VL7: Ausnahmebehandlung in C++

Inhalt

1. Einleitung

C++ bietet gegenüber der Sprache C die Möglichkeit, die Behandlung von Ausnahmen von dem normalen Programmablauf sauber zu trennen. Dies trägt erheblich zur Klarheit des Programms bei.

Dabei ist wichtig zu wissen, dass die Ausnahmebehandlung erst sehr spät --ab 1989-- zu C++ hinzugefügt worden ist. Dadurch musste das ursprüngliche Compilerkonzept vollständig überarbeitet werden.

2. Schlüsselworte 'try', 'catch', 'throw'

Die Grundidee der Ausnahmebehandlung besteht darin, dass man zwischen dem Auftreten einer Ausnahme und deren Behandlung trennt. Derjenige Bereich, in dem Ausnahmen auftreten können, wird durch einen *try-block* umschlossen:

```
try{ ... }
```

Die Behandlung möglicher Ausnahmen wird dann ausserhalb des try-Blocks in einer Folge von *catch-Anweisungen* geleistet:

```
catch( ARGUMENT_1 ){ .... }
 ...
catch( ARGUMENT_k ){ .... }
```

Die Argumente der catch-Anweisungen müssen mit dem *Ausnahme-Typ* übereinstimmen. Ähnlich wie bei if-Anweisungen wird der Block derjenigen catch-Anweisungen ausgeführt, deren Argument mit einer aktuellen Ausnahme übereinstimmt.

Ausnahmen können entweder im Rahmen der Sprache C++ *vordefiniert* generiert werden oder aufgrund einer *throw-Anweisung* des Benutzers.

Im folgenden Beispiel except0.cpp wird von der Tatsache Gebrauch gemacht, dass es in C++ vordefinierte Fehlerklassen gibt (siehe unten), die aktiviert werden, wenn entsprechende Fehler auftreten. Im konkreten Beispiel gibt es zwei Möglichkeiten, vordefinierte Ausnahmen zu generieren: (i) das Ablegen eines Speicherbereiches für ein String-Objekt misslingt (bad_alloc) oder (ii) der Zugriff auf einen Index misslingt, da dieser Index im String-Objekt garnicht vorhanden ist (Ausnahme 'out_of_range', die über 'exception' abgefangen wird).

```
//*********************************
//
// except0.cpp
//
// author: gerd doeben-henisch
//
// idea: simple exception-demo
//
// Compilation: g++ -o except0 except0.cpp
// Usage: except0
//
//*****************************************

#include <iostream>    // Headerdatei für I/O
#include <string>      // Headerdatei für Strings
#include <cstdlib>     // Headerdatei fuer EXIT_FAILURE
#include <exception>   // Headerdatei für Ausnahmen

int main()
{
    try {
        std::string vorname("gerd");      // kann std::bad_alloc auslösen

        vorname.at(15) = 'B';    // löst std::out_of_range aus

    }
    catch (const std::bad_alloc& e) {
        // Spezielle Ausnahme: Speicherallokierung misslungen
        std::cerr << "Speicherallokierung misslungen" << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }
    catch (const std::exception& e) {
        // sonstige Standard-Ausnahmen
        std::cerr << "Standard-Exception: " << e.what() << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }
    catch (...) {
        // alle (bisher nicht behandelten) Ausnahmen
        std::cerr << "unerwartete sonstige Exception" << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }

    std::cout << "OK, bisher ging alles gut" << std::endl;
}
```

```
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> g++ -o except0
except0.cpp
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except0
Standard-Exception: pos >= length ()
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12>
```

Im letzten Beispiel ist speziell hervorzuheben die catch-Anweisung *catch(...){ ... }*, die statt eines expliziten Parameters nur drei Punkte enthält; dies bewirkt, dass alle Ausnahmen, die zuvor noch nicht behandelt werden konnten, hier behandelt werden. Dies eröffnet die Möglichkeit (siehe unten), dass der Benutzer eine einzige catch-Anweisung benutzt, um dann alle auftretenden Ausnahmen zu behandeln. Bevor dies vorgeführt wird, hier ein weiteres einfaches Beispiel except1.cpp.

Hinzuweisen ist auch noch auf die Elementfunktion '*e.what()*'; diese gibt einen vordefinierten String aus, der die aufgetretene Ausnahme kommentiert; der String ist implementierungsabhängig, d.h. Compilerspezifisch.

Eine durch den Benutzer generierte Ausnahme kann man erzeugen, wenn man das Schlüsselwort *throw* benutzt gefolgt von einem Ausdruck. Dadurch besteht die Möglichkeit, dass der Benutzer die jeweilige Ausnahme sehr spezifisch setzen kann, so dass bei der anschliessenden Ausnahmebehandlung ein Maximum an Informationen über die mögliche Ursache zur Verfügung steht. In except1.cpp wird einfach nur eine integer-Zahl generiert, die dann mit catch abgefangen wird.

```
//*********************************
//
// except1.cpp
//
// author: gerd doeben-henisch
//
// idea: simple exception-demo
//
// Compilation: g++ -o except1 except1.cpp
// Usage: except1
//
//*********************************************

#include <iostream>
#include <exception>

using namespace std;

int main(){

  try{
    throw 5;

  }

  catch(const int& n){
    cout << "Demo fuer try-catch" << std::endl;

  }
}


gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except1
Demo fuer try-catch
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12>
```

Im folgenden Beispiel von [JOSUTTIS 2001] wird die selbsterzeugte Ausnahme mit einer Klasse verknüpft, die der Anwender selbst definiert. Hier genügt die blosse 'Existenz' der Klasse und ihr Name *class NennerIstNull*; tritt ein Fehler in einem bestimmten Kontext auf, dann wird eine Ausnahme nur mit Hilfe des Klassennamens erzeugt *throw NennerIstNull();* (siehe bei Josuttis auf der Webseite http://www.josuttis.de/cppbuch/ die Dateien bruch.hpp, bruch8.cpp, btest8.cpp ; Kompilierung mit *g++ -o bruch8 btest8.cpp bruch8.cpp*):

```
  class Bruch {

  private:
    int zaehler;
    int nenner;

  public:
    /* neu: Fehlerklasse
```

```
     */
    class NennerIstNull {
    };
    ...
  };


 Bruch::Bruch (int z, int n)
{
    /* Zähler und Nenner wie übergeben initialisieren
     * - 0 als Nenner ist allerdings nicht erlaubt
     * - ein negatives Vorzeichen des Nenners kommt in den Zähler
     */
    if (n == 0) {
        // neu: Ausnahme: Fehlerobjekt für 0 als Nenner auslösen
        throw NennerIstNull();
    }
    if (n < 0) {
        zaehler = -z;
        nenner  = -n;
    }
    else {
        zaehler = z;
        nenner  = n;
    }
}
```

Im Programm wird dies so genutzt, dass bei Aufruf derjenigen Klasse, in der diese Ausnahme definiert wurde, dann automatisch eine Ausnahme von diesem speziellen Typ generiert wird. Mittels einer geeigneten catch-Anweisung kann dann dieser spezielle Fehler erfasst und behandelt werden:

```
  try {//...
       x = Bsp::Bruch(z,n);   //Hier kann Nenner=0 auftreten
      //...
    }

 catch (const Bsp::Bruch::NennerIstNull&) {
        /* Programm mit einer entsprechenden
         * Fehlermeldung beenden
         */
        std::cerr << "Eingabefehler: Nenner darf nicht Null sein"
                  << std::endl;
        return EXIT_FAILURE;
    }
```

```
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./bruch8
Zaehler: 3
Nenner: 0
Eingabefehler: Nenner darf nicht Null sein
```

Eine andere Variante zeigt das folgende Beispiel aus dem C++-Standard. Hier wird die spezielle Ausnahme-Klasse Overflow zusätzlich mit einem Funktions-Prototypen versehen.

```
class Overflow {
//...
      public:
      Overflow(char,double,double);
};
void f(double x) {
// ...
   throw Overflow( + ,x,3.45e107);
}
```

Eine so selbstdefinierte Ausnahmeklasse kann man auf folgende Weise zur Diagnose benutzen:

```
try {
     // ...
     f(1.2);
     // ...
}

catch(Overflow& oo) {
// Hier Anweisungen zur Behandlung von Ausnahmen des Typs Overflow
}
```

M.a.W. wenn der Funktionsaufruf f(1.2) einen Overflow-Fehler erzeugt, dann kann dieser entsprechend der eigenen Spezifikation abgefangen und speziell behandelt werden.

Man kann ferner try-Blöcke schachteln und mittels der Schlüsselworte *goto*, *break*, *return* oder *continue* aus diesen wieder 'herausspringen', nicht aber 'hinein'. Das folgende Beispiel entstammt dem C++Standard:

```
lab: try {
          T1 t1; try {
                       T2 t2;
                       if (condition) goto lab; }
                       catch(...) { /* handler 2 */ } }
          catch(...) { /* handler 1 */ }
```

Hier existieren zwei geschachtelte try-Blöcke, jeder mit einer catch-Anweisung ('handler'). Das ganze ist durch ein Label 'lab:' markiert. Trifft die eine if-Abfrage zu, dann springt der Programmablauf zum Label 'lab:' zurück und setzt die Abarbeitung hinter dem try-Block fort.

Eine throw-Anweisung ohne Argument wiederholt das Ausnahmeereignis indem es die Ausnahme neu erzeugt. Damit ist der Wert *uncaught_exception()* wieder wahr. Diese Eigenschaft kann man ausnutzen:

```
try {// ... }
catch (...) { // Erfasst alle Ausnahmen
              // Antwort auf eine spezielle Ausnahme
               throw;
               //Uebergabe der Ausnahme an eine andere catch-Anweisung
 }
```

Eine Ausnahme gilt als *'gefangen' ('caught')* wenn entweder die formalen Parameter einer catch-Anweisung erfüllt werden oder wenn *terminate()* oder *unexpected()* aufgrund einer throw-Anweisung verursacht werden. Eine Ausnahme gilt als vollständig behandelt, wenn die korrespondierende catch-Anweisung beendet wird oder *unexpected()* aufgerufen wird bedingt durch eine throw-Anweisung. Findet sich keine catch-Anweisung zu einer Ausnahme, dann wird *terminate()* ausgeführt.

Bevor auf die Definition eigener Fehlerklassen eingegangen wird zunächst eine Übersicht über die Standard-Fehlerklassen, die C++ zur Verfügung stellt.

3. Standardfehlerklassen

Die folgende Übersicht wurde entnommen aus [JOSUTTIS 2001:102] und aus den Headerdateien des GNU-C++-Compilers Version 2.95 unter Linux; weitere Informationen finden sich natürlich im Standard selbst. Benutzt wurde auch [Kalev 1999].

Wie man aus dem Schaubild entnehmen kann, sind die Ausnahmen in C++ als Ausnahme-Hierarchie organisiert: an oberster Stelle steht die Ausnahmeklasse *exception*, und von dieser abgeleitet sind 7 weitere Ausnahmeklassen (siehe Bild). Die Ausnahmeklassen *runtime_error* sowie *logic_error* haben zusätzliche Unterklassen.

l ogic_error nimmt Bzug auf die Logik des Programms. Logische Fehler können vom Programm aus vorausgesehen werden. *runtime_errors* dagegen nicht; diese liegen 'jenseits des Programmbereiches' und können nur von der Ablaufumgebung selbst behandelt werden.

```
class exception {
public:
  exception () { }
  virtual ~exception () { }
  virtual const char* what () const;
};

class bad_exception : public exception {
public:
  bad_exception () { }
  virtual ~bad_exception () { }
};


class logic_error : public exception {
  string _what;
public:
  logic_error(const string& what_arg): _what (what_arg) { }
  virtual const char* what () const { return _what.c_str (); }
};

class domain_error : public logic_error {
public:
  domain_error (const string& what_arg): logic_error (what_arg) { }
};

class invalid_argument : public logic_error {
public:
  invalid_argument (const string& what_arg): logic_error (what_arg) { }
};

class length_error : public logic_error {
public:
  length_error (const string& what_arg): logic_error (what_arg) { }
};

class out_of_range : public logic_error {
public:
  out_of_range (const string& what_arg): logic_error (what_arg) { }
};

class runtime_error : public exception {
  string _what;
public:
  runtime_error(const string& what_arg): _what (what_arg) { }
  virtual const char* what () const { return _what.c_str (); }
protected:
  runtime_error(): exception () { }
};

class range_error : public runtime_error {
public:
  range_error (const string& what_arg): runtime_error (what_arg) { }
};

class overflow_error : public runtime_error {
public:
  overflow_error (const string& what_arg): runtime_error (what_arg) { }
};

class underflow_error : public runtime_error {
public:
  underflow_error (const string& what_arg): runtime_error (what_arg) { }
```

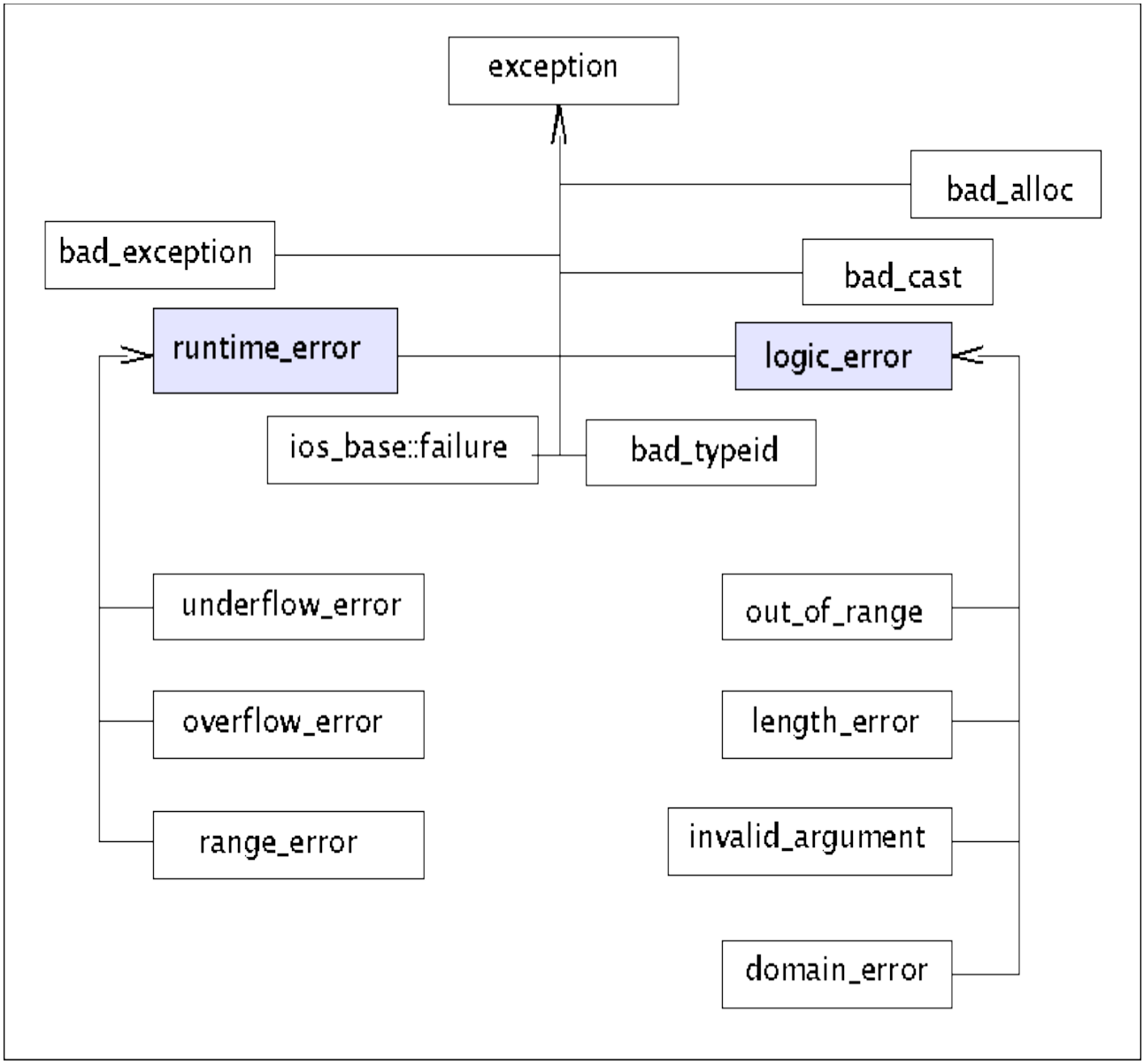
```
};
```

Ereignisse entsprechend diesen Ausnahmeklassen werden von der C++-Ablaufumgebung automatisch generiert; der Benutzer muss sich allerdings selbst darum kümmern, die automatisch erzeugten Ausnahmeereignisse mit catch() abzufangen und zu behandeln.

Bei der Bearbeitung durch catch-Anweisungen ist allerdings zu beachten, dass die interne Hierarchie der Ausnahmeklassen gewahrt wird. Dies hängt mit der *Stapel-Rückabwicklung ('stack unwinding')* zusammen (siehe unten).

Im nachfolgenden Beispiel wird von diesen Diagnose-Klassen Gebrauch gemacht. Zu diessem Zweck muss nur die Headerdatei <stdexcept> zusätzlich eingebunden werden.

Wie schon im vorausgehenden Beispiel wird eine Ausnahme dadurch erzeugt, dass bei einem String-Objekt der Bereich überschritten wird. In diesem Fall wird aber nicht allgemein nach einer 'exception' gefragt, sondern speziell nach einer 'out_of_range'-Ausnahme.



Ausnahmeklassen in der STL

```
//*********************************
//
// except0b.cpp
//
// author: gerd doeben-henisch
//
// idea: simple exception-demo
//
// Compilation: g++ -o except0b except0b.cpp
// Usage: except0b
//
//*********************************************

#include <iostream>    // Headerdatei für I/O
#include <string>      // Headerdatei für Strings
#include <cstdlib>     // Headerdatei fuer EXIT_FAILURE
#include <exception>   // Headerdatei für Ausnahmen
#include <stdexcept>   // Headerdatei für Ausnahme-Diagnostik

int main()
{
    try {
        std::string vorname("gerd");      // kann std::bad_alloc auslösen

        vorname.at(15) = 'B';    // löst std::out_of_range aus

    }
    catch (const std::out_of_range& e) {
        // Spezielle Ausnahme: Bereich überschritten
        std::cerr << "Bereich ueberschritten" << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }
    catch (const std::exception& e) {
        // sonstige Standard-Ausnahmen
        std::cerr << "Standard-Exception: " << e.what() << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }
    catch (...) {
        // alle (bisher nicht behandelten) Ausnahmen
        std::cerr << "unerwartete sonstige Exception" << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }

    std::cout << "OK, bisher ging alles gut" << std::endl;
}
```

```
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> g++ -o except0b
except0b.cpp
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except0b
Bereich ueberschritten
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12>
```

4. Stapel Rückabwicklung ('stack unwinding')

Tritt bei Programmablauf eine Ausnahme auf, dann sucht die Ablaufumgebung nach der nächsten catch-Anweisung. Wird sie nicht in dem aktuellen Block gefunden, dann wird der aktuelle Block vom Stapel gelöscht. Kommen dabei Objekte mit Destruktoren vor, werden diese Destruktoren ausgeführt. Dieser Vorgang ist iterativ; der Stapel wird solange 'abgebaut', bis entweder eine zutreffende catch-Anweisung gefunden wird oder aber das Programm beendet wird.

Das Beispiel except3.cpp zeigt den Fall, dass eine Funktion f1() im Falle einer Ausnahme dazu führt, dass das Programm hinter f1() nicht mehr zu Ausführung kommt. Eine Ausnahme wäre die Übergabe eines Strings mit Länge length, die kleiner ist als der Index bei Funktionsaufruf. Tritt in f1() keine Ausnahme auf, dann wird das Programm hinter f1() weitergeführt und führt dort zu einer Ausnahme wegen falschem Index.

```
/* Die folgenden Code-Beispiele stammen aus dem Buch:
 *  Objektorientiertes Programmieren in C++
 *   Ein Tutorial für Ein- und Umsteiger
 * von Nicolai Josuttis, Addison-Wesley München, 2001
 *
 * (C) Copyright Nicolai Josuttis 2001.
 * Diese Software darf kopiert, verwendet, modifiziert und verteilt
 * werden, sofern diese Copyright-Angabe in allen Kopien vorhanden ist.
 * Diese Software wird "so wie sie ist" zur Verfügung gestellt.
 * Es gibt keine explizite oder implizite Garantie über ihren Nutzen.
 */
//*******************************
//
// except3.cpp
//
// author: JOSUTTIS
// modified by: gerd doeben-henisch
//
// idea: Stapel-Rueckabwicklung
//
// Compilation: g++ -o except3 except3.cpp
// Usage: except3
//
//*******************************************

#include <iostream>    // Headerdatei für I/O
#include <string>      // Headerdatei für Strings
#include <cstdlib>     // Headerdatei fuer EXIT_FAILURE
#include <exception>   // Headerdatei für Ausnahmen

using namespace std;

char f1 (const std::string s, int idx)
{
    std::string tmp = s;  // lokales Objekt, das bei einer Ausnahme
                          // automatisch zerstört wird
    char c = s.at(idx);   // könnte Ausnahme auslösen

    return c;
}

int main()
{

    try {
```

```
      string s;

        std::string vorname("gerd");      // kann std::bad_alloc auslösen

        cin >> s;
        f1(s,11);    // löst eine Ausnahme aus
        cout << "Dies ist hinter der Funktion f1() " << std::endl;
        vorname.at(15) = 'B';    // löst std::out_of_range aus

    }
    catch (const std::exception& e) {
        // sonstige Standard-Ausnahmen
        std::cerr << "Standard-Exception: " << e.what() << std::endl;
        return EXIT_FAILURE;    // main() mit Fehlerstatus beenden
    }

    std::cout << "OK, bisher ging alles gut" << std::endl;
}
```

```
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> g++ -o except3
except3.cpp
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except3
abc
Standard-Exception: pos >= length ()
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except3
abcdefghijklm
Dies ist hinter der Funktion f1()
Standard-Exception: pos >= length ()
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12>
```

Das Beispiel except4.cpp zeigt eine Variante, bei der die Ausnahmebehandlung durch catch-All --also catch (...)--- nicht zum Abbruch führt; dann wird die Abarbeitung hinter der catch-Anweisung fortgesetzt.

```
//*********************************
//
// except4.cpp
//
// author: JOSUTTIS
// modified by: gerd doeben-henisch
//
// idea: Stapel-Rueckabwicklung
//
// Compilation: g++ -o except4 except4.cpp
// Usage: except4
//
//*********************************************

#include <iostream>    // Headerdatei für I/O
#include <string>      // Headerdatei für Strings
#include <cstdlib>     // Headerdatei fuer EXIT_FAILURE
```

```
#include <exception>   // Headerdatei für Ausnahmen


char f1 (const std::string s, int idx)
{
    std::string tmp = s;  // lokales Objekt, das bei einer Ausnahme
                          // automatisch zerstört wird
    char c = s.at(idx);   // könnte Ausnahme auslösen

    return c;
}

int main()
{

    try {

      string s;

        std::string vorname("gerd");      // kann std::bad_alloc auslösen

        cin >> s;
        f1(s,11);    // löst eine Ausnahme aus
        cout << "Dies ist hinter der Funktion f1() " << std::endl;
        vorname.at(15) = 'B';    // löst std::out_of_range aus

    }
    catch (...) {
        std::cerr << "Exception, wir machen aber weiter" << std::endl;
    }

    std::cout << "OK, bisher ging alles gut" << std::endl;
}
```

```
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> g++ -o except4
except4.cpp
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except4
abc
Exception, wir machen aber weiter
OK, bisher ging alles gut
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12> ./except4
abcdefghijklm
Dies ist hinter der Funktion f1()
Exception, wir machen aber weiter
OK, bisher ging alles gut
gerd@goedel:~/WEB-MATERIAL/fh/I-PROGR2/I-PROGR2-EX/EX12>
```

5. Eigene Fehlerklassen

Bei grösseren Programmen empfiehlt sich grundsätzlich, die Fehlerbehandlung in einer gemeinsamen Fehlerklasse zu bündeln. Man fängt mit catch-All alle Ausnahmen an einer stelle auf und behandelt dann dort alle Fälle. Ein dazu passendes Kodefragment könnte wie folgt aussehen:

```
//**********************************
//
// Gemeinsame Fehlerklasse
//
//**********************************


class Fehler {

public:

  std::string message;  //Default-Fehlermeldung

  //Konstruktor: Fehlermeldung initialisieren
  Fehler (const string& s) :message(s) {
  }

   ....

};


int main()
{

  try {
        ...
    }
  catch (const Fehler& fehler) {

        //Hier koennten komplexe Fallunterscheidungen stehen

      std::cerr << "FEHLER" << fehler.message << std::endl;
        return EXIT_FAILURE;
    }
}
```

6. Weitere Beispiele

Als zusätzliches Beispiel sei hier die spezielle Fehlerklasse vorgestellt, die [WEISS 2001] für die Behandlung von binären Suchbäumen benutzt:

```
#ifndef EXCEPT_H_
#define EXCEPT_H_

#ifndef NO_RTTI
    #include <typeinfo>
#endif

#ifdef SAFE_STL
    #include "mystring.h"
    #include "StartConv.h"
#else
    #include <string>
    using namespace std;
#endif
```

```
class DSException
{
  public:
    DSException( const string & msg = "" ) : message( msg )
      { }
    virtual ~DSException( )
      { }
    virtual string toString( ) const
#ifndef NO_RTTI
      { return "Exception " + string( typeid( *this ).name( ) ) + ": " + what
( ); }
#else
      { return "Exception " + string( ": " ) + what( ); }
#endif
    virtual string what( ) const
      { return message; }

  private:
    string message;
};

class GraphException : public DSException
{
  public:
    GraphException( const string & msg = "" ) : DSException( msg )
      { }
};

class NullPointerException : public DSException
{
  public:
    NullPointerException( const string & msg = "" ) : DSException( msg )
      { }
};



class IndexOutOfBoundsException : public DSException
{
  public:
    IndexOutOfBoundsException( const string & msg = "" ) : DSException( msg )
      { }
    IndexOutOfBoundsException( int idx, int sz, const string & msg = "" )
      : DSException( msg ), index( idx ), size( sz ) { }

    int getIndex( ) const
      { return index; }
    int getSize( ) const
      { return size; }

  private:
    int index;
    int size;
};


class ArrayIndexOutOfBoundsException : public IndexOutOfBoundsException
{
  public:
    ArrayIndexOutOfBoundsException( const string & msg = "" )
      : IndexOutOfBoundsException ( msg ) { }
    ArrayIndexOutOfBoundsException( int idx, int sz, const string & msg = "" )
      : IndexOutOfBoundsException( idx, sz, msg ) { }
```

```
};


class StringIndexOutOfBoundsException : public IndexOutOfBoundsException
{
  public:
    StringIndexOutOfBoundsException( const string & msg = "" )
      : IndexOutOfBoundsException ( msg ) { }
    StringIndexOutOfBoundsException( int idx, int sz, const string & msg = "" )
      : IndexOutOfBoundsException( idx, sz, msg ) { }
};


class UnderflowException : public DSException
{
  public:
    UnderflowException( const string & msg = "" ) : DSException( msg )
      { }
};


class OverflowException : public DSException
{
  public:
    OverflowException( const string & msg = "" ) : DSException( msg )
      { }
};


class ItemNotFoundException : public DSException
{
  public:
    ItemNotFoundException( const string & msg = "" ) : DSException( msg )
      { }
};


class DuplicateItemException : public DSException
{
  public:
    DuplicateItemException( const string & msg = "" ) : DSException( msg )
      { }
};


class IteratorException : public DSException
{
  public:
    IteratorException( const string & msg = "" ) : DSException( msg )
      { }
};


class IteratorOutOfBoundsException : public IteratorException
{
  public:
    IteratorOutOfBoundsException( const string & msg = "" ) : IteratorException
( msg )
      { }
};


class IteratorUninitializedException : public IteratorException
{
  public:
```

```
    IteratorUninitializedException( const string & msg = "" ) :
IteratorException( msg )
      { }
};


class IteratorMismatchException : public IteratorException
{
  public:
    IteratorMismatchException( const string & msg = "" ) : IteratorException
( msg )
      { }
};

class BadArgumentException : public DSException
{
  public:
    BadArgumentException( const string & msg = "" ) : DSException( msg )
      { }
};

#ifdef SAFE_STL
    #include "EndConv.h"
#endif

#endif
```

Die zugehörige catch-Anweisung lautet wie folgt:

```
#include "weiss-fehlerklasse.hpp"

int main( )
{
    try {
    ...

    }
    catch( const DSException & e )
    {
        cout << e.toString( ) << endl;
    }

    return 0;
}
```
