Praxis der Programmierung

Ausnahmefehler, Namensräume, Template-Funktionen und -Klassen

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Ausnahmefehler

Begriff Ausnahmefehler (Exception)

- Ausnahmefehler sind Laufzeitfehler oder logische Fehler

- würden *ohne besondere Behandlung* ein unerwartetes Verhalten oder einen Absturz zur Laufzeit verursachen, *z.B.*
  - Zugriff auf Array- oder Listenelemente ausserhalb des alloziierten Bereichs
  - Zugriff auf Datenelemente von Instanzvariabeln, die auf kein Objekt zeigen
  - Aufruf von Methoden mit Instanzvariabeln, die auf kein Objekt zeigen
  - Division durch 0
  - Öffnen einer Datei, die nicht voranden ist
  - Lesen aus einer Datei, die während des Zugriffs gelöscht wird

- sind (als Klassen) definierte Fehler, die gezielt ausgelöst und abgefangen werden können

Behandlung von Ausnahmefehlern

- Ausnahmefehler können vom Programmierer *abgefangen* werden
  - $\rightsquigarrow$ kein Programmabbruch
  - $\rightsquigarrow$ kontrolliertes, vom Programmierer festgelegtes Verhalten im Ausnahmefall
  - $\rightsquigarrow$ Trennung von normalem Verhalten und Fehlerbehandlung

```
try {
   // Versuch, das Programm fehlerfrei auszufuehren
   //
   // Anweisungen, die einen Fehler ausloesen koennten
}
catch(...) {  // '...' muss dort stehen! (genereller Fehlertyp)
   // Anweisungen, die ausgefuehrt werden,
   // falls ein Ausnahmefehler aufgetreten ist
}
```

Vordefinierte Ausnahmefehler

- als Klassen der Standardbibliothek
- Basisklasse: `exception` (`#include <exception>`)
  - mit Konstroktoren zum Erzeugen von Fehlern, z.B. mit spezifischen Fehlermeldungen
  - Auslösen durch `throw exception();` (oder mit parametrisierten Konstruktoren)
  - mit einer Methode `what()`, liefert einen C-String mit einer Fehlermeldung (Standardmeldung oder durch Parameter eines Konstruktors definiert)
- mehrere abgeleitete Klassen (`#include <stdexcept>`):
  - `runtime_error` – Laufzeitfehler
  - `logic_error` – Logische Fehler
  - einige weitere Standardausnahmen, *z.B.*
    `bad_alloc` – `new` kann keinen Speicher anfordern
    ...

Beispiele für Standardausnahmen

- `ios_base::failure` – Fehlerklasse der Stream-Klassen (`runtime_error`)
- `out_of_range` – Zugriff mit unzulässigem Index (`logic_error`)

```
int atIndex(int arr [], int index) throw(out_of_range) {
  if (index < 0 )
     throw out_of_range("Negativer Index");
  else
     return arr[index];
}
```

```
int main() {
  int a[2] = {0,1};
  atIndex(a,-1);
}
```

$\rightsquigarrow$ *Absturz mit Ausgabe:*

```
terminate called after throwing 'std::out_of_range'
   what(): Negativer Index
```

Abfangen von Standardausnahmen

```
int main() {
   int a[2] = {0,1};
   try {
      atIndex(a,-1);
   }
   catch(...) {
      cout << "Achtung: Index war negativ." << endl;
   }
   cout << atIndex(a,0);
}
```

⇝ *liefert:*

```
Achtung: Index war negativ.
0
```

`throw`-Deklaration an Funktionen

```
int atIndex(int arr [], int index) throw(out_of_range) {
 if (index < 0 )
    throw out_of_range("Negativer Index");
 else
    return arr[index];
}
```

- Deklarieren der Exceptions, die von der Funktion ausgelöst werden können, mehrere ggf. durch Komma getrennt

- `int foo() throw() { ... }`

  $\rightsquigarrow$ Verpflichtung, keine Exception auszulösen

Eigene Ausnahmefehler

- Man kann eigene Unterklassen von `exception` definieren
  $\rightsquigarrow$ Überschreiben der Methode `what()` aus `exception`:

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

- *Beispiel:*

```
class MyException : public exception {
String s;
public:
   MyException(String s) {this->s = s;}
   virtual ~MyException() throw() {}
   virtual const char * what() throw() {
      return s.c_str();
   }
};
```

Unterscheidung von Ausnahemefehlern

- mehrere `catch`-Blöcke zu einem `try`-Block
- erst spezielle Fehler (Vererbungshierarchie beachten!), dann allgemeinere
- `catch(...)` zuletzt (Behandlung "aller weiteren" Ausnahmefehler)

```
try { // Anweisungen }
catch(MyException&) {
      // Behandlung von Fehlern eines selbst definierten Typs
}
catch(ios::failure&)
      // Behandlung von Fehlern bei Stream-Nutzung
}
catch(...) {
      // wenn sonst etwas schief geht
}
```

`throw`: Ausnahmefehler ohne Fehlerklasse erzeugen

```
void foo(int problem) {
   if (problem > 0)
      throw 0;      // erzeugt eine Exception
   // ...
}

int main() {
   try {
      foo(1);
   }
   catch(...) {
      cout << "Ein Fehler beim Aufruf von foo(int)."
   }
```

Übergabe der Fehlernummer?!

`throw`: Ausnahmefehler ohne Fehlerklasse erzeugen

```
void foo(int problem) {
   if (problem == 1)
      throw 1;
   if (problem == 2)
      throw 2;
   if (problem > 2)
      throw (char *) "message";
}

int main() {
   int n = 0;
   cin >> n;
   try { foo(n); }
```

```
catch(int i)   { // Anweisungen fuer int }
catch(char* s) { // Anweisungen fuer char* }
catch(...)     { // Anweisungen sonst }
}
```

Namensräume

Erinnerung: Namensräume

- definieren Bereiche, in denen Namen / Bezeichner eindeutig sein müssen
- in verschiedenen Namensräumen kann der gleiche Name verwendet werden
- *Beispiele*:
  - Telefonnummern in Vorwahlbereichen: `0331 123456`, `030 123456`
  - Telefonnummern mit Vorwahl in Ländernetzen: `+49 30 123456`, `+36 30 123456`
  - Dateinamen in Ordnern; diese in übergeordneten Ordnern, ...
  - in Netzwerken absolute Pfadnamen auf Hosts (Rechnernamen), ...

Erinnerung: Qualifizierte Namen

- unqualifizierte Namen: die Bezeichner selbst

  Beispiel: `meineDatei`

- qualifizierte Namen: mit Angabe des Namensraums

  Beispiel: `/home/rlehre/meineDatei`

- in **C++**: `namensraum::bezeichner`

  Beispiel: `std::cout`, `std::cin`, `std::endl`

- Namensraum `std` enthält Bezeichner der Standardbibliothek

- Definition eigener Namensräume möglich ...

Benutzerdefinierte Namensräume

- Zuordnung von Definitionen zu einem Namensraum:

```
namespace name {
   void function1();     // Definitionen, die dem Namensraum 'name'
   int function2();      // zugeordnet werden
}
```

- Zugriff mit `name::function1();`

  oder durch

```
using namespace name;
// ...
function1();
```

Anonyme Namensräume

- Namensräume ohne Namen: `namespace { // ... }`
- Zugriffe sind auf Funktionen beschränkt, die in derselben Quelltextdatei definiert sind.
- Alternative: alle Funktionen `static` definieren
  - `static` definierte Funktionen werden dem Linker nicht bekannt gegeben
    $\rightsquigarrow$ stehen nur innerhalb der Quelltetdatei mit ihrer Definition zur Verfügung
  - *aber:* statische Methoden von Klassen sind Klassenmethoden

Beispiel

util.h:

```
namespace util {
    void foo();
}
```

bsp.cpp:

```
#include "util.h"
#include <iostream>
using namespace std;
namespace {
    void myFunction(int a) {
        cout << a << endl;
    }
}
void util::foo() {
    myFunction(4);
}
```

Template-Funktionen

Minimumfunktion und offene Typen

- Aufruf `min(x,y)` sollte für möglichst alle Datentypen funktionieren, für die eine Ordnungsrelation definiert ist

- Keine Code-Verdopplung! *Aber:*
  - Überladen der Funktion ungünstig

  $\rightsquigarrow$ Funktion mit offenem Typ definieren

- Template-Funktionen
  - verwenden offene Typen
  - definieren eine Schablone einer Funktion, die typunabhängiges Verhalten hat
  - Typen werden beim Aufruf der Funktion festgelegt

    $\rightsquigarrow$ Compiler erzeugt aus der Schablone eine zu den Typen passende Funktion

Beispiel: Minimumfunktion

```
template <typename T>   // eine Schablone mit dem Typparameter T
T min(T a, T b) {       // T kann jetzt wie ein Typ verwendet werden

     return (a < b) ? a : b;

}

int main() {
   int i = 19;
   int j = 66;
   int a = min(i,j);  // Funktion mit T = int wird jetzt erzeugt
}
```

- Statt `typename` kann synonym auch `class` verwendet werden.
- mehrere Typparameter durch Komma getrennt, *z.B.* `<class T, class S>`

Festlegung der Zieltypen

- entweder durch Parameterübergabe
- oder durch explizite Angabe der Zieltypen:

```
template <typename T1, typename T2, typename T3>
T3 foo(T1 x, T2 y) {
   // ...
}

int main() {
   cout << foo<short,long,int>(23,11);  // Rueckgabetyp int
}
```

swap als Template

```
template <typename T> void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

Template-Klassen / generische Typen

Definition von Template-Klassen

```
// Schablone einer Klasse mit einem Typparameter T
template <typename T> class Cls {
   // jetzt kann T wie ein Datentyp verwendet werden, z.B.:
   T var;
   T foo(int n, T& t);
};

template <typename T> T Cls<T>::foo(int n, T& t) {
   // ...
}

int main() {
   Cls<int> instance;  // Cls<int> ist ein generischer Typ
                       // Compiler legt generischen Typ erst bei der
   // ...              // Definition eines Objekts an
}
```