8

Fehlerbehandlung

Dieses Kapitel behandelt die folgenden Themen:

- Strategien zur Fehlerbehandlung
- Fehler abfangen
- Was tun, wenn Abfangen nicht möglich ist?

Oft tritt ein Fehler in einer Funktion auf, der innerhalb der Funktion selbst nicht behoben werden kann. Der Aufrufer der Funktion muss Kenntnis von dem aufgetretenen Fehler bekommen, damit er den Fehler abfangen oder noch weiter »nach oben« melden kann. Ein Programmabbruch in *jedem* Fehlerfall ist benutzungsunfreundlich und nicht immer erforderlich. Eine generelle Fehlerbehebung ist leider nicht möglich; jeder Einzelfall ist gesondert zu überlegen. Mit den bisher behandelten Mitteln lassen sich verschiedene Strategien zur Fehlerbehandlung realisieren. In der Aufzählung wird angenommen, dass der Fehler in einer aufgerufenen Funktion stattfindet.

1. Das programmiertechnisch Einfachste ist der sofortige Programmabbruch innerhalb der Funktion, die einen Fehler feststellt. Wenn keine erläuternden Meldungen über den Abbruch ausgegeben werden, ist die Fehlerdiagnose erschwert.
2. Ein üblicher Mechanismus zur Fehlerbehandlung ist die Übergabe eines Parameters an den Aufrufer der Funktion, der Auskunft über Erfolg oder Misserfolg der Funktion

gibt. Der Aufrufer hat den Parameter auszuwerten und entsprechend zu reagieren. Der Parameter ist nach jedem Funktionsaufruf abzufragen.

```
int ergebnis1 = f(par1, par2, fehler); // fehler wird per Referenz übergeben.
if(fehler) {
   switch (fehler) {
      case 1: ergebnis1 = -1; break;
      case 2: ergebnis1 = 0; break;
      default: cout << "nicht behebbarer Fehler in f()!" << endl;
               exit(-2); // Programmabbruch
   }
}

// ...
int ergebnis2 = g(par3, fehler);
if(fehler)
  exit(-3);              // Programmabbruch
// ... usw.
```

Der Vorteil liegt in der selektiven Art und Weise, wie der Aufrufer einer Funktion Fehler an der Stelle des Auftretens behandeln kann, sodass vielleicht sogar ein Abbruch nicht nötig ist. Der Nachteil dieser Verfahrensweise besteht darin, dass der Programmcode durch viele eingestreute Prüfungen schwerfällig wirkt und dass die Lesbarkeit leidet. Ein zweiter Nachteil ist, dass sich der Programmierer die Abfragen aus Schreibfaulheit spart in der Hoffnung, dass alles gut gehen wird.

3. Eine Funktion kann im Fehlerfall die in der C-Welt übliche globale Variable `errno` setzen, die dann wie im vorhergehenden Fall abgefragt wird – oder auch nicht! Globale Variablen sind jedoch grundsätzlich nicht gut geeignet, weil sie die Portabilität von Funktionen beeinträchtigen und weil beim Zusammenwirken mehrerer Programmteile Werte der Variablen möglicherweise nicht mehr eindeutig sind.
4. Eine Funktion, die einen Fehler feststellt, kann eine andere Funktion zur Fehlermeldung und Bearbeitung aufrufen, die gegebenenfalls auch den Programmabbruch herbeiführt. Diese Methode erspart dem Aufrufer die Fehlerabfrage nach Rückkehr aus der Funktion. Es ist in Abhängigkeit von der Art des Fehlers zu überlegen, ob dem Aufrufer die Information des Fehlschlags mitgeteilt werden muss. Dies gilt sicher dann, wenn die Funktion die ihr zugedachte Aufgabe nicht erledigen konnte.
5. Die leider noch anzutreffende »Kopf-in-den-Sand-Methode« ist die, einem Aufrufer trotz eines Fehlers einen gültigen Wert zurückzuliefern und weiter nichts zu tun. Beispiel: Den Index-Operator `[ ]` (siehe Kapitel 9) für eine String-Klasse so zu programmieren, dass bei einer Indexüberschreitung ohne Meldung immer das Zeichen an der Stelle 0 zurückgegeben wird. Diese Methode ist besonders tückisch, weil der Fehler sich möglicherweise durch die falschen Daten an einer ganz anderen Stelle und sehr viel später bemerkbar macht und daher schwer zu finden ist. Oder: Einzutragende Daten ohne Meldung verwerfen, wenn die Datei voll ist. Oder ähnlich Schlimmes mehr.

Die Art der Fehlerbehandlung hängt auch davon ab, wie sicherheitskritisch der Einsatz der Software ist. Eine Fehlermeldung *Index-Fehler im Array CONTROLPARAMETER, Index = 2198, max= 82* ist in einem Textverarbeitungsprogramm noch vertretbar und führt vielleicht zu einem Fehlerbericht an den Hersteller des Programms. Ganz anders kann es

sein, wenn diese Fehlermeldung in der Software eines in der Luft befindlichen Flugzeugs auftritt. Es reicht wahrscheinlich nicht aus, die Meldung mit Uhrzeit in einer Log-Datei zu speichern und dann im Programm fortzufahren. Es würde auch nichts helfen, dem Piloten *diese* Fehlermeldung anzuzeigen, weil er damit nichts anfangen kann.

Unter den oben genannten Möglichkeiten sind die zweite und noch mehr die vierte akzeptabel. C++ stellt zusätzlich die Ausnahmebehandlung bereit, mit der Fehler an spezielle Fehlerbehandlungsroutinen des aufrufenden Kontexts übergeben werden können.

8.1 Ausnahmebehandlung

Die vierte Fehlerbehandlungsstrategie von Seite 302 hat den Vorteil, dass der Programmcode, der die eigentliche Aufgabe erledigen soll, von vielen eingestreuten Fehlerprüfungen entlastet wird. C++ bietet die *Ausnahmebehandlung* (englisch *exception handling*) an, die ebenfalls den Fehlerbehandlungscode vom »normalen« Programm sauber trennt und die eine spezielle Abfrage von Fehlerparametern an vielen Stellen im Programm überflüssig macht. Zu unterscheiden ist zwischen der *Erkennung* von Fehlern wie zum Beispiel

- Division durch Null,
- Bereichsüberschreitung eines Arrays,
- Syntaxfehler bei Eingaben,
- Zugriff auf eine nichtgeöffnete Datei,
- Fehlschlag der Speicherbeschaffung oder
- Nichteinhaltung der Vorbedingung einer Methode

und der *Behandlung* von Fehlern. Die Erkennung ist in der Regel einfach, die Behandlung schwierig und häufig genug unmöglich, sodass ein Programm gegebenenfalls abgebrochen werden muss. Außer den *vorhersehbaren* Fehlern gibt es natürlich noch andere! Die Fehlerbehandlung sollte einen von zwei Wegen einschlagen:

- Den Fehler beheben und den Programmlauf fortsetzen oder, falls das nicht gelingt, das Programm mit einer aussagefähigen Meldung abbrechen *oder*
- den Fehler an das aufrufende Programm melden, das ihn dann ebenfalls auf eine dieser beiden Arten bearbeiten muss.

Für den Fall, dass Fehler in einer Funktion vom Aufrufer behandelt werden können, stellt C++ alternativ zur vierten Strategie auf Seite 302 die Ausnahmebehandlung zur Verfügung. Der Ablauf lässt sich wie folgt skizzieren:

1. Eine Funktion versucht (englisch *try*) die Erledigung einer Aufgabe.
2. Wenn sie einen Fehler feststellt, den sie nicht beheben kann, wirft (englisch *throw*) sie eine Ausnahme (englisch *exception*) aus.
3. Die Ausnahme wird von einer Fehlerbehandlungsroutine aufgefangen (englisch *catch*), die den Fehler bearbeitet.

try und catch

Die Funktion kann der Fehlerbehandlungsroutine ein Objekt eines beliebigen Datentyps »zuwerfen«, um Informationen zu übergeben. Im Unterschied zu der oben erwähnten vierten Strategie wird nach der Fehlerbehandlung *nicht* in die Funktion zurückgesprungen. Das Programm wird vielmehr mit dem der Fehlerbehandlung folgenden Code fortgesetzt.

Hinweis

Wenn aus einem Block herausgesprungen wird, werden die Destruktoren aller in diesem Block definierten automatischen Objekte aufgerufen! Diese Objekte werden dabei vom Laufzeit-Stack entfernt (englisch *stack unwinding*).

Dies kann ausgenutzt werden, um durch Exceptions verursachte Speicherlecks zu vermeiden (mit `shared_ptr`, siehe Seite 567). Aus dem vorhergehenden Absatz folgt, dass Destruktoren selbst *keine* Exceptions werfen dürfen, um das Aufräumen des Stacks nicht durch einen Sprung nach außen abrupt zu beenden. Das folgende Schema zeigt die syntaktische Struktur zum Werfen und Auffangen von Exceptions.

```
try {
     func();
     // Falls die Funktion func() einen Fehler entdeckt, wirft sie eine Ausnahme
     // aus (throw), wobei ein Objekt übergeben werden kann, um die geeignete
     // Fehlerbehandlung anzustoßen. Oder es wird in den Anweisungen ein Fehler
     // festgestellt, der zum Auswerfen einer Exception führt, etwa so:
     // weitere Anweisungen ...
     if(EsIstEinFehlerPassiert)
        throw Exception();
}

catch(Datentyp1 e) { // Syntax für Grunddatentypen, z.B. const char*
     // durch ausgeworfenes Objekt e ausgewählte Fehlerbehandlung
     // ...
}

catch(const Datentyp2& e) { // Klassenobjekte per Referenz übergeben
     // durch ausgeworfenes Objekt e ausgewählte Fehlerbehandlung
     // ...
}

// gegebenenfalls weitere catch-Blöcke
// Fortsetzung des Programms nach Fehlerbearbeitung an dieser Stelle!
// ...
```

Die Ausnahmebehandlung wird hier ausschließlich als Fehlerbehandlung verstanden. Um die Ausnahmebehandlung am Beispiel zu zeigen, wird davon leicht abgewichen, weil zum Beispiel das Erreichen des Dateiendes kein Fehler, sondern etwas Normales ist. Das Einlesen einer Zahl mit `cin >>` wird mit einer Fehlererkennung versehen, sodass fehlerhafte Eingaben ignoriert werden. Im folgenden Programm soll bei Fehleingaben die Meldung »Syntaxfehler« ausgegeben werden, ehe der Eingabestrom weiter verarbeitet wird. Das Dateiende wird bei Umleiten der Standardeingabe auf Betriebssystemebene erkannt. Die Kennung für ein Dateiende kann über die Tastatur eingegeben werden, falls die Stan-

dardeingabe nicht umgeleitet wird, in der Regel mit der Tastenkombination Strg+Z oder Strg+D.

Listing 8.1: Beispielprogramm zur Fehlerbehandlung

```
// cppbuch/k8/stream/exstream.cpp
#include<iostream>
using namespace std;

class DateiEnde : public exception {}; // Hilfsklasse (siehe unten),
                                       // erbt von exception, siehe Seite 307
int liesZahl(std::istream& ein) {
    int i;
    ein >> i;
    // Das eof-Bit bewirkt den Auswurf eines Objekts vom Typ
    // DateiEnde, das hier durch den Aufruf des systemgenerierten
    // Konstruktors erzeugt wird, an den umgebenden Kontext. Die Eingabe von
    // falschen Zeichen setzt das fail-Bit, das durch die Funktion fail()
    // abgefragt wird und den Auswurf eines const char*-Objekts bewirkt.
    // Jedes throw führt zum Verlassen der Funktion. Details zu eof(), fail()
    // und bad() siehe Abschnitt 10.4.
    if(ein.eof()) throw DateiEnde();
    if(ein.fail())throw "Syntaxfehler";
    if(ein.bad()) throw;          // nicht behebbarer Fehler
    return i;
} // liesZahl()

void zahlen_lesen_und_ausgeben() {
    int zahl;
    while(true) {                 // Endlosschleife
        cout << "Zahl eingeben:";
        bool erfolgreich = true;
        try{ // Versuchsblock
           // Zahl von der Standardeingabe lesen:
           zahl = liesZahl(cin);
        }
        // Fehlerbehandlung: Der folgende Ausnahme-Handler wird
        // angesprungen, wenn ein Objekt des Typs DateiEnde im try-Block
        // ausgeworfen wurde.
        catch(const DateiEnde& e) {
             cout << "Ende der Datei erreicht! e.what() liefert : "
                  << e.what() // von exception geerbte Methode
                  << endl;
             cin.clear();   // Fehlerbits rücksetzen, siehe Abschnitt 10.4
             break;         // Schleife verlassen
        }
        // Der folgende Ausnahme-Handler wird angesprungen, wenn ein Objekt des Typs
        // const char* ausgeworfen wurde. Die Funktion liesZahl() wirft einen
        // C-String aus, wenn in der Eingabe die Syntax von int-Zahlen verletzt
        // wird, zum Beispiel Buchstaben statt Ziffern.
        catch(const char* z) {
             cerr << z << endl;
             erfolgreich = false;
```

```
            cin.clear(); // Fehlerbits rücksetzen
            cin.get();   // fehlerhaftes Zeichen entfernen, s. Abschnitt 10.2
        }
        // bad() wird nicht abgefangen, ggf. Programmabbruch.
        // Fortsetzung des Programms nach der Fehlerbehandlung
        if(erfolgreich)
            cout << "Zahl = " << zahl << endl;
    }
}

int main() {
    zahlen_lesen_und_ausgeben();
}
```

Die Klasse `DateiEnde` kann von der Standardklasse `exception` (Beschreibung folgt) erben und deren Methoden nutzen, wie mit dem Aufruf von `what()` gezeigt wird. Falls `exception`-Methoden nicht benötigt werden, muss `DateiEnde` auch nicht von `exception` erben.

Für schwere Fehler, oben angezeigt durch `bad()`, ist hier keine Fehlerbehandlungsroutine definiert. Sie werden deswegen so lange an die nächsthöhere Ebene weitergereicht, bis sie auf einen geeigneten Ausnahme-Handler treffen. Ist wie hier keiner vorhanden, wird das Programm abgebrochen. Die Ausnahme-Handler werden der Reihe nach abgefragt, sodass einer, der auf alle Fehler passt, am Ende der Liste stehen muss, um die anderen nicht zu verdecken. Das obige Programm könnte entsprechend ergänzt werden. Die Folge von drei Punkten innerhalb runder Klammern wird *Ellipse* genannt, was so viel wie Auslassung bedeutet. Nichtangabe eines Datentyps durch ... bedeutet »beliebige Anzahl beliebiger Datentypen«.

```
catch(...) { // Ellipse ... für nicht spezifizierte Fehler
     cerr << "nicht behebbarer Fehler!" << endl;
     throw;   // Weitergabe an nächsthöhere Instanz
}
```

Der Vorteil der Trennung von Fehlererkennung und -behandlung wird durch einen Verlust an Lokalität erkauft, weil von der fehlerentdeckenden Stelle an eine ganz andere Stelle gesprungen wird, die auch Anlaufpunkt vieler anderer Stellen sein kann. Es lohnt sich daher, sich vor der Programmierung eine angemessene Strategie zur Fehlerbehandlung zu überlegen.

8.1.1 Exception-Spezifikation in Deklarationen

Mit Hilfe des Schlüsselworts `noexcept` kann dokumentiert werden, ob eine Funktion Ausnahmen auswerfen kann. Dabei sind drei Fälle zu unterscheiden:

1. `void zahlen_lesen_und_ausgeben();`
   Es wird nichts ausgesagt, ob diese Funktion Ausnahmen auswerfen kann oder nicht.
2. `void zahlen_lesen_und_ausgeben() noexcept;`
   verspricht, *keine* Ausnahmen auszuwerfen.
3. `void zahlen_lesen_und_ausgeben() noexcept(false);`
   kann Ausnahmen auswerfen.

Der Benutzer einer Funktion kennt zwar ihren Prototyp, im Allgemeinen aber nicht die Implementierung der Funktion. Es kann sein, dass das Versprechen im Punkt 2 nicht eingehalten wird! In so einem Fall wird das Programm abgebrochen und damit eine spezialisierte Fehlerbehandlung unmöglich gemacht. `noexcept` sollte man selbst daher nur verwenden (wenn überhaupt), wenn man absolut sicher ist. Ohne `noexcept` hat der Aufrufer einer Funktion immer die Chance, mit `catch(...)` jede auftretende Exception abzufangen. `noexcept` wird von vielen Compilern noch nicht unterstützt.

8.1.2 Exception-Hierarchie in C++

Eigene Klassen zur Ausnahmebehandlung können sinnvoll sein, C++ stellt aber auch eine Reihe vordefinierter Exception-Klassen zur Verfügung (Abbildung 8.1).

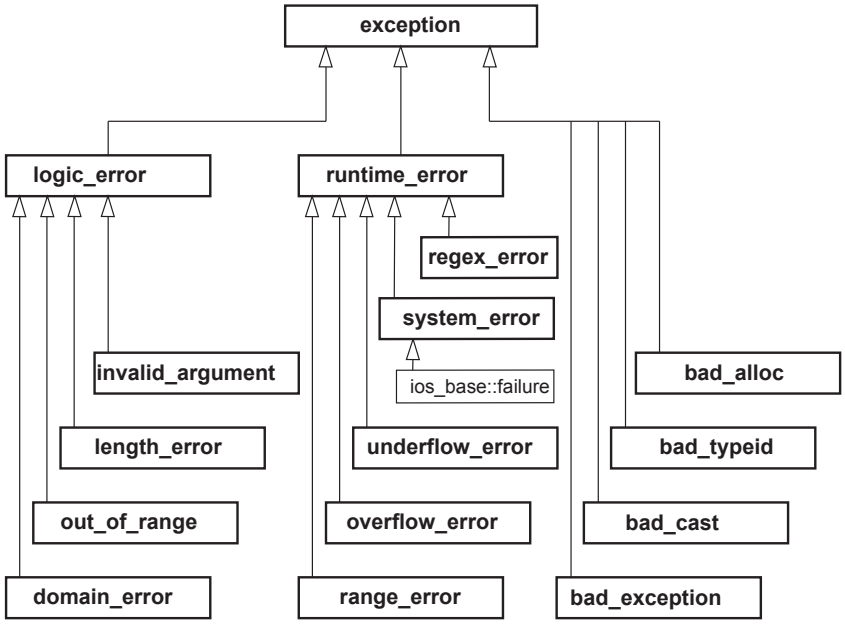

Abbildung 8.1: Exception-Hierarchie in C++

Dabei erben alle speziellen Exception-Klassen von der Basisklasse `exception`, wie Abbildung 8.1 zeigt. Die Basis-Klasse `exception` hat die folgende öffentliche Schnittstelle:

```
namespace std {
  class exception {
    public:
      exception() noexcept;
      exception(const exception&) noexcept;
      exception& operator=(const exception&) noexcept;
      virtual ~exception(); // Destruktor: wirft keine Exception
      virtual const char* what() const noexcept;
  };
}
```

Die Tabelle 8.1 zeigt die Zuständigkeit der Exception-Klassen.

Tabelle 8.1: Bedeutung der Exception-Klassen

Klasse	Bedeutung	Header
`exception`	Basisklasse	`<exception>`
`logic_error`	theoretisch vermeidbare Fehler, zum Beispiel Verletzung von logischen Vorbedingungen	`<stdexcept>`
`invalid_argument`	ungültiges Argument bei Funktionen	`<stdexcept>`
`length_error`	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet	`<stdexcept>`
`out_of_range`	Bereichsüberschreitungsfehler	`<stdexcept>`
`domain_error`	anderer Fehler des Anwendungsbereichs	`<stdexcept>`
`runtime_error`	nicht vorhersehbare Fehler, zum Beispiel datenabhängige Fehler	`<stdexcept>`
`regex_error`	Fehler bei regulären Ausdrücken	`<regex>`
`system_error`	Fehlermeldung des Betriebssystems	`<system_error>`
`range_error`	Bereichsüberschreitung	`<stdexcept>`
`overflow_error`	arithmetischer Überlauf	`<stdexcept>`
`underflow_error`	arithmetischer Unterlauf	`<stdexcept>`
`bad_alloc`	Speicherzuweisungsfehler (Details siehe Abschnitt 8.2)	`<new>`
`bad_typeid`	falscher Objekttyp (vgl. Abschnitt 7.10)	`<typeinfo>`
`bad_cast`	Typumwandlungsfehler (vgl. Abschnitt 7.9)	`<typeinfo>`

`noexcept` nach den Deklarationen bedeutet, dass die Methode verspricht, selbst keine Exception zu werfen, weil es sonst eine unendliche Folge von Exceptions geben könnte. Die von der Klasse `exception` geerbte Methode `what()` gibt einen Zeiger auf `char` zurück, der auf eine Fehlermeldung verweist. Eigene Exception-Klassen können durch Vererbung die Schnittstelle übernehmen, so wie die Klasse `logic_error`:

```
namespace std {
  class logic_error : public exception {
    public:
      explicit logic_error(const string& argument);
        // Zur Erinnerung: explicit verbietet implizite Typumwandlung.
  };
}
```

Dem Konstruktor kann ein `string`-Objekt mitgegeben werden, das eine Fehlerbeschreibung enthält, die im `catch`-Block ausgewertet werden kann.

8.1.3 Besondere Fehlerbehandlungsfunktionen[1]

Nun kann es sein, dass mitten in einer Fehlerbehandlung selbst wieder Fehler auftreten. Die dann aufgerufene Funktion `terminate()` wird in diesem Abschnitt beschrieben. Das hier beschriebene Verhalten wird von manchen Compilern nur bedingt unterstützt. Daher ist es empfehlenswert, die Systemdokumentation zu Rate zu ziehen. Das ganze

[1] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

Thema »Exception Handling« kann hier nicht annähernd vollständig dargestellt werden. Eine ausführliche und gut lesbare Auseinandersetzung damit findet sich in [ScMb]. `terminate()` ist im Header `<exception>` deklariert.

terminate()

Die Standardimplementierung von `void terminate()` beendet das Programm. Die Funktion `terminate()` wird (unter anderem) aufgerufen, wenn

- der Exception-Mechanismus keine Möglichkeit zur Bearbeitung einer geworfenen Exception findet;
- ein Destruktor während des Aufräumens (englisch *stack unwinding*) eine Exception wirft oder wenn
- ein statisches (nichtlokales) Objekt während der Konstruktion oder Zerstörung eine Exception wirft.

Benutzerdefinierte Fehlerbehandlungsfunktion

Die oben dargestellte Funktion kann bei Bedarf selbst definiert werden, um die vorgegebene zu ersetzen. Dazu ist standardmäßig ein Typ für Funktionszeiger definiert:

```
typedef void (*terminate_handler)();
```

Dazu passend gibt es eine Funktion, der ein Zeiger auf die selbst definierte Funktion dieses Typs übergeben wird, um die vorgegebene zu ersetzen:

```
terminate_handler set_terminate(terminate_handler f) noexcept;
```

Übergeben wird der Zeiger auf eine selbst definierte Funktion des Typs `terminate_handler`, die das Programm ohne Rückkehr an den Aufrufer beendet.

8.1.4 Erkennen logischer Fehler

Mit einem »logischen Fehler« ist gemeint, dass die gewünschte Semantik des Programms nicht korrekt in Programmcode umgesetzt worden ist, vermutlich durch einen »Denkfehler« des Autors. Wie können logische Fehler in einer Funktion oder allgemein in einem Programm gefunden werden, insbesondere während der Entwicklung? Dies ist möglich, indem in das Programm eine *Zusicherung* einer logischen Bedingung (englisch *assertion*) eingebaut wird. Das auf Seite 133 beschriebene Makro `assert()` dient diesem Zweck. Das Argument des Makros nimmt eine logische Annahme auf. Ist die Annahme wahr, passiert nichts; ist sie falsch, wird das Programm mit einer Fehlermeldung abgebrochen. Sämtliche logischen Überprüfungen mit `assert`-Makros werden auf einmal abgeschaltet, wenn die Compilerdirektive `#define NDEBUG` vor dem `assert()`-Makro definiert wurde. Damit entfallen aufwendige Arbeitsvorgänge mit dem Editor für auszuliefernde Software, die keinen Debug-Code mehr enthalten soll.

Der Nachteil dieses Makros besteht in dem erzwungenen Programmabbruch. Wenn an die Stelle eines Abbruchs eine besondere Fehlerbehandlung (mit oder ohne Programmabbruch) treten soll, kann einfach ein eigenes Makro unter Benutzung der Ausnahmebehandlung geschrieben werden, das zum Beispiel in der Datei *assertex.h* abgelegt wird:

Listing 8.2: Zusicherungs-Makro mit Exception

```
// cppbuch/k8/logik/assertex.h
#ifndef ASSERTEX_H
#define ASSERTEX_H
  #ifdef NDEBUG
    #define Assert(bedingung, ausnahme) ; // Leeranweisung
  #else
    #define Assert(bedingung, ausnahme) \BS
          if(!(bedingung)) throw ausnahme
  #endif
#endif // ASSERTEX_H
```

Das folgende Programm zeigt beispielhaft, wie das Makro benutzt wird. Wenn 1 eingegeben wird, endet das Programm normal. Wenn 0 eingegeben wird, gibt es zusätzlich eine Fehlermeldung. In allen anderen Fällen wird das Programm mit einer Fehlermeldung abgebrochen. Es wird gezeigt, wie Exception-Objekten Informationen mitgegeben werden können, die innerhalb des Objekts ausgewertet werden. In diesem Fall ist es nur die Entscheidung, ob die übergebene Zahl gleich 0 oder gleich 1 ist.

Listing 8.3: Assert-Makro mit Exception

```
// cppbuch/k8/logik/main.cpp
#include<iostream>
#include<cstdlib> // exit()
// ggf. Abschalten der Zusicherungen mit NDEBUG
// #define NDEBUG
#include"assertex.h"

class GleichNull { // Exception-Klasse ohne Konstruktor-Argument
   public:
     const char* what() const {
        return "Fehler GleichNull entdeckt";
     }
};

class UngleichEins { // Exception-Klasse mit Konstruktor-Argument
   public:
     UngleichEins(int i) : zahl(i) {}
     const char* what() const {
        return "Fehler UngleichEins entdeckt";
     }
     int wieviel() const { return zahl;}
   private:
     int zahl;
};

using nameapace std;
int main() {
    int i;
    cout << "0          : GleichNull-Fehler\n"
         << "1          : normales Ende\n"
         << "!= 1       : UngleichEins-Fehler\n i = ?";
    cin >> i;
```

```
    try {
        Assert(i, GleichNull());       // wirft ggf. Exception
        Assert(i == 1, UngleichEins(i)); // wirft ggf. Exception
    }
    catch(const GleichNull& fehlerObjekt) {
       cerr << fehlerObjekt.what() << endl
            << "keine weitere Fehlerbehandlung\n";
    }
    catch(const UngleichEins& fehlerObjekt) {
       cerr << fehlerObjekt.what() << endl
            << fehlerObjekt.wieviel() << '!'
            << " Abbruch"           << endl;
       exit(1);                  // Programmabbruch
    }
    cout << "normales Programmende mit i = " << i << endl;
}
```

8.1.5 Arithmetische Fehler / Division durch 0

Es kann sein, dass eine arithmetische Operation nicht erlaubt ist, wie etwa die Division durch 0. Der C++-Standard sieht dafür keine Exception vor: »If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.« [ISOC++, Kap. 5] (Falls während der Auswertung eines Ausdrucks das Ergebnis mathematisch undefiniert oder nicht durch seinen Datentyp darstellbar ist, ist das Verhalten undefiniert.) Das folgende Beispiel einer Ganzzahl-Division durch 0 funktioniert daher im Allgemeinen nicht.

```
int a = 7;
int b = 0;
try {
   a = a / b;
} catch(...) {   // fängt den Fehler nicht ab!
    cerr << "Division durch 0!";
}
```

Um C++-konform zu bleiben, muss man sich selbst darum kümmern, indem zum Beispiel der Nenner vor der Division auf 0 geprüft wird. Nur manche Compiler stellen Hilfsmittel zur Erkennung zur Verfügung, die von der Art der CPU abhängen. Bei Gleitkommazahlen stellt die Programmiersprache C, die ein Teil von C++ ist, einen Mechanismus zur Verfügung. Bei einer fehlerhaften Operation wird ein Flag gesetzt, das abgefragt werden kann. Nach der Abfrage sollte es gleich wieder gelöscht werden, um für nachfolgende Operationen bereit zu sein. Das folgende Programm zeigt nur die Arbeit mit dem Flag für die Division durch 0. Weitere Einzelheiten sind dem Abschnitt 7.6 des C-Standards [ISOC] zu entnehmen.

Listing 8.4: Test auf Division durch 0

```
// cppbuch/k8/division0.cpp
#include<iostream>
#include<cfenv>
using namespace std;
```

```
float f(float a, float b) {
   float c = a/b;
   if(fetestexcept(FE_DIVBYZERO)) { // abfragen
      feclearexcept(FE_DIVBYZERO); // zurücksetzen
      throw("Division durch 0!");
   }
   return c;
}

int main() {
   float z;
   float n;
   cout << "Zähler :";
   cin >> z;
   cout << "Nenner :";
   cin >> n;
   try {
      cout << "Ergebnis = " << f(z, n) << endl;
   } catch(const char* ex) {
      cerr << ex << endl;
   }
}
```


8.2 Speicherbeschaffung mit new

Eine Möglichkeit, das Fehlschlagen des `new`-Operators festzustellen, war früher die Abfrage, ob ein Null-Zeiger zurückgegeben wird. Es ist umständlich, nach jedem `new` diese Prüfung durchzuführen. Das C++-Standardkomitee hat sich deshalb entschlossen, dass bei Fehlschlagen der Speicherbeschaffung eine Ausnahme ausgeworfen wird, nämlich ein Objekt vom Typ `bad_alloc`. Die Alternative, einen Nullzeiger zurückzugeben, bleibt bestehen (siehe Ende des Abschnitts). In C++ gibt es einen Zeiger namens `new_handler` auf eine Funktion, die vom `new`- oder `new[]`-Operator gerufen wird, wenn die Speicherplatzanforderung nicht erfüllt werden kann. Der Zeiger ist wie folgt definiert:

```
typedef void (*new_handler)();
```

Zunächst zeigt `new_handler` auf eine Standardfunktion, die gegebenenfalls eine Ausnahme auswirft:

```
try {
     int *vielSpeicher = new int[30000];
}
catch(const bad_alloc&) {
    cerr << "kein Speicher vorhanden!" << endl;
    exit(1);
}
// ... normale Fortsetzung des Programms
```

Ohne try- und catch-Blöcke würde das Programm bei Speichermangel mit der Fehlermeldung »abnormal program termination« oder etwas Ähnlichem beendet. Der new_handler-Zeiger kann jedoch auf eine selbst definierte Funktion gerichtet werden, die Speicher beschaffen soll, wenn möglich. Dann wird im Fehlerfall diese Funktion aufgerufen, entsprechend der vierten in der Einführung zu diesem Kapitel genannten Möglichkeit. Der new-Operator arbeitet etwa auf folgende Weise:

```
void* ergebnis;
do {
    ergebnis = beschaffe_irgendwie_Speicher();
    if(ergebnis == NULL) {     // d.h. nicht erfolgreich
       if(new_handler == NULL) { // d.h. nicht definiert
          throw bad_alloc();
       }
       else {
          new_handler();        // Aufruf
       }
    }
} while(ergebnis == NULL);     // d.h. nicht erfolgreich
return ergebnis;
```

Die Schleife wird beendet, wenn es entweder gelungen ist, Speicher zu beschaffen, oder wenn kein Rücksprung aus (*new_handler)() erfolgt, weil eine Ausnahme ausgeworfen wurde. Im folgenden Programm wird der new_handler-Zeiger mithilfe der Bibliotheksfunktion set_new_handler() auf eine selbst definierte Funktion speicherfehler() gerichtet, wobei set_new_handler() einen Zeiger auf die vorher zugewiesene Funktion zurückgibt. Im Verlauf des main()-Programms wird ein Array von Zeigern angelegt.

Jedem Zeiger soll in der Schleife ein großer Block zugewiesen werden. Irgendwann ist jedoch der verfügbare Speicherplatz erschöpft. In diesem Moment wird während der Ausführung von new die Funktion (*new_handler)(), also Speicherfehler(), aufgerufen. Die Funktion schafft mit delete Platz und deaktiviert sich selbst, weil sie nicht noch mehr Speicher beschaffen kann. Dann kehrt sie nach new zurück, wo sofort noch einmal versucht wird, den Speicherplatz zu beschaffen, was dieses Mal gelingt. Die Schleife läuft weiter. Irgendwann stellt sich wieder das Problem des knappen Speichers. Weil speicherfehler() nun eine Ausnahme auswirft, wird das Programm beendet. Falls in speicherfehler() *kein* Platz beschafft werden kann, ist durch Erzeugen einer Ausnahme das Programm abzubrechen, weil es sonst eine unendliche Schleife gibt, indem abwechselnd erfolglos versucht wird, Speicherplatz zu beschaffen und die Fehlerbehandlungsfunktion aufzurufen (siehe Programmbeispiel obiges zur Arbeitsweise von new).

Listing 8.5: Beispiel mit new_handler

```
// cppbuch/k8/speicher/newhdl.cpp
#include<iostream>
#include<new>  // set_new_handler() und bad_alloc
using namespace std;
const unsigned int BLOCKGROESSE = 64000, MAXBLOECKE = 50000;
int* reserveSpeicher = 0;

void speicherfehler() throw (bad_alloc) {
   cerr <<"Memory erschöpft! speicherfehler() aufgerufen!\n";
```

```
    if(reserveSpeicher) {
       cerr << "Einmal Platz schaffen!\n";
       delete [] reserveSpeicher;
       reserveSpeicher = 0;              // deaktivieren
    }
    else {
       cerr << "Exception auslösen!\n";
       throw bad_alloc();
    }
}

int main() {
    // eigene Fehlerbehandlungsfunktion eintragen
    set_new_handler(Speicherfehler);
    reserveSpeicher = new int[10*BLOCKGROESSE]; // Speicher belegen
    int* ip[MAXBLOECKE] = {0};
    unsigned int blockNr = 0;
    try {
       while(blockNr < MAXBLOECKE) { // Speicher fressen
            ip[blockNr] = new int[BLOCKGROESSE];
            cout << "Block " << blockNr << " beschafft" << endl;
            ++blockNr;
       }
       if(blockNr == MAXBLOECKE-1) { // Zählung ab 0
            cout << "Block-Array erschöpft" << endl;
       }
    }
    catch(const bad_alloc& exc) {
       cerr << ++blockNr
            << " Blöcke beschafft\n"
            << "bad_alloc ausgeworfen! Grund: "
            << exc.what() << endl;
    }
}
```

Das früher übliche Standardverhalten, bei Speichermangel NULL zurückzugeben, kann durch ein `nothrow`-Argument des `new`-Operators erreicht werden:

```
// p1 ist nie Null, bei Speichermangel wird eine bad_alloc-Exception ausgeworfen:
T *p1 = new T;
// bei Speichermangel wird Null zurückgegeben, eine Exception kann nicht erzeugt werden:
T *p2 = new(nothrow) T;
if(!p2) {
   cerr << "zuwenig Speicher!" << endl;
}
else {
  // ...
}
```

8.3 Exception-Sicherheit

Ein Programm ist exception-sicher (englisch *exception safe*), wenn Laufzeitfehler keine nachteiligen Auswirkungen haben. Um den Begriff zu präzisieren, gibt es verschiedene Stufen:

1. Auf der niedrigsten Stufe gibt es keinerlei Zusicherungen, ob und wie sich Fehler auswirken.
2. In dieser Stufe kann ein Programm zwar falsche Daten erzeugen, es soll aber weder abstürzen noch Speicherlecks oder verwitwete Objekte hinterlassen.
3. Eine starke Exception-Sicherheit liegt vor, wenn eine Operation entweder vollständig oder, bei einem Fehler, gar nicht oder so ausgeführt wird, dass das betroffene Objekt anschließend im selben Zustand wie vor der fehlerhaften Operation ist. Dies entspricht einem Commit bzw. Rollback in der Datenbank-Terminologie. Ein Beispiel ist die folgende Funktion zum Ablegen eines Elements auf einem Stack (Variante der Funktion von Seite 247):

   ```
   template<typename T>
   void SimpleStack<T>::push(const T& x) {
       if(full()) {
          throw std::logic_error("Stack-Overflow!");
       }
       array[anzahl++] = x;
   }
   ```

   Wenn noch Platz auf dem Stack ist, hat `push()` die gewünschte Wirkung. Wenn nicht, bleibt der Stack in seinem vorherigen Zustand.
4. In der sichersten Stufe wird garantiert, dass keine Fehler auftreten oder aber alle auftretenden Fehler abgefangen werden, sodass kein Fehler nachteilige Auswirkungen hat. Möglicherweise auftretende Exceptions werden nicht zum Aufrufer durchgereicht.

Die sicherste Stufe wird oft nicht zu realisieren sein. Selbst ein korrektes Programm kann nicht verhindern, dass es mit falschen Daten gefüttert wird. Die zweithöchste Stufe lässt sich hingegen meistens erreichen. Ob das immer wünschenswert ist, hängt vom Einzelfall ab. So kann man sich vorstellen, dass ein Objekt eine große Anzahl von Rechenergebnissen in einem Vektor speichert oder als Datei ausgibt. Wenn mitten in einer Berechnung ein Fehler auftritt, ist es vielleicht interessant, bis zu welcher Stelle die Berechnungen fehlerfrei erfolgen konnten. Ein gelöschter Vektor oder eine gelöschte Datei wären da nicht hilfreich.

Ein anderer Aspekt ist der zu treibende Aufwand und an welcher Stelle er betrieben werden soll. Nach dem »Design by Contract«-Prinzip [Mey] gewährleistet eine Funktion die Nachbedingung, wenn der Aufrufer die Vorbedingung einhält. Die Frage ist nun, ob eine Funktion jede Verletzung der Vorbedingung in den Eingabeparametern aufspüren und melden oder auf einen korrekten Aufruf vertrauen soll. Dazu zwei Beispiele:

1. Eine Funktion, die die Quadratwurzel aus einer positiven `double`-Zahl ≥ 0 berechnet, sollte eine Exception werfen, wenn sie mit einem negativen Argument aufgerufen wird. Die Prüfung kostet nicht viel Laufzeit und ist einfach durchzuführen.

2. Mit Hilfe der binären Suche einen Eintrag in einem Array zu finden, geht sehr schnell (z.B. Algorithmus `binary_search` von Seite 681). Der mittlere Aufwand ist proportional zum Logarithmus der Arraylänge *N*. Um in einem Array von 1024 Elementen einen Eintrag zu finden oder festzustellen, dass er nicht vorhanden ist, werden maximal 10 Schritte benötigt ($1024 = 2^{10}$). Vorbedingung ist, dass das Array sortiert ist. Wenn innerhalb von `binary_search()` diese Vorbedingung geprüft werden sollte, wären zusätzlich *N* Schritte notwendig, und der Performance-Vorteil wäre gänzlich dahin.

Es kommt also auf den Einzelfall an. Wenn sich eine Methode auf `binary_search()` verlässt, ohne die Sortierung des an `binary_search()` übergebenen Arrays zu prüfen, kann das betroffene Objekt in einen fehlerhaften Zustand geraten. Eine starke Exception Safety ist hier nur möglich, wenn die Sortierung des möglicherweise von außen stammenden Arrays sichergestellt ist oder überprüft wird.

Mehr dazu lesen Sie in Abschnitt 20.3.