auch Zeiger (z.B.: auf mit `new` angelegte Objekte)
- Bei Zeigern Problem: Wer ist für die Löschung zuständig
  - Jeder `catch`-Block, der eine Exception behandelt, muss gefangenes Objekt per `delete` löschen, oder "weiterwerfen"
  - Fehlerträchtig

Faustregel

- **Objekte werfen – Referenzen fangen**
- Polymorphismus funktioniert, und Zerstörung von Objekten passiert automatisch

Catch All - Handler

- "Catch-All-Handler":

```
catch(...) {}
```

- Fängt alle Exceptions
  - Einfacher Weg, um alle Exceptions gleich zu behandeln, ohne explizit auf Typ der Exception zu achten

```
try {
  Image *i = Image::load("test.pgm");
}
catch (...) {
  std::cout << "Oh Oh, something's up!" << std::endl;
}
```

Exceptions weiterwerfen

- Ein `catch`-Block hat Möglichkeit, Exception weiterzuwerfen
- Schlüsselwort `throw`

```
try {
  Image *i = Image::load("test.pgm");
}
catch (const Error &e)
{
  // Hier kann man z.B. die Exception ausgeben
  throw; // analog zu "throw e": wird weiter geworfen
}
catch (...) {
  std::cout << "Oh Oh, something's up!" << std::endl;
  throw; // throw ... geht ja nicht!
}
```

Exceptions weiterwerden

- Jetzt fängt das Spiel von vorne an
  - Die Exception wird weitergeworfen
  - Aktuelle Funktion/Methode wird beendet
  - Erster passender `catch`-Block ausserhalb dieser Funktion/Methode wird angesprungen
  - Es werden wenn nötig mehrere Funktionen beendet
  - Falls kein passender `catch`-Block gefunden wird:
    - Spezielle Funktion void `terminate();` wird aufgerufen

Exceptions weiterwerfen - Beispiel

```
void f() { throw int(1); }

void g() {
  try { f(); }
  catch(const Error &e) { } // handler passt nicht
}

struct X {
  std::string m_str;
  X() { g(); }
};

int main() {
  try { X x; }
  catch (const Error &e)  { } // handler passt nicht
  catch (...) { throw; } // passt, aber wird "rethrown"
}
```

Exceptions weiterwerfen

- In vorherigem Beispiel hatte `X::X()` keinen `try-catch`-Block
- Bedeutet, dass alles "weitergeworfen" wird

Konstruktoren und Exceptions

- Konstruktoren:
  - Falls in einem Konstruktor (in irgendeinem) eine Exception geworfen wird, dann werden alle Elemente, die bis zu diesem Zeitpunkt initialisiert wurden, wieder freigegeben
- Objekte werden "ganz oder gar nicht" initialisiert
- Aber keine dynamisch angelegten Elementvariablen. C++ garantiert hier "nur", dass der Zeiger zerstört wird

Konstruktoren - Beispiel

```
struct Turboloader {
  Turboloader(float pressure){
      if(pressure>10) throw std::runtime_error("oops");
     else ..
   }
};
struct Auto {
  Engine m_engine;
  Turboloader m_turboLoader;
  Wheels *m_wheels;
  Auto() : m_engine("V8"), m_turboLoader(11.0),
        m_wheels(new Wheels(4)) {
  }
};
int main() {
  try {
    // TurboLoader Konstruktor wirft Exception
    Auto krassesGeraet;
  }
  catch (...) { // krassesGeraet existiert hier nicht mehr }
}
```

Konstruktoren - Beispiel

```
struct Auto {
  Engine m_engine;
  Wheels *m_wheels;          // Reihenfolge nun anders
  Turboloader m_turboLoader; // hier auch!
  Auto() : m_engine("V8"), m_wheels(new Wheels(4)),
       m_turboLoader(11.0) {
  }
};

int main() {
  try {
    // TurboLoader() wirft Fehler nachdem fuer m_wheels
    // dynamisch ein Objekt angelegt wurde
    Auto karre;
  }
  catch (...) {
    // karre existiert hier nicht mehr
    // aber das mit new Wheels(4) angelegte Wheels-Objekt
  }
}
```

Freie Funktionen

- Das gleiche Problem mit in Funktionen per `new` angelegten Objekten...

```
void concat_print(const char *a, const char *b) {
  char *s = new char[strlen(a) + strlen(b) + 1];

  if (strcmp(a,"concatenate " == 0)
    throw Error("i feel sick of this!");

  strcpy(s,a); strcpy(s+strlen(a),b);
  std::cout << s << std::endl;
  delete[] s;
}

int main() {
  try { concat_print("concatenate ", "me"); }
  catch (...) {  }
}
```

Dynamischer Speicher - Lösungen

- Lösungen:
  - Aufräumen bevor Fehler geschmissen wird
    - Relativ fehlerträchtig
    - In Initialisierungsliste unmöglich
  - Aufräumen nachdem Fehler geschmissen wurde
    - Sehr schwierig und fehlerträchtig
    - In Initialisierungsliste unmöglich

Dynamischer Speicher - Lösungen

- Lösungen contd.
  - Keine Exceptions werfen, nachdem dynamisch Objekte angelegt wurden
    - Manchmal umständlich oder unmöglich
  - Keinen dynamischen Speicher benutzen
    - aja!
  - Keine Exceptions werfen
    - Wird ja immer schlimmer :)

Smart Pointer - std::auto_ptr

- C++ STL bietet einen *Smart Pointer* zu genau diesem Zweck an
  - Leider aber eben auch nur *für genau diesen* Zweck
- Typ `std::auto_ptr<T>` im Header `<memory>`

```
#include <memory>
#include <stdexcept>

struct Image { ... };

int main() {
 std::auto_ptr<Image>(new Image(640,480));
 throw std::runtime_error("booring!!");
}
```

- Ganz analog in e.g. Konstruktor einer Klasse
- Mehr zu Smart Pointern später

throw - Deklarationen

- Problem: Der Deklaration von `Image::load()` sieht man es nicht an, dass Exceptions geworfen werden
- Schlüsselwort `throw` hat zusätzliche Rolle in Deklaration einer Funktion/Methode:
- Zeigt an, welche Exceptions geworfen werden können (mit Ausnahmen)

```
Image *load(const char *fileName) throw (
  FileOpenError,
  OutOfMemoryError,
  FileFormatError
);
```

throw - Deklarationen - Syntax

- Syntax `throw(Typ1, Typ2, ..., TypN)`
- `throw`-Liste ist Teil der Funktions-Signatur
- Es kann nicht über unterschiedliche `throw`-Listen überladen werden
- Fehlende `throw-Liste`: Everything goes!
- Leere `throw`-Liste am Ende einer Funktionsdeklaration zeigt an, dass keine Exceptions geworfen werden sollten:

```
void f() throw();
```

- Man kann aber trotzdem welche werfen
- Seeeeehr schlechter Stil!

throw - Schlechter Stil

- Nicht nur schlechter Stil
- C++ springt in diesem Fall zu spezieller Funktion void `unexpected();`
- Mit `std::set_unexpected(void(*)());` kann eigene Funktion benutzt werden
  - Liefert alten Funktionszeiger zurück
- Default-Funktion ruft `terminate();` auf

unexpected() - Beispiel

```
void f() throw(X) {
  throw Y();
}

void really_unexpected() {
  std::cout << "Naughty Function!" << std::endl;
}

int main() {
  std::set_unexpected(really_unexpected);
  try {
    f(); // Wirft Y-Objekt als Exception
  }
  catch (...) {
    // Wird trotz "..." nicht gefangen
  }
}
```

- Programm wird trotzdem beendet
- Lösung `std::bad_exception` (Moment!)

Standard-Exception-Typen

- C++-Standardbibliothek bietet Standard-Exception-Typen
- Deklariert im Header <exception>
- Basisklasse std::exception
- Hat Methode

```
virtual const char *what() const throw();
```

- Liefert textuelle Beschreibung des Fehlers
- In diesem Fall: "std::exception"

Standard-Exception-Typen

- Abgeleitete Klassen:
- Header <exception>
  - std::bad_exception
- Header <stdexcept>
  - std::runtime_error
  - std::logic_error
- Header <new>
  - std::bad_alloc
- Header <typeinfo>
  - std::bad_typeid
  - std::bad_cast
- Header <ios_base.h>
  - std::ios_base::failure

Standard-Exception-Typen

- `std::bad_exception`
- Header `exception`
- Wird geworfen, wenn `unexpected()`-Funktion Exception "weiterwirft" und `throw`-Liste der Funktion/Methode `std::bad_exception` enthält
- Kann mit `catch(std::exception& e)` oder ähnlichem gefangen werden

bad_exception - Beispiel

```
void f() throw(X, std::bad_exception) {
  throw Y();
}

void really_unexpected() {
  throw;
}

int main() {
  std::set_unexpected(really_unexpected);

  try {
    f(); /* Wirft Y-Objekt als Exception */
  }
  catch (std::bad_exception &e) {
    // Wird gefangen
  }
  catch (...) { /* usw.. */ }
  std::cout << "life goes on though" << std::endl;
}
```

Standard-Exception-Typen

- Header <typeinfo>
- `std::bad_typeid`
  - wird geworfen, wenn ein `NULL`-Zeiger in einer `typeid`-Expression benutzt wird
- `std::bad_cast`
  - wird geworfen, wenn `dynamic_cast` mit referenzen fehlschlägt

Standard-Exception-Typen

- Es existieren auch Typen, die "für den Benutzer da sind"
- Header `<stdexcept>`
- `std::runtime_error`
  - zeigt Fehler ausserhalb des Programms an (kaputte Netzwerkverbindung, Datei lässt sich nicht öffnen, etc..)
- `std::logic_error`
  - zeigt Fehler in der Programmlogik an

runtime_error, logic_error

- `std::logic_error` und `std::runtime_error` haben Konstruktor

```
std::logic_error(const std::string &);
std::runtime_error(const std::string &);
```

- Mit diesen kann Fehlertext gesetzt werden
- Ist dann über Methode `what()` zugänglich
- Guter Stil, eigene Exceptions von diesen beiden erben zu lassen