C++ FOR (Donnerstagnachmittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

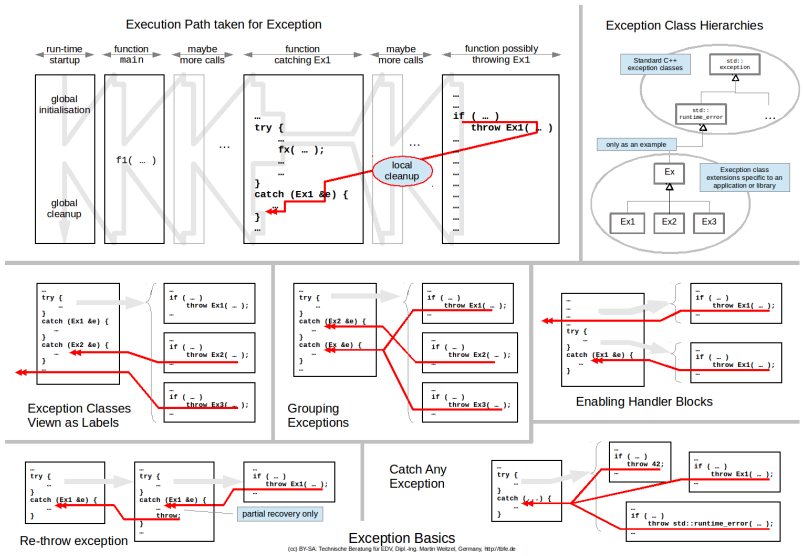
Grundlegendes zu Exceptions

- Hierarchien von Exception-Klassen

- Kontrollfluss mit und ohne Exceptions

- Exception Klassen als "Label" verstanden
- Gruppieren ähnlicher Exceptions
- Aktivieren eines Behandlungs-Blocks

- Unvollständig behandelte Exceptions
- Fangen aller Exceptions



Hierarchien von Exception-Klassen

Die im Rahmen von Bibliotheksfuntionen ggf. geworfenen Exceptions bilden eine Klassenhierarchie:

- An deren Spitze steht die Klasse `std::exception`.
- Weitere Klassen sind oft als Basisklassen für eigene Exception-Klassen sinnvoll, etwa
  - `std::logic_error` oder
  - `std::runtime_error`.

Click here for Info-Graphic
ExceptionBasics
Execution Path taken for Exception

Kontrollfluss mit und ohne Exception

Solange keine Exceptions geworfen werden,

- folgt der Kontrollfluss den üblichen Regeln, und
- alle auf `try`-Blöcke folgenden `catch`-Blöcke werden übersprungen.

Exception Klassen als "Label" verstanden

Sobald eine `throw`-Anweisung ausgeführt wird, stellen die den aktiven `try`-Blöcken nachfolgenden `catch`-Blöcke eine Art Label dar:

- Verzweigt wird grundsätzlich im Kontrollfluss rückwärts, also in Richtung auf die `main`-Funktion.
- Die Auswahl der `catch`-Blöcke erfolgt dabei
  - gemäß der dynamischen Schachtelung der Funktionsaufrufe, und
  - dort ggf. für die einen einzelnen `catch-`Blöcke` von oben nach unten.
- Der erste `catch`-Block, bei dem der Typ des "Arguments" der `catch`-Anweisung zum Typ der geworfenen Exception passt, wird gewählt.

Gruppieren ähnlicher Exceptions

Bei der Auswahl des `catch`-Blocks werden in Bezug auf den Typ der geworfenen Exceptions auch mögliche Umwandlungen berücksichtigt:

- Prinzipiell gelten die selben Umwandlungen wie bei der Parameterübergabe im Rahmen von Funktionsaufrufen.
- Insbesondere gilt das LSP:
  - abgeleitete Klassen passen zu ihren jeweiligen Basisklassen.
- Dies macht es sinnvoll, ähnliche Exceptions mittels kleiner oder größerer Klassenhierarchien zusammenzufassen.

Aktivieren der Ausnahmebehandlung

Als Sprungziel in Frage kommen die **aktiven** `catch`-Blöcke.

- Das sind alle `catch`-Blöcke,
  - deren vorangehender `try`-Block vom Kontrollfluss erreicht ...
  - ... und noch nicht wieder verlassen wurde.

Ein `try`-Block ist nicht mehr aktiv, wenn er explizit verlassen wird durch

- `return`
- `break`
- `continue`

oder die letzte enthaltene Anweisung vollständig ausgeführt wurde.

Ab diesem Moment werden auch die nachfolgenden `catch`-Blöcke nicht mehr als "Sprungziel" für ein `throw` in Betracht gezogen.

Unvollständig behandelte Exceptions

Catch-Blöcke, welche nur eine teilweise Auflösung der Fehlersituation leisten, können die gefangene Exception erneut werfen:

```
    try {
        …
        … // code that may throw SomeException
        …
    }
    catch (SomeException &ex) {
        …
        … // partial recovery only
        …
        throw;
    }
}
```

Fangen aller Exceptions

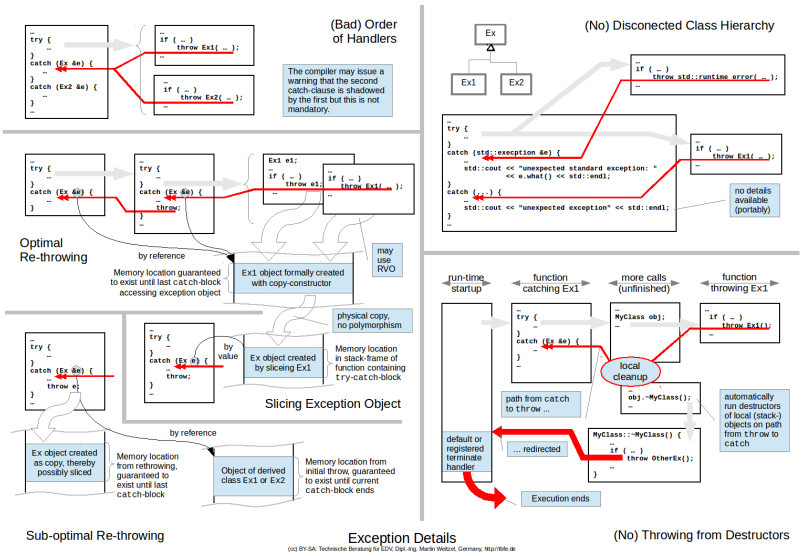
Es besteht die Möglichkeit, einen `catch`-Block für alle erdenklichen Exception "passend" zu machen:

- Hierzu sind in den runden Klammern drei Punkte (analog zu variadischen Funktionen) anzugeben.
  - Ein solcher Block sollte immer am Ende aller `catch`-Blöcke zu einem `try`-Block stehen.
  - Andernfalls gilt er vorrangig zu spezifischeren, nachfolgenden `catch`-Blöcken des selben `try`-Blocks.

```
int main() {
    try {
        …
        …
    }
    catch (...) {
        std::cerr << "!! unhandled exception !!\n";
    }
}
```

Richtlinien für den Exception-Einsatz

- (Keine) isolierte Klassenhierarchie
- (Falsche) Behandlungsblock-Abfolge
- Optimale Weiterleitung
  - (Rückschnitt auf Basisklasse)
  - (Sub-optimale Weiterleitung)
- Exceptions in Destruktoren vermeiden



(Keine) isolierte Klassenhierarchie

Bilden eigene Exceptions eine isolierte Klassenhierarchie, so kann (sehr allgemeiner) Code diese **nicht** über Standard-Exceptions abfangen:

```
int main() {
    using namespace std;
    try {
        …
        … // ordinary application
        …
        return EXIT_SUCCESS; // macro in <cstdlib>
    }
    catch (exception &e) {
        cerr << "terminated by standard exception: "
             << e.what() << endl;
    }
    catch (...) {
        cerr << "terminated by unknown exception" << endl;
    }
    return EXIT_FAILURE; // macro in <cstdlib>
}
```

(Falsche) Behandlungsblock-Abfolge

Werden die `catch`-Blöcke für Exceptions falsch angeordnet,

- kann ein weiter oben stehender, allgemeiner Block
- einen weiter unter stehenden, spezifischeren Block unwirksam machen.

Optimale Weiterleitung

Das Weiterwerfen eines gefangenen Exception-Objekts mit

```
…
catch (SomeException &ex) {
    …
    // only partial recovery
    throw ex;
}
```

führt zur Erstellung einer Kopie **im Rahmen der Anweisung `throw ex;`**

> Mit `throw;` kann das ursprünglich gefangene Exception-Objekt **ohne Kopieren** weitergeworfen werden.

(Vermeidbarer) Rückschnitt auf Basisklasse

Durch (vermeidbares) Kopieren des Exception-Objekts können geworfene Exceptions einer abgeleiteten Klasse per Slicing auf die Basisklasse reduziert werden.

Dies ist im Allgemenen nicht wünchenswert.

Daher sollte ein Kopieren vermieden werden:

- Argumente von `catch`-Blöcken als Referenz spezifizieren.
- Bei nur teilweise Auflösung Exception mit `throw;` weiterwerfen.

Sub-optimale Weiterleitung

Sub-optimal ist grundsätzlich jede Weiterleitung, bei der das Exception-Objekt unnötig kopiert wird.

(Keine) Exceptions aus Destruktoren werfen

[!] Exceptions, die von Destruktoren geworfen werden, können ein Programm abbrechen, wenn der Destruktor im Rahmen eines Exception-Handlings ausgeführt wird.

Falls ein Destruktor Operationen benutzen muss, die u.U. eine Exception auslösen, sollte der Code sicherheitshalber in einen `try`-Block verpackt werden, der alle Exceptions fängt und ignoriert:*

```
class MyClass {
    …
    ~MyClass() {
        try {
            … // whatever needs be done
        }
        catch (...) {
            if (!std::uncaught_exception()) throw;
        }
    }
};
```

*: Ob es tatsächlich sinnvoll ist, die Exception wie gezeigt weiterzuwerfen, falls der Destruktor **nicht** im Rahmen eines Exceptions-Handlings abläuft, muss je nach Sachlage individuell entschieden werden.

`throw`-Spezifikationen vermeiden

Nachdem über viele Jahre hinweg von C++-Experten praktisch einhellig empfohlen worden war, die seinerzeit von C++98 eingeführten `throw`-Spezifikationen zu vermeiden, wurden diese nun mit C++11 zum *deprecated feature*.

Exception-Freiheit mit `noexcept` ankündigen

Neu eingeführt wurde mit C++11 die `noexcept`-Qualifikation für Funktionen, welche folgendes bewirkt:

- Wirft eine damit qualifizierte Funktion dennoch eine Exception, kommt es zum unwiderruflichen Prgrammabbruch (nach Durchlaufen eines frei festzulegenden Handlers).
- Andere Funktionen betrachten eine mit `noexcept` qualifzierte Funktion als *aufrufbar ohne Risiko, dass es zu einer Exceptions kommt*.*

Insgesamt liegen mit der Umsetzung von `noexcept` durch gängige Compiler noch zu wenige Erfahrungen vor, um eindeutige Richtlinien und Empfehlungen zur Benutzung dieses Features geben zu können.

*: Dies ist insofern richtig, als Exceptions – treten Sie dennoch auf – prakisch unmittelbar zum Programmabbruch führen, d.h. im Kontext des Aufrufs muss die Möglichkeit von Exceptions bei solchen Funktionen in der Tat nicht berücksichigt werden ... dennoch wird u.U. das Programm abgebrochen.

Bedingte Exception-Freiheit mit noexcept

Über das auf der vorhergenden Seite gesagte hinaus ist `noxecept` auch eine Compilezeit-Funktion, mit welcher – insbesondere in Templates – die **bedingte Freiheit Exceptions** deklariert werden kann.

- Hiermit kann eine Funktion zum Ausdruck bringen, dass Exceptions nur dann auftreten werden, wenn bestimmte aufgerufene Funktinen solche werfen.
- Dadurch kann zur Compilezeit bei der Instanziierung von Templates mit konkreten Typen entschieden werden, ob bei einer bestimmten Operation das Risiko von Exceptions besteht.
- Dies eröffnet letztlich die Möglichkeit, dass Bibliotheksfunktionen über Template-Metaprogrammierung unterschiedliche Implementierungen wählen.

Übung

Ziel der Aufgabe:

Erweiterung einer Klasse mit einer Fehlerbehandlung unter Nutzung von Exceptions.*

> Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.

*: Diese Aufgabe bietet über die Anwendung von Exceptions hinausgehend auch noch die Möglichkeit, entweder mit Hilfe virtueller Member-Funktionen oder mit Hilfe von Templates eine besonders flexible Lösung zu schaffen.