Notizen AIK ProgTech 3: Exceptions (“Ausnahmen”)

Klaus Kusche

Sinn und Zweck:

- Vor allem zur *Fehlerbehandlung* und Handhabung außergewöhnlicher Ereignisse (z.B. Speicher voll, I/O-Fehler, math. Fehler, falsche Eingabe, ...) im Programm.
- *Springt* (an der normalen Programmlogik vorbei, auch mit *Return aus Funktionen!*) *direkt* von der *Erkennung* des Fehlers (**`throw`**) zur *Behandlung* des Fehlers (**`catch`**) und kann dabei auch *beliebige Daten mit übertragen* (Fehlerwerte / Fehlermeldungen / ...).

  ==› *Erspart* zusätzliche **`if`**'s für “Notausgänge”, separate Fehler-Returnwerte, ...

  ==› Kann mehr Information übertragen als ein einfacher Returnwert.

  ==› Funktioniert auch an Programmstellen, wo man kein normales Fehler-Return machen könnte (z.B. bei *Fehlern in Konstruktoren oder Destruktoren*).

Das **`throw`** “wirft” eine Exception:

Bei einem **`throw`** wird sofort zum passenden **`catch`** gesprungen,
der Code unmittelbar nach dem **`throw`** wird *nicht* mehr ausgeführt
(für auf dem Weg zum **`catch`** freigegebene lokale Objekte
wird allerdings noch ordnungsgemäß der *Destruktor* ausgeführt).

Ein **`throw`** steht fast immer in einem **`if`**, das die Fehlerbedingung prüft.

Nach dem **`throw`** kann ein *beliebiger Wert* stehen,
nämlich der Wert, der zum **`catch`** übertragen wird (ähnlich dem Wert beim **`return`**):

- Ein **`int`**, ein String, ...
- Ein *Objekt* einer eigenen oder vordefinierten Klasse:
  - Man macht üblicherweise direkt im **`throw`** einen *Konstruktor-Aufruf.*
  - Man macht üblicherweise *kein* **`new`** im **`throw`**:
    - *Ohne* **`new`** wird ein *temporäres Objekt* erzeugt, das Objekt selbst geworfen, und nach dem **`catch`** *automatisch* wieder gelöscht ==› Gut!
    - *Mit* **`new`** wird ein *dynamisch angelegtes Objekt* erzeugt und ein *Pointer* darauf geworfen (man muss daher im **`catch`** einen Pointer fangen!). Das Objekt wird *nicht* automatisch gelöscht, man muss es im oder nach dem **`catch`** selbst freigeben ==› Unnötig kompliziert!

  Üblicherweise werden Objekte von Klassen geworfen, die von der *vordefinierten Klasse* **`exception`** *abgeleitet* sind (das muss aber nicht sein!).
  `exception` definiert eine virtuelle Methode **`what()`**, die abgeleitete Klassen so implementieren sollten, dass sie als Returnwert einen *Fehlertext* liefert.

- *Achtung:*
  Keinen Pointer auf ein bestehendes *lokales Objekt oder Array* werfen! (der Pointer kommt zwar im **`catch`** an, aber das Objekt, auf das er zeigt, gibt es dann nicht mehr…)

Manche *vordefinierte Funktionen* machen bei Fehlern *intern ein* **`throw`**,
es gibt dafür einige vordefinierte, von **`exception`** abgeleitete Fehlerklassen!

`try` & **`catch`** : Das **`catch`** "fängt" eine Exception:

- Der gesamte Code innerhalb von **`try`** (inklusive aller aufgerufenen Funktionen) wird normal ausgeführt und dabei *auf Exceptions geprüft*.
- Tritt *keine Exception* auf, werden die **`catch`** alle *übersprungen*:
  Das Programm macht *nach* dem **`try`** + **`catch`** normal weiter.
- Tritt eine *Exception* auf (d.h. wird irgendwo ein **`throw`** ausgeführt),
  wird das *erste passende* **`catch`** gesucht und ausgeführt.
  Dann macht das Programm *nach* dem **`try`** + **`catch`** normal weiter
  (es wird *nicht* unmittelbar nach dem **`throw`** weitergemacht,
  d.h. *nicht* an der Stelle, wo der Fehler aufgetreten ist!).

  Reihenfolge beim Suchen des **`catch`**:

  - Die das **`throw`** umgebenden **`try`** werden *von innen nach außen* geprüft
    (in umgekehrter Schachtelungs-Reihenfolge innerhalb einer Funktion,
    in umgekehrter Aufrufsreihenfolge von der gerade laufenden Funktion
    zurück bis zum **`main`**).
  - Bei jedem **`try`** werden die **`catch`** (falls es mehrere gibt)
    von oben nach unten geprüft.
- Nach dem **`catch`** steht eine ganz normale *Parameter-Deklaration*, z.B.:
  `catch(const char *msg)`

  1.) Der *Typ* in dieser Deklaration entscheidet, ob dieses **`catch`**
  zum geworfenen Wert *passt*, d.h. ob dieses **`catch`** *"zuständig"* ist.
  2.) Wenn ja: Der *gefangene Wert* wird im Parameter (hier in **`msg`**) gespeichert.

  Achtung:
  Der Typ muss genau passen! Werden z.B. String-*Konstanten* **`"…"`** geworfen,
  so muss der Parameter **`const`** deklariert sein, sonst fliegt die Exception vorbei!

  Bei Objekten wird normalerweise eine **`const`**-*Referenz* gefangen
  (das ist effizienter und sicherer als ein "by Value"-Objekt-Parameter).

  Ein **`catch`** mit Objekt-Parameter, Objekt-Referenz-Parameter oder
  Objekt-Pointer-Parameter fängt auch *Objekte abgeleiteter Klassen*.
- Als *letztes* kann ein **`catch(...)`** stehen (wörtlich "**`...`**").
  Es fängt *alle* Exception, aber man kann *nicht* auf den gefangenen Wert zugreifen.
- Wenn *kein passendes* **`catch`** gefunden wird ("ungefangene Exception"),
  wird das Programm bei einer Exception *mit einer Fehlermeldung abgebrochen*.

Im Praktikum nicht gefordert:

- **Das Weiterwerfen von Exceptions:**

  In einem **`catch`** kann der aktuell gefangene Fehler mit einem **`throw`** *ohne* Angabe eines Wertes nochmals geworfen werden, damit ein weiter außen liegendes zweites **`catch`** ihn *nochmals* fangen und bearbeiten kann.

- **Das Deklarieren von Exceptions:**

  Achtung:
  Dieser Mechanismus wurde mit dem C++-11-Standard als *veraltert* erklärt!

  Im Prototyp jeder Funktion / Methode kann *nach den Parametern* angegeben werden, was die Funktion alles werfen kann.

  Sinn:

  - Für den *Leser* des Programms: Was kann da alles geflogen kommen? Welche **`catch`** sollte man rund um die Funktion machen?
  - Für den *Compiler*: Er kann ev. besser optimierten Code erzeugen.
  - Für die *Programmsicherheit*: Wirft die Funktion eine andere als die angegebenen Exceptions, wird das Programm automatisch mit einer Fehlermeldung abgebrochen!

  Möglichkeiten:

  - Keine Angabe: Es kann *alles* geflogen kommen.
  - **`throw()`**: Die Funktion wirft sicher *keine Exceptions*.
  - **`throw(`***typ1***`,`** ***typ2***`,` …`)`**: Die Funktion darf nur *Werte der angegebenen Typen* (bzw. Objekte der angegebenen Klassen) werfen.

  Seit C++11:
  Ersetzt durch **`noexcept`** : **`noexcept`** bedeutet, dass die Funktion *nichts wirft*, d.h. dass keine Exceptions aus der Funktion herauskommen (kommt trotzdem eine, wird das Programm abgebrochen).