

Programmiertechnik (C++)

Prof. Dr. Stefan Enderle
NTA Isny

6. Exceptions

6.1 Das Problem

- Einführungsbeispiel:

Aufgabe:

Schreiben Sie eine Funktion, die eine Quadratwurzel berechnet.

- (Bemerkung: Was ist schlecht an der Fragestellung?)

Das Problem

- 1. Implementierung:

```
double squareRoot(double value)
{
    return sqrt(value);
}
```

- Problem:
 - Kein Typ und Wertebereich in der Aufgabenstellung spezifiziert!
 - Annahme: „Gutfall“ also $value \geq 0$!
 - Was passiert bei $value < 0$?

1. Lösung

- 1. Lösung: Fehlerbehandlung in der Funktion selber

```
double squareRoot(double value)
{
    if (value < 0)
    {
        cerr << "Wert kleiner 0!" << endl;
        exit(-1);
    }
    return sqrt(value);
}
```

- Probleme?

1. Lösung

- Probleme der 1. Lösung:
 - Beendigung des Programmes nicht immer erwünscht. Evtl. lieber Fehlermeldung, Nochmal-Eingeben, usw.
 - Ausgabe auf cerr nicht immer möglich. Z.B. in Windows-Programm besser Dialogbox.
- Anwender der Funktion will selber entscheiden, was im Fehlerfall passieren soll.

2. Lösung

- 2. Lösung: Rückmeldung eines Fehlercodes

```
const double INVALID = -1.0;
```

```
double squareRoot(double value)
{
    if (value < 0) return INVALID;

    return sqrt(value);
}
```

- Probleme?

2. Lösung

- Probleme der 2. Lösung:
 - Finden von Fehlercodes ist manchmal schwierig, z.B. wenn ALLE Werte zulässig sind.
 - Alle Fehlercodes müssen bekannt sein und vom Anwender der Funktion abgefragt werden.
 - Eine Änderung der Bibliothek kann eine Änderung der Anwendung zur Folge haben.

Verschärfte 2. Lösung

- Verschärfte 2. Lösung:
Old-style Fehlercodes und umständliche Rückgabe

```
const int INVALID = -1;
```

```
int squareRoot(double value, double* result)
{
    if (value < 0) return INVALID;

    *result = sqrt(value);
    return 0;
}
```

Verschärfte 2. Lösung

- Probleme der verschärften 2. Lösung:
 - „Rückgabewert“ ist kein Rückgabewert.
 - Alle Fehlercodes müssen bekannt sein und vom Anwender der Funktion abgefragt werden.
 - Eine Änderung der Bibliothek kann eine Änderung der Anwendung zufolge haben.
 - Umständlicher Aufruf:

```
double result;  
int error = squareRoot(3.7, &result);  
if (error!=0) ...
```

Anforderungen

- Anforderungen an eine saubere Fehlerbehandlung:
 - Melden einer Fehlersituation soll unabhängig sein von normalem Funktionsaufruf mit Parameter-Übergabe und -Rückgabe.
 - Das Auftreten eines Fehlers soll nicht durch Unachtsamkeit übersehen werden können.
 - Das Auftreten eines Fehlers soll jedoch explizit ignoriert werden können.

Lösung: Exceptions

- Exceptions (Ausnahmen) bieten im Fehlerfall einen parallelen Ausstieg aus der Funktion:
 - Normalfall: `return` → Rückgabewert
 - Fehlerfall: „Werfen“ einer Ausnahme
ggfs. mit Übergabe von Daten
- Konzept: Ausnahmen werden durch einen – vom normalen Code unabhängigen - „Behandler“ (Handler) abgearbeitet.

Beispiel

- Beispiel: „Werfen einer Ausnahme“:

```
double squareRoot(double value)
{
    if (value < 0)
        throw "Wert kleiner 0";

    return sqrt(value);
}
```

throw

- Aufruf: `throw <wert / objekt>`
- Beendet die Abarbeitung der Funktion sofort und übergibt den geworfenen Wert oder das Objekt direkt einem Exception-Handler.

try / catch

- Exception-Handler:

```
try {  
    a = sqareRoot(1.0);  
    b = sqareRoot(-1.0);  
    c = sqareRoot(2.0);  
}  
catch (const char* exc)  
{  
    cerr << "Fehler: " << exc << endl;  
    exit(1);  
}
```

try / catch

- Der Exception-Handler ist keine Funktion, sondern ein Block, der direkt dem try-Block folgt.
- Der try-Block beinhaltet den Code, dessen Exceptions gefangen werden sollen.
- try-Block und catch-Block stehen immer direkt hintereinander.

Ablauf

- ```
try {
 a = sqareRoot(1.0);
 b = sqareRoot(-1.0);
 c = sqareRoot(2.0);
}
catch (const char* exc)
{
 cerr << "Fehler: " << exc << endl;
 exit(1);
}
```
- Aufruf 1 – ganz normal
- Aufruf 2 – Werfen der Ausnahme → Ausgabe, exit
- Aufruf 3 wird nicht mehr ausgeführt.

# Anmerkungen

- Bei der Implementierung einer Funktion muss man sich keine Gedanken machen, WIE auf ein Problem reagiert wird. Man muss lediglich die Fehler erkennen und melden (also Ausnahmen werfen).
- Die Funktion `squareRoot` kann sogar übersetzt werden, ohne dass bekannt ist, vom wem sie später aufgerufen wird, also welcher Exception-Handler verantwortlich sein wird.
- Im normalen Programm kann man sich dem eigentlichen Problem kümmern und „sich sicher sein, dass kein Fehler passiert“. Dieser würde ja in einen Handler laufen.

# Achtung

- Man darf nicht davon ausgehen, dass ALLE Anweisungen in einem try-Block auch wirklich ausgeführt werden!
- Beispiel:

```
try {
 A* ap = new A; // lege Objekt an
 f(ap); // tue etwas damit
 delete ap; // lösche wieder
}
```

- Bei Ausnahme in f(ap) wird delete nicht mehr aufgerufen!  
→ Handler muss sich um das Aufräumen kümmern!

# throw-catch über mehrere Funktionen

- Beispiel:

```
void g()
{
 throw 21;
}
```

```
void f()
{
 g();
}
```

```
void h()
{
 try { f(); }
 catch (int) { exit(1); }
}
```

# throw-catch über mehrere Funktionen

- Trotz Verschachtelung wird der Handler hinter dem *aktiven* try-Block stets gefunden:
  - Steht der aktuelle Code nicht direkt in einem try-Block wird die aufrufende Funktion untersucht.
  - Wird irgendwann ein umschließender try-Block gefunden, wird dessen Handler aufgerufen.
  - Falls nicht, wird das Programm beendet.

# throw-catch über mehrere Stufen

- Beispiel (selten):

```
try
{
 try
 {
 throw 21;
 }
 catch (int)
 {
 cout << "catch 1" << endl; // wird aufgerufen
 }
}
catch (int)
{
 cout << "catch 2" << endl;
}
```

# throw-catch über mehrere Funktionen

- Beispiel (häufig):

```
void g()
{
 throw 21;
}
```

```
void f()
{
 try { g(); }
 catch (int) { cout << "catch 1" << endl; } // auch!
}
```

```
void h()
{
 try { f(); }
 catch (int) { cout << "catch 2" << endl; }
}
```

# Typisierte Ausnahmen

- throw wirft immer ein Objekt eines bestimmten Typs.
- Ein Handler ist dann *qualifiziert*, wenn der Typ in seinem catch-Block
  - exakt der geworfene Typ ist, oder
  - eine implizite Konvertierung erfolgen kann.

```
try {
 throw 21;
}
catch (int) { // OK
 exit(1);
}
```

# Typisierte Ausnahmen

- Beispiele:

```
try {
 throw 21;
}
catch (int) { // OK
 exit(1);
}
```

```
try {
 throw 21;
}
catch (long) { // OK
 exit(1);
}
```

# Typisierte Ausnahmen

- Beispiel:

```
try {
 throw 21;
}
catch (const char*) { // fängt 21 nicht!
 exit(1);
}
```

- Hier wird der catch-Block ignoriert und die Suche nach einem qualifizierten catch-Block geht weiter.

# Typisierte Ausnahmen

- Mehrere Catch-Blöcke mit unterschiedlichen Exception-Typen sind möglich.
- Beispiel:

```
try {
 throw 21;
}

catch (const char*) { // fängt 21 nicht!
 exit(1);
}
catch (int) { // fängt 21
 exit(1);
}
```

# "Catch All"

- Als Platzhalter für beliebige Typen kann "..." verwendet werden.
- Beispiel:

```
try {
 throw 1.75;
}

catch (const char*) {
 exit(1);
}
catch (int) {
 exit(1);
}
catch (...) { // fängt 1.75
 exit(1);
}
```