UNIVERSITÄT DORTMUND

Wintersemester 2006/07

Einführung in die Informatik für Naturwissenschaftler und Ingenieure

(alias Einführung in die Programmierung)

(Vorlesung)

Prof. Dr. Günter Rudolph

Fachbereich Informatik

Lehrstuhl für Algorithm Engineering

Behandlung von **Ausnahmen** (engl. *exceptions*) im „normalen" Programmablauf:

→ Fehler, die zur Programmlaufzeit entdeckt werden (z.B. Datei existiert nicht)

→ können meist nicht an dieser Stelle im Programm behandelt werden

→ sie können vielleicht auf höherer Programmebene „besser verstanden" werden

→ sie können vielleicht an übergeordneter Stelle „geheilt" werden

Konzept:

Entdeckt eine Funktion einen Fehler, den sie nicht selbst lokal behandeln kann

⇒ dann wirft (engl. *throw*) sie eine Ausnahme mit der Hoffnung, dass ihr direkter oder indirekter Aufrufer den Fehler beheben kann

⇒ aufrufende Funktionen, die den Fehler behandeln können, können ihre Bereitschaft anzeigen, die Ausnahme zu fangen (engl. *catch*)

Vergleich mit anderen Ansätzen zur Fehlerbehandlung:

1. Programm beenden.

   Durch `exit()`, `abort()` ⇒ lästig!
   z.B. Versuch, schreibgeschützte Datei zu beschreiben → Programmabbruch!
   z.B. unzulässig in Bibliotheken, die nicht abstürzen dürfen!

2. Wert zurückliefern, der » `Fehler` « darstellt.

   Nicht immer möglich! Z.B. wenn `int` zurückgegeben wird, ist jeder Wert gültig!
   Wenn möglich, dann unbequem: teste auf `Fehler` bei jedem Aufruf!
   ⇒ Aufblähung des Programmcodes; Test wird leicht vergessen …

3. Gültigen Wert zurückliefern, aber Programm in ungültigen Zustand hinterlassen.

   z.B. in C-Standardbibliothek: Fkt. setzt globale Variable `errno` im Fehlerfall!
   Test auf `errno`-Wert wird leicht vergessen ⇒ gefährliche Inkonsistenzen
   ⇒ Programm in ungültigem Zustand ⇒ Folgefehler verdecken Fehlerursprung

4. Funktion aufrufen, die für Fehlerfall bereitgestellt wurde. ☺

Realisierung in C++

Drei Schlüsselwörter (plus Systemroutinen): `try, throw, catch`

```
try {
  // Code, der Ausnahme vom Typ
  // AusnahmeTyp auslösen kann
}
catch (AusnahmeTyp ausnahme){
  // behandle Ausnahme!
}
```

Wird irgendwo in diesem Block eine Ausnahme vom Typ „AusnahmeTyp" ausgelöst, so wird Block **sofort** verlassen!

Die Ausnahme vom Typ „AusnahmeTyp" wird hier gefangen und behandelt.

Auf `ausnahme` kann im `catch`-Block zugegriffen werden!

```
throw AusnahmeTyp();
```

Erzeugt Ausnahme vom Typ „AusnahmeTyp"

Ausnahmen fangen

```
void Funktion() {
  try {
    throw E();
  }
  catch(H) {
    // Wann kommen wir hierhin?
  }
}
```

E: *exception*

H: *handler* für Typ H

1. H ist vom selben Typ wie E
2. H ist eindeutige öffentliche Basisklasse von E
3. H und E sind Zeigertypen; (1) oder (2) gilt für Typen, auf die sie zeigen
4. H ist Referenz; (1) oder (2) gilt für Typ, auf den H verweist

Weiterwerfen

```
void Funktion() {
  try {
    // Code, der evtl. E() wirft
  }
  catch(E e) {
    if (e.kann_komplett_behandelt_werden) {
      // behandle Ausnahme ...
      return;
    }
    else {
      // rette, was zu retten ist ...
      throw;
    }
  }
}
```

die Original-ausnahme wird weitergeworfen

Übersetzen und Weiterwerfen

```
void Funktion() {
  try {
    // Code, der evtl. E() wirft
  }
  catch(E e) {
    if (e.kann_komplett_behandelt_werden) {
      // behandle Ausnahme ...
      return;
    }
    else {
      // rette, was zu retten ist ...
      throw new Ausnahme(e);
    }
  }
}
```

Übersetzung der Ausnahme in eine andere:

- Zusatzinformation
- Neuinterpretation
- Spezialisierung: einige Fälle schon behandelt oder ausgeschlossen

eine andere Ausnahme wird ausgelöst

Ausnahmehierarchie: Beispiel

```
class MathError {};
class Overflow : public MathError {};
class Underflow : public MathError {};
class DivisionByZero : public MathError {};
```

```
void Funktion() {
  try {
    // u.a. numerische Berechnungen
  }
  catch (Overflow) {
    // behandle Overflow und alles davon Abgeleitete
  }
  catch (MathError) {
    // behandle jeden MathError, der kein Overflow ist
  }
}
```

Reihenfolge wichtig!

Bsp: **Reihenfolge von Exception Handlern und der „Allesfänger“**

```
void Funktion() {
  try {
    // u.a. numerische Berechnungen
  }
  catch (Overflow) { /* … */ }
  catch (Underflow) { /* … */ }
  catch (DivideByZero) { /* … */ }
  catch (MathError) {
    // behandle jeden anderen MathError (evtl. später eingeführt)
  }
  catch (…) {
    // behandle alle anderen Ausnahmen (irgendwie)
  }
}
```

Reihenfolge der `catch`-Handler entgegengesetzt zur Klassenhierarchie

Achtung: Die 3 Pünktchen … im Argument von `catch` sind C++ Syntax!

Was geschieht beim Werfen / Fangen?

Wird Ausnahme geworfen, dann:

1. Die `catch`-Handler des „am engsten umschließenden“ `try`-Blockes werden der Reihe nach überprüft, ob Ausnahmetyp irgendwo passt.
2. Passt ein Ausnahmetyp auf einen der Handler, dann wird er verwendet. ☑
3. Passt kein Ausnahmetyp auf einen der Handler, dann wird die Aufrufkette aufwärts gegangen.
4. Existiert auf dieser Ebene ein try-Block, dann → 1.
5. Existiert kein try-Block, dann wird wird Aufrufkette aufwärts gegangen. → 4.

Falls Ende der Aufrufkette erreicht, dann wurde Ausnahme nicht gefangen!

→ Es wird die Systemfunktion `terminate()` aufgerufen.
Keine Rückkehr zu `main()`!

Ausnahmen im Konstruktor

… wird immer wieder diskutiert!

⇒ Alternative:
keine Ausnahme im Konstruktor,
„gefährliche“ Operationen mit mögl. Ausnahme in einer `Init()`-Funktion

⇒ Problematisch:
wurde Init() schon aufgerufen?
2 x Init()? Methodenaufruf ohne Init()?

```
class A {
protected:
  int a;

public:
  A(int aa) {
    if (aa < 0) throw "< 0";
    a = aa;
  }
};
```

Was passiert denn eigentlich?

Wenn Ausnahme im Konstruktor geworfen wird, dann werden Destruktoren für alle Konstruktoren aufgerufen, die erfolgreich beendet wurden.

Da Objekt erst „lebt“, wenn Konstruktor beendet,
wird zugehöriger Destruktor bei Ausnahme nicht aufgerufen!


```
class Base {
public:
  Base() { cout << "Base in Erzeugung" << endl; }
  ~Base() { cout << "Base stirbt" << endl; }
};

class Member {
public:
  Member() { cout << "Member in Erzeugung" << endl; }
  ~Member() { cout << "Member stirbt" << endl; }
};

class Derived : public Base {
private:
  Member member;
public:
  Derived() { cout << "Derived in Erzeugung" << endl;
    cout << "Throwing ..." << endl; throw "boom!"; }
  ~Derived() { cout << "Derived stirbt" << endl; }
};
```

Ausnahmen im Konstruktor

```
int main() {
  try {
    Derived d;
  }
  catch (char *s) {
    cout << "gefangen: " << s << endl;
  }
}
```

Ausgabe:

```
Base in Erzeugung
Member in Erzeugung
Derived in Erzeugung
Throwing ...
Member stirbt
Base stirbt
gefangen: boom!
```

Destruktor von **`Derived`** wird nicht aufgerufen!

Ausnahmen im Konstruktor

```
class C: public A {
  // …
  B b;
};
```

Achtung! Sonderfall:

Auch wenn Ausnahme im Konstruktor gefangen worden ist, so wird sie automatisch (ohne explizites **`throw`**) weiter geworfen!

```
C::C()
try
  : A( /* … */), b( /* … */)
{
  // leer
}
catch ( … ) {
  // Ausnahme von A oder B
  // wurde gefangen
}
```

Initialisierungsliste auch überwacht!

der gesamte Konstruktor steht im **`try`**-Block

gelingt **`A::A()`**, aber **`B::B()`** wirft ⇒ **`A::~A()`** wird aufgerufen

… man achte auf die ungewöhnliche Syntax!

Ausnahmen im Destruktor

Verlässt eine Ausnahme einen Destruktor,
wenn dieser als Folge einer Ausnahmebehandlung aufgerufen wurde,
dann wird das als Fehler der Ausnahmebehandlung gewertet!

⇒ es wird die Funktion **`std::terminate()`** aufgerufen (Default: **`abort()`**)

Wird im Destruktor Code ausgeführt, der Aussnahmen auslösen könnte,
dann muss der Destruktor geschützt werden:

```
C::~C()
try {
  f(); // könnte Ausnahme werfen
}
catch (…) {
  // Fehlerbehandlung
}
```


Ein Blick zurück: ADT Stack

```
const int maxStackSize = 100;

class Stack {
protected:
  int a[maxStackSize];
  int size;
public:
  Stack();
  void Push(int value);
  void Pop();
  int  Top();
};
```

hier: realisiert mit statischem Feld

entspricht *create:* → *Stack*

mögliche Ausnahmen:

Push	→ Feld schon voll	⇒	Fehlermeldung und Abbruch (exit)
Pop	→ Feld ist leer	⇒	Ignorieren
Top	→ Feld ist leer	⇒	Fehlermeldung und Abbruch (exit)

Ausnahmebehandlung bisher:

Ein Blick zurück: ADT Stack

```
Stack::Stack() : size(0) {
}
void Stack::Push(int value) {
       if (size == maxStackSize) throw "Stack voll";
       a[size++] = value;
}
void Stack::Pop() {
       if (size == 0) throw "Stack leer";
       size--;
}
int Stack::Top() {
       if (size == 0) throw "Stack leer";
       return a[size-1];
}
```


Ein Blick zurück: ADT Stack

```
int main() {
  Stack s;
  try { s.Top(); }
  catch (char *msg) {
    cerr << "Ausnahme : " << msg << endl;
  }

  int i;
  try {
    for (i = 1; i < 200; i++) s.Push(i);
  }
  catch (char *msg) {
    cerr << "Ausnahme : " << msg << endl;
    cerr << "Iteration: " << i << endl;
    cerr << "Top()    : " << s.Top() << endl;
  }
```

Anmerkung: Variable i wird außerhalb des `try`-Blockes definiert, damit man auf sie im `catch`-Block zugreifen kann.

Fortsetzung auf nächster Folie …

Ein Blick zurück: ADT Stack

(… Fortsetzung)

```
  try {
    for (i = 1; i < 200; i++) s.Pop();
  }
  catch (char *msg) {
    cerr << "Ausnahme : " << msg << endl;
    cerr << "Iteration: " << i << endl;
  }
  return 0;
}
```

Ausgabe:

```
Ausnahme : Stack leer
Ausnahme : Stack voll
Iteration: 101
Top()    : 100
Ausnahme : Stack leer
Iteration: 101
```


Noch besser: Verwendung von Fehlerklassen

```
class StackError {
public:
  virtual void Show() = 0;
};
class StackOverflow : public StackError {
public:
  void Show() { cerr << "Stack voll" << endl; }
};
class StackUnderflow : public StackError {
public:
  void Show() { cerr << "Stack leer" << endl; }
};
```

abstrakte Klasse

Vorteile:

1. Differenziertes Fangen und Behandeln durch verschiedene catch-Handler
2. Hinzufügen von Information möglich (auch Mehrsprachigkeit der Fehlermeldung)

Noch besser: Verwendung von Fehlerklassen

```
Stack::Stack() : size(0) {
}

void Stack::Push(int value) {
  if (size == maxStackSize) throw new StackOverflow();
  a[size++] = value;
}

void Stack::Pop() {
  if (size == 0) throw new StackUnderflow();
  size--;
}

int Stack::Top() {
  if (size == 0) throw new StackUnderflow();
  return a[size-1];
}
```

Warum dynamische Objekte (via **new**)? Wg. dynamischer Bindung (**virtual**)!

Noch besser: Verwendung von Fehlerklassen

```
int main() {
  Stack s;
  try { s.Top(); }
  catch (StackUnderflow *ex) { ex->Show(); }   <-- passt
  catch (StackError *ex) { ex->Show(); }
  try { for (int i = 1; i < 200; i++) s.Push(i); }
  catch (StackOverflow *ex) { ex->Show(); }    <-- passt
  catch (StackError *ex) { ex->Show(); }
  try { for (int i = 1; i < 200; i++) s.Pop(); }
  catch (StackOverflow *ex) { ex->Show(); }    <-- passt nicht!
  catch (StackError *ex) { ex->Show(); }       <-- passt
}
```

Ausgabe:
```
Stack leer
Stack voll
Stack leer
```
← wegen dynamischer Bindung!

Noch besser: Verwendung von Fehlerklassen

```
int main() {
  Stack s;
  try { s.Top(); }
  catch (StackError *ex) { ex->Show(); }
  try { for (int i = 1; i < 200; i++) s.Push(i); }
  catch (StackError *ex) { ex->Show(); }
  try { for (int i = 1; i < 200; i++) s.Pop(); }
  catch (StackError *ex) { ex->Show(); }
}
```

Warum nicht so?

Ausgabe:
```
Stack leer
Stack voll
Stack leer
```
☑

Aber: Keine differenzierte Fehlererkennung und –behandlung möglich durch verschiedene **catch**-Handler!

Noch ein Beispiel (war Klausuraufgabe)

Funktion **ReadValue**

- liest Integer aus Datei und liefert ihn als Rückgabewert der Funktion
- gibt einen Fehlercode zurück per Referenz in der Parameterliste
- Fehlercode == 0 → alles OK
- Fehlercode == 1 → Datei nicht geöffnet
- Fehlercode == 2 → bereits alle Werte ausgelesen

```
int ReadValue(ifstream &s, int &errorCode) {
  int value = errorCode = 0;
  if (!s.is_open()) errorCode = 1;
  else if (s.eof()) errorCode = 2;
  else s >> value;
  return value;
}
```

Hauptprogramm öffnet Datei, liest alle Werte aus, addiert sie und gibt Summe aus.

Muss Fehlercodes abfragen und geeignet reagieren.

```
int main() {
  ifstream file;
  int sum = 0, err = 0;
  file.open("data.txt");
  do {
    int v = ReadValue(file, err);
    if (!err) sum += v;
  } while (!err);
  if (err == 1) {
    cerr << "Datei unlesbar!" << endl;
    exit(1);
  }
  file.close();
  cout << "Summe = " << sum << endl;
  return 0;
}
```

Umständlich!

Aufgaben:

1. `ReadValue` mit Ausnahmen
2. `main` anpassen

Version mit Ausnahmen

Fehlerklassen (minimalistisch)

```
class CannotOpenFile { };
class EndOfFile { };

int ReadValue(ifstream &s) {
  if (!s.is_open()) throw CannotOpenFile();
  if (s.eof()) throw EndOfFile();
  int value;
  s >> value;
  return value;
}
```


Version mit Ausnahmen

```
int main() {
  ifstream file("data.txt");
  int sum = 0;
  try {
    while (true) sum += ReadValue(file);
  }
  catch (CannotOpenFile) {
    cerr << "Datei unlesbar!" << endl;
    exit(1);
  }
  catch (EndOfFile) {
    file.close();
  }
  cout << "Summe = " << sum << endl;
  return 0;
}
```

keine Fehlerabfragen mehr in der eigentlichen Programmlogik

Fehler oder sonstige Ausnahmen werden gesondert behandelt