Dr. E.Huckert 12-2006

# Ausnahmebehandlung in C++ , Windows und Linux

## Inhaltsverzeichnis



## *1 Überblick*

Die Ausnahmebehandlung unter C++ ist weniger verbreitet als unter Java, weil Java zur Behandlung vieler Ausnahmen zwingt, C++ dagegen nicht. Da aber die Behandlung von Ausnahmen "state of the art" ist habe ich versucht, die Grundlagen für C++ unter besonderer Berücksichtigung der Betriebssystem Windows und Linux zu erforschen. Dabei standen die folgenden Fragen im Vordergrund:

- welche Standardmittel zur Ausnahmebehandlung bietet C++?
- wie verhalten sich Ausnahmen unter Windows?
- wie verhalten sich Ausnahmen unter Linux?

Nun gibt es einen wichtigen Unterschied in den Szenarien, unter denen Java im Gegensatz zu C++ eingesetzt wird: Java Programme sind aus naheliegenden Gründen (Kapselung der Java Virtual Machine) wenig ins Betriebssytem integriert. C++ Programme oder gar C Programme arbeiten dagegen sehr oft dicht am jeweiligen Betriebssystem. Zudem verwenden C++ Programme sehr oft Binärbibliotheken, Linkmodule oder Quellcodemodule, die keine Ausnahmebehandlung aufweisen. In solchem Altcode "knallt" es nicht selten: das sind meist "unhandled exceptions", also Ausnahmen, mit denen der Programmierer nicht gerechnet hat. Was kann man dagegen tun?

Eine Bemerkung: die nachstehenden Beispiel versuchen, eine Ausnahme durch Division durch 0 zu provozieren. Dabei ist allerdings zu beachten, dass eine solche Ausnahme in neueren Systemen nicht mehr in allen Fällen zu einer Ausnahme führt. Sehr oft kann die Behandlung von "Division durch 0" für Gleitkommazahlen durch Konfiguration oder API Aufrufe ausgeschaltet werden. Die Integer Division durch 0 (die ja im Integer Fall nicht in Bibliotheken auftritt, sondern direkt in den erzeugten Maschinenbefehlen) führt hingegen

immer zu einer Ausnahme.

## *2 Ausnahmebehandlung in Standard C++*

Standard C++ enthält als Sprachelemente die Schlüsselwörter *try, catch und throw* für die Ausnahmebehandlung. Ein *try* Block kapselt dabei den anwendungsorientierten Code, der eine Ausnahme auslösen kann. Ein oder mehrere *catch* Blöcke hinter dem try Block enthalten den Code, der auf die Ausnahme reagieren soll. Eine *throw* Anweisung erlaubt es, eigene Ausnahmen zu provozieren. Normalerweise werden dazu eigene Ausnahmenklassen konzipiert, die von der Basisklasse exception abgeleitet sind oder aus einer der weiteren Ausnahmeklassen, die in C++ (genauer gesagt: in der C++ Standard Library) definiert sind.

Wichtig: ohne weitere Maßnahmen werden mit Standard C++ Mitteln unerwartete Ausnahmen (wie z.B. nicht erkannte Division durch 0) nicht abgefangen. Die Programme stürzen weiterhin kläglich ab. Die Ausnahmebehandlung mit Standard C++ Mitteln ist aber natürlich weiterhin sinnvoll, wenn eigene Exceptions geworfen werden oder wenn neuere Standardbibliotheken wie die STL eingesetzt werden. Neuere Bibliotheken werfen im Allgemeinen eigene Ausnahmen oder die wenigen Standardausnahmen, die in C++ normiert sind (z.B. *bad_alloc*).

Das nachfolgende Beispiel zeigt eine Lösung, die nur C++ Standardmittel verwendet. Bitte beachten Sie die eigene Ausnahmenklasse *MyException,*. die von der Standardklasse *exception* abgeleitet ist. Diese Standardklasse *exception* erzwingt die Definition einer Methode *what()*. Ansonsten zeigt das Beispiel deutlich, dass wir nicht auf Ausnahmen warten, sondern versuchen, den Ausnahmefall zu erkennen und eigene Ausnahmen zu provozieren:

```
// Beispiel 1
// Demonstrates exceptions in Standard C++
// Copyright Dr.E.Huckert 12-2006
// Note: You cannot catch by simple methods any crash that is
//       caused by errors in libraries that do not throw explicit
//       exceptions! The "division by zero" error must be detected
//       by our own program!

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <exception>

using namespace std;

class MyException : public exception
{
private:
  char reason[256];
public:
  MyException(const char *msg)
  {
    if (strlen(msg) < sizeof(reason)) strcpy(this->reason, msg);
    else strcpy(reason, "reason code too long");
  };
  const char *what() const throw()
```

```
  {
     return reason;
  }
};

int main(int argc, char *argv[])
{
  double dv1 = 10.0;
  double dv2 = 0.0;
  //
  // wir provozieren Division durch 0
  try
  {
    // wirft erstaunlicherweise keine Exception und das
    // Programm stuerzt auch nicht ab...
    double dv3 = dv1 / dv2;
    cout << "Ergebnis Division durch 0(double):" << dv3 << endl;
    //
    int iv1 = 10;
    int iv2 = 0;
    //
    // wir provozieren Division durch 0
    if (iv2 == 0) throw(MyException("Division durch 0"));
     int iv3 = iv1 / iv2;
      printf("Ergebnis Division durch 0(int): %d\n", iv3);
  }
  catch (const exception& e)
  {
    cout << "Exception aufgetreten und abgefangen: " <<
     e.what() << endl;
  }
  return 0;
}   // end main()
```

## *3 Ausnahmebehandlung unter Windows*

Unter Windows gibt es Möglichkeiten zur Behandlung von Ausnahmen, die weit über die Mittel von Standard C++ hinausgehen. Insgesamt lassen sich drei Szenarien unterscheiden:

- Standardmittel von C++ auch unter Windows (s.oben)
- Verwendung von SEH ("structured exception handling")
- Verwendung von API Calls

Die beiden letzten, Windows32 spezifischen  Mittel sind auch mit Standard C einsetzbar (und wahrscheinlich auch mit anderen Programmiersprachen, die unter Windows einsetzbar sind).

## 3.1 Benutzung des "Structured Exception Handling"

Die üblichen Compiler definieren für Windows vier Makros: __*try*, __*except*, __*finally* und __*leave*. Dabei entspricht __*try* völlig dem *try* aus Standard C++ (d.h. es kapselt den anwendungsorientierten Code ein). __*except* entspricht dem *catch* aus Standard C++.

__*finally* entspricht dem *finally* unter Java: der danach folgende Block wird immer ausgeführt – gleichgültig ob eine Ausnahme aufgetreten ist oder nicht („termination handler“). __*finally* wird (trotz anderer Aussagen in der Dokumentation) vom Digital Mars Compiler nicht erkannt.

Wichtig: die Mittel von Windows erlauben es auch, nicht behandelte Ausnahmen (d.h. katastrophale Fehler die für den Programmierer unerwartet und unbehandelt auftreten) abzufangen. Schon die einfachste Variante (s. Beispiel 2) reagiert auf Ausnahmen wie z.B. Division durch int - 0, wenn der Code durch __*try* gekapselt wird und wenn im __*except* Block auf alle Ausnahmen reagiert wird z.B. durch :

```
__except(true)  // wir reagieren durch "true" auf alle Exceptions
{
   int reason = GetException();
   ….
}
```

Wie man im nachstehende Beispiel sieht, werden nicht nur Spracherweiterungen wie _*try* verwendet, sondern auch einige wenige API Aufrufe (*RaiseException()* und *GetExceptionCode()*). Zum Testen sollten Sie den Aufruf von *stuerzeAb()* auch mal in Kommentare setzen:

```
// Beispiel 2
// Demonstrates execptions in Windows
// Copyright Dr.E.Huckert 12-2006
// This is a console program deigned to run under Windows 32 bit

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#define MEIN_FEHLER_DIV0I   10001
#define MEIN_FEHLER_DIV0D   10002

void stuerzAb()
{
  strcpy(letzteProzedur, "stuerzAb()");
 int *pNull = NULL;
  *pNull  = 0;
}

int division(int p1, int p2)
{
  strcpy(letzteProzedur, "division()");
  return p1 /p2;
}

int main(int argc, char *argv[])
{
  double dv1 = 10.0;
  double dv2 = 0.0;
  DWORD dword = 17L;  // nur so...
 //
  stuerzAb(); // ggf. zum Testen in Kommentare setzen
  //
  // der folgende Code sollte nicht mehr erreicht werden.
  __try
  {
```

```
    // wirft erstaunlicherweise keine Exception und das
    // Programm stuerzt auch nicht ab...
  if (dv2 == (double)0.0) RaiseException(MEIN_FEHLER_DIV0D,
                                          0, 1, &dword);
    double dv3 = dv1 / dv2;
  cout << "Ergebnis Division durch 0(double):" << dv3 << endl;
    //
    int iv1 = 10;
    int iv2 = 0;
    //
    // wir provozieren Division durch int 0
  if (iv2 == 0) RaiseException(MEIN_FEHLER_DIV0I, 0, 1, &dword);
    int iv3 = division(iv1, iv2);
  printf("Ergebnis Division durch 0(int): %d\n", iv3);
  }
  __except(true)
  {
    // wird erstaunlicherweise auch erreicht fuer Exceptions, die nicht
    // selbst ausgeloest werden (d.h. RaiseException() ist nicht noetig)
    cout << "Exception aufgetreten und abgefangen: " <<
             GetExceptionCode() << endl;
  }
  /*
  // _finally funktioniert syntaktisch mit Digital Mars nicht...?
 __finally
  {
    cout << "im __finally Block" << endl;
 }
  */
  return 0;
}   // end main()
```

Der wichtige Unterschied zur reinen C++ Variante liegt darin, dass der Abfangblock hinter *__except* auch erreicht wird, wenn die Exceptions nicht selbst ausgelöst werden über *RaiseException()*, d.h. wenn sie erst vom Betriebssystem erkannt werden.

## 3.2 Ausschließliche Verwendung von API Funktionen

Bisher wurde davon ausgegangen, dass der absturzträchtige Code speziell markiert wird durch Syntaxelemente wie *__try* („guards“). Die in diesen Codestrecken auftretenden Ausnahmen sind meist "handled exceptions", d.h. der Code ist darauf vorbereitet.

Ausnahmen, die in speziell angereichertem Code – also Code mit *try* und *catch* Blöcken – an den erwarteten Stellen auftreten, heißen im Programmier-Englisch „handled exceptions“. Es gibt unter Windows auch die Möglichkeit, den Code unbehandelt zu lassen und trotzdem auf Ausnahmen zu reagieren - ob das programmiertheoretisch sinnvoll ist, soll hier nicht weiter diskutiert werden. Dazu muss man eine Routine definieren und mit *SetUnhandledExceptionHandler()* bekanntmachen, die unerwartete Ausnahmen auffängt und in die Kontrolle des Programmierers gibt. In alten Assemblertagen hätte man gesagt: man biegt einen Interruptvektor um.

Hier konzentrieren wir uns auf "unhandled exceptions", d.h. auf Ausnahmen, mit denen wir nicht gerechnet haben und für die wir demzufolge keine besonders feingliedrigen Maßnahmen (*try* .. *catch* etc.) unternommen haben. Die hier beschriebene Strategie ist übrigens nicht an C++ gebunden: da sie keine syntaktischen Elemente von C++ verwendet, funktio-

niert sie auch mit Standard C und sollte auch (ich habe es nicht ausprobiert) mit C# und anderen Programmiersprachen unter Windows funktionieren. Statt besondere syntaktischer Elemente werden die von Windows32 angebotenen API Funktionen zur Ausnahmebehandlung benutzt.

Die wichtigste Maßnahme besteht darin, am Anfang des Programms oder an sonstiger geeigneter Stelle über die Funktion *SetUnhandledExceptionFilter()* eine eigene Prozedur (hier *catchUnhandledException()*) zu definieren, die im Falls eines Absturzes angesteuert werden soll. Normalerweise erscheint unter Windows eine MessageBox, die einige obskure Adressen ausgibt um anzuzeigen, wo der Absturz ausgelöst wurde. Wenn man dann eine Symboltabelle (die man hoffentlich zuvor erzeugt hatte) lesen kann, kommt man dem Fehler vielleicht auf die Spur. Die hier statt dessen verwendete eigene Prozedur zeigt auch eine MessageBox - allerdings mit einem eigenen Text. Um dem Programmierer wenigstens eine Idee zu geben, in welcher Routine denn der Absturz ausgelöst wurde, habe ich beim Einstieg in jede anwendungsorientierte Prozedur mir den Prozedurnamen gemerkt. Dieser Prozedurname wird im Falle des Absturzes in der MessageBox angezeigt. Diese elementare Maßnahme kann natürlich beträchtlich ausgebaut werden.

Man kann Windows auch zwingen, die Verarbeitung fortzusetzen, wenn man den entsprechenden Return - Code setzt (s. die Kommentare im Beispielsprogramm) - allerdings landet man dann u.U. schnell in einer Endlosschleife. Vorausetzung für eine Fortsetzung des Programms ist eine Korrektur des Stacks - das allerdings führt hier zu weit und ist auch etwas kompliziert.

Auch bei Verwendung einer eigenen Prozedur zum Aufffangen des Absturzes kann man weiterhin einen Debugger starten. Vorausetzung ist auch hier der richtige Returncode (s. Kommentare im Beispielsprogramm) und natürlich die Registrierung eines Debuggers in der Registry unter dem Schlüssel:

```
HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows NT/CurrentVersion/AeDebug
```

Eine Kombination beider Methoden - Einsatz von *try .. catch* bzw. *__try .. __except* und Definition einer eigenen Fangprozedur - kann sinnvoll sein und funktioniert unter Windows. Um einem falschen Eindruck vorzubeugen: man kann natürlich auch unterschiedliche Fangroutinen definieren - für jede schützenswerte Codestrecke eine eigene.

```
// Beispiel 3// Demonstrates exceptions under Windows
// Copyright Dr.E.Huckert 12-2006
// This is a console program designed to run under Windows 32 bit
// This version demonstrates the use of standard Windows API calls
// It concentrates on "unhandled exceptions"

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

static char letzteProzedur[256]  = "";  // last procedure name

// will be invoked when an unhandled execption happens
LONG WINAPI catchUnhandledException(EXCEPTION_POINTERS* ep)
{
  static char msg[1024] = "Unhandled Exception catched! last procedure=";
  strcat(msg, letzteProzedur);
  MessageBox(NULL, msg, NULL, MB_OK);
  //return (EXCEPTION_CONTINUE_SEARCH);     // calls registered debugger
  //return (EXCEPTION_CONTINUE_EXECUTION);  // endless loop!!!
```

```
  return (EXCEPTION_EXECUTE_HANDLER);       // terminates;
}  //  end catchUnhandledException()

// leads to an access violation
void stuerzAb()
{
  strcpy(letzteProzedur, "stuerzAb()");
  int *pNull = NULL;
  *pNull  = 0;
}

// leads to an acccess violation (divide by zero) if
// p2 is 0
int division(int p1, int p2)
{
  strcpy(letzteProzedur, "division()");
  return p1 /p2;
}

int main(int argc, char *argv[])
{
 cout << "Demonstrates traps for unhandled exceptions.  By Dr.E.Huckert"
       << endl;
 //
  // install a filter for unhandled exceptions
 SetUnhandledExceptionFilter(catchUnhandledException);
  //
  // Floating Point: wirft erstaunlicherweise keine Exception und das
  // Programm stuerzt auch nicht ab...
 double dv1 = 10.0;
  double dv2 = 0.0;
  double dv3 = dv1 / dv2;
  cout << "Floating point result division by 0(double):" << dv3 << endl;
  //
  stuerzAb();
  //
  // this code will (probably) never be reached...
  int iv1 = 10;
  int iv2 = 0;
  int iv3 = division(iv1, iv2);
  printf("integer result division by 0(int): %d\n", iv3);
 //
  return 0;
}   // end main()
```

In der C++ Standard Bibliothek werden zwei Routinen *std::set_unexpected()* und *std::set_terminate()* - definiert. Die erste Routine erlaubt die Festlegung einer Fangroutine für unerwartete Ausnahmen - allerdings nur für solche Ausnahmen, die nicht in der *throw* Liste aufgeführt sind. Diese Bibliotheksroutine löst m.E. keine wirklich wichtigen Probleme. Die zweite Routine wird nur durch den Aufruf von *terminate()* aktiviert und ist deshalb auch nicht wirklich interessant.

## *4 Ausnahmebehandlung unter Unix/Linux*

Unter Unix/Linux sind natürlich die Standardmittel von C++ einsetzbar. Beispiel 1 läuft also auch problemlos unter Unix/Linux. Allerdings können auch hier Ausnahmen auftreten, die erst vom Betriebssystem erkannt werden – wenn auch in den von mir getesteten Fällen die

erzeugten Meldungen aussagekräftiger als unter Windows waren. Es wird also auch hier notwendig, unser Rumpfprogramm anzureichern um API Funktionen von Unix/Linux.

In Unix/Linux lösen Ausnahmen das Senden von "Signalen" aus. Signale sind eine besonders einfache (und frühe) Variante von Meldungen zwischen Prozessen, also ein Mittel der Prozesskommunikation. Unter Signale, die durch Codeereignisse oder Hardware-ereignisse ausgelöst werden, fallen auch einfache benutzergenerierte Signale (z.B. *SIGUSR1*) oder betriebssystemgenerierte Meldungen (wie CTL-S und CTL-Q). Die Signale sind in Linux definiert in *../include/bits/bits.h.*

Signale werden traditionell in Signalhandlern („Fangroutinen“) erkannt und behandelt. Diese Methode ist natürlich nicht an C++ gebunden. Sie stammt im Gegenteil aus ganz frühen Unix und C-Tagen. Die in Linux verbreiteten Varianten der Signalverarbeitung setzen durchgängig auf den POSIX Normen auf. Generell sind im Umfeld der Signalverarbeitung drei Typen von API Calls zu unterscheiden :

- Definition von Signalmasken: *sigfillset(), sigemptyset, sigaddset(), sigdelset(), sigprocmask()*
- Zuordnung von Fangroutinen: *sigaction()*
- Auslösen von Signalen*: kill(), raise()*

Über Signalmasken wird festgelegt, an welchen Signalen die Anwendung interessiert ist. Dabei ist zu beachten, dass bestimmte prozess- und systemkritische Signale wie *SIGKILL*, *SIGSTOP* oder auch *SIGSEGV* faktisch nicht ausgeblendet werden können. „Harmlose“ Signale wie *SIGUSR1* (ein „user signal“) können z.B. so ignoriert werden:

```
sigset_t set1;
sigemptyset(&set1);
sigaddset(&set1, SIGUSR1)
sigprocmask(SIG_SETMASK, &set1, NULL);
```

Diese Anweisungsfolge sorgt dafür, dass alle Signale außer SIGUSR1 durchkommen. Die Signalmaske wird durch *sigemptyset()* leer gesetzt: damit sind alle Signale zugelassen. Mit *sigaddset()* wird dann das einzelne Signal SIGUSR1 fürs Blockieren vorgemerkt. Schickt man jetzt der Anwendung ein Signal *SIGUSR1* (z.B. über `kill(::getpid(), SIGUSR1`)) so wird der hoffentlich definierte Signalhandler für *SIGUSR1* nicht betreten.

Es geht hier nicht darum, die aufgetretenen Fehler zu korrigieren oder gar im Programm weiterzufahren, als ob nichts geschehen wäre. Posix bietet dafür die Funktionen *sigsetjmp()* und *siglongjmp()* an. Der Einsatz solcher Korrekturroutinen muss jedoch im Einzelfall sorgfältig durchdacht werden (kein Aufruf von Destruktoren!). Vielmehr geht es hier darum, den aufgetretenen Fehler etwas freundlicher zu protokollieren und vor allem dem Programmierer einen Hinweis zu geben, wo und warum der Fehler aufgetreten ist. Eine Protokollierung des Namens der auslösenden Codestrecke wie im obigen Beispiel für Windows ist deshalb natürlich sinnvoll:

Das Aufsetzen eines Signalhandlers erfordert zwei Schritte:

- Definition der Fangroutine (s. Beispiel)
- Zuweisung der Fangroutine zu einem Signal durch *setaction()*

Das nachstehende Beispiel setzt Signalhandler auf für drei Fälle: Fließkommaausnahmen (auch Division durch Integer 0, Signal *SIGFPE*), Zugriffsschutzverletzungen (Signal *SIGSEGV*) und benutzerdefiniertes Signal *SIGUSR1* – das allerdings zu Demozwecken blockiert werden soll:

```
// Beispiel 4
// Demonstrates signal handling and execptions in C++ under Linux
// Copyright Dr.E.Huckert 12-2006

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <iostream.h>
#include <exception>
#include <signal.h>

using namespace std;

class MyException : public exception
{
private:
  char reason[256];
public:
  MyException(const char *msg)
  {
    if (strlen(msg) < sizeof(reason)) strcpy(this->reason, msg);
    else strcpy(reason, "reason code too long");
  };
  const char *what() const throw()
  {
     return reason;
  }
};

// Definition von Fangroutinen
// Die Signalnamen SIGSEV etc. sind in include/bits/signum.h deklariert (Linux)
void sigfpe_handler(int sig)
{
  cout << "signal SIGFPE=" << sig << " received" << endl;
  exit(SIGFPE);  // floating point exception
}   // end sigfpe_handler()

void sigsegv_handler(int sig)
{
  cout << "signal SIGSEGV=" << sig << " received" << endl;
  exit(SIGSEGV); // segmentation violation
}   // end sigsegv_handler()

void sigusr1_handler(int sig)
{
  cout << "signal SIGUSR1=" << sig << " received" << endl;
}   // end sigusr1_handler()

int main(int argc, char *argv[])
{
  struct sigaction act;
  double dv1 = 10.0;
  double dv2 = 0.0;
  //
  // set up signal handlers
  sigemptyset(&(act.sa_mask));
  sigaddset(&(act.sa_mask), SIGUSR1);   // SIGUSR1  blockieren
```

```
  sigprocmask(SIG_SETMASK, &(act.sa_mask), NULL);
  //
  act.sa_handler = sigfpe_handler;
  sigaction(SIGFPE, &act, NULL);
  //
  act.sa_handler = sigsegv_handler;
  sigaction(SIGSEGV, &act, NULL);
  //
  act.sa_handler = sigusr1_handler;
  sigaction(SIGUSR1, &act, NULL);
  //
  // wir schicken uns selbst ein Signal SIGUSR1:  sollte nicht durchkommen
  kill(::getpid(), SIGUSR1);   // auch: raise(SIGUSR1)
  //
  // wir provozieren Division durch 0
  try
  {
    // wirft erstaunlicherweise keine Exception und das
    // Programm stuerzt auch nicht ab...
    double dv3 = dv1 / dv2;
    cout << "Ergebnis Division durch 0(double):" << dv3 << endl;
    //
    // wir provozieren eine "access violation": Signal SIGSEGV
    char *p =  NULL;
    *p = 'X';
    //
    // wir provozieren Division durch 0: Signal SIGFPE
    int iv1 = 10;
    int iv2 = 0;
    // Unter Linux: Absturz mit Meldung "Gleitkomma-Ausnahme"
    // Unter Windows:  Absturz mit entsprechender MessageBox
    //if (iv2 == 0) throw(MyException("Division durch 0"));  // ggf. in Kommentar
    int iv3 = iv1 / iv2;
    printf("Ergebnis Division durch 0(int): %d\n", iv3);
  }
  catch (const exception& e)
  {
    cout << "Exception aufgetreten und abgefangen: " <<
         e.what() << endl;
  }
  cout << "Programmende ohne Ausnahme und ohne Absturz" << endl;
  return 0;
}   // end main()
```

## *5 Zusammenfassung*

Die in C++ definierten Syntaxelemente zur Ausnahmenbehandlung sind selten ausreichend, wenn auch die Grundsyntax über die Betrissbsystemgrenzen portabel ist. Um Ausnahmen in den Griff zu kriegen, mit denen der Programmierer nicht gerechnet hat oder um Altcode zu kapseln muss man  zusätzlich API Funktionen des jeweiligen Betriebssystems verwenden.  Generell sollte die Strategie verfolgt werden, Ausnahmen soweit möglich immer zu erkennen und selbst zu behandeln. Dazu können die C++ Sprachelemente eingesetzt werden.  Für alles weitere – insbesondere für nicht erwartete Ausnahmen – sind dann API Calls zuständig. Die betriebssystemabhängigen Komponenten können auch verallgemeinert und  in Bibliotheken gekapselt werden, sodass im Quellcode des Anwendungsprogramms nur ein neutraler Aufruf wie  *initalisiereAusnahmen()* sichtbar ist.

## *6 Literatur*

Lauer, Thomas: Professionelle Win32-Programmierung, Bonn: Thomson Publishing Company, 1996, ISBN 3-8266-2664-8

Richter, Jeffrey: Advanced Windows 3rd edition, ohne Ort: Microsoft Press, ohne Jahr, ISBN 1-57231-548-2

Havilland, Keith et al.: Unix System Programming, Harlow: Addison Wesley Longman, 1999, ISBN 0-201-87758-9

Stroustrup, Bjarne: The C++ programming language, 3rd edition, Reading: Addison Wesley, 1997 (Kap. 14)