

Programmieren in C++ (5): Ausnahmebehandlung

Silke Seehusen, Fachhochschule Lübeck

In den vorhergehenden Artikeln wurden Programme vorgestellt, in denen meistens angenommen wurde, daß bei der Abarbeitung keine Fehler und insbesondere keine unvorhergesehenen Fehler auftreten. Dies ist als Einführung didaktisch gut, aber längerfristig irreführend. Deshalb wird in diesem Artikel auf die Behandlung von Fehlern, insbesondere auf die Behandlung von Ausnahmen (*exception handling*) eingegangen.

Die Behandlung von Fehlern ist ein ungeliebtes Thema, sowohl in der Programmierung als auch in der Softwaretechnik, obwohl propagiert wird, möglichst jeder Fehler sollte erkannt und es sollte "vernünftig" darauf reagiert werden. Es ist unter anderem deshalb ein ungeliebtes Thema, weil die konsequente Fehlerbehandlung jeden Code erheblich vergrößert (bis zu 80 %, 50 % ist nichts Ungewöhnliches). Dadurch kann der Algorithmus des normalen Ablaufs sehr undurchsichtig werden.

Fehler und "Nichtfehler"

Der Begriff Fehler wird hier eingeschränkt auf Fehler, die den normalen Programmablauf "stören". Eine Negativabgrenzung erscheint mir hier angebracht:

Eine fehlerhafte Eingabe eines Benutzers ist in der Regel kein Fehler für ein Anwendungsprogramm. Es sollte z.B. den Benutzer über die fehlerhafte Eingabe informieren und zu einer wiederholten Eingabe auffordern.

Beim Traversieren einer Liste ist das Erreichen des Listenendes auch kein Fehler, sondern ein ganz normaler Zustand. Versucht ein Programm jedoch, ein Element hinter dem Listenende anzuschauen, ist es in der Regel ein Fehler.

Ein typischer Fehler ist auch, auf ein Feldelement zuzugreifen, das nicht existiert.

Ausnahmen

Als **Ausnahmen** (*exceptions*) werden Ereignisse bezeichnet, die bei der Ausführung eines Programms auftreten und die nicht im normalen Kontrollfluß des Programms behandelt werden. Es ist eine typische Ausnahme, wenn bei einer Gleitkommaoperation ein Überlauf stattfindet. Eine andere Ausnahme tritt z.B. auf, wenn die Kommunikationsverbindung zwischen zwei Prozessen während eines Datenaustausches abbricht.

Ausnahmebehandlung (*exception handling*) ist die Bearbeitung solcher Ereignisse innerhalb des Programms und die Unterstützung durch die Programmiersprache und das Laufzeitsystem.

Ausnahmen sind somit eine spezielle Art von Fehlern und Ausnahmebehandlung eine spezielle Fehlerbehandlung.

Die Abgrenzung zwischen "normalen" Fehlern und Ausnahmen ist unscharf und insbesondere anwendungsabhängig.

Im folgenden werden nur Fehler und Ausnahmen behandelt, die während der Programmausführung auch entdeckt werden.

Fehlerbehandlung

Wie wird in normalen Programmen, z.B. C-Programmen, mit Fehlern umgegangen, die bei der Ausführung einer Funktion erkannt werden? Es gibt klassisch die folgenden Möglichkeiten:

- Der Fehler wird **ignoriert**. Dies geschieht anscheinend in der Hoffnung, daß es *niemand merkt* oder daß es egal ist, welches Ergebnis das Programm produziert. (In einigen zur Zeit weit verbreiteten Programmen scheint diese Fehlerbehandlung vorherrschend zu sein.)
- Der Fehler wird **entdeckt**, und das Programm wird sofort **abgebrochen**. Dies ist auch eine sehr weit verbreitete Fehlerbehandlung, die wesentlich besser als das pure Ignorieren ist. Dennoch wäre oft ein "sanfter" Ausstieg aus dem Programm angenehmer, der noch einige Aufräumarbeiten, wie Ressourcen freigeben, durchführt. Auch ist eventuell ein Versuch der Rekonstruktion möglich.
- Die aufgerufene Funktion, die auf den Fehler stößt, **liefert** einen **Fehlercode zurück**. Die Frage ist hier, ob der aufrufende Kontext den Rückgabewert abprüft.
- Es wird ein **globaler Fehlercode** gesetzt, z.B. *errno* der C-Standardbibliothek. Auch hier muß eigentlich nach jedem Funktionsaufruf der Fehlercode abgeprüft werden. Das ist mühselig und eventuell sogar aufwendig hinsichtlich der Laufzeit.
- Es wird eine **spezielle Fehlerbehandlungsroutine**, z.B. mit *setjmp* und *longjmp*, aufgerufen. Eine geschachtelte Fehlerbehandlung wird damit direkt aber nicht unterstützt. Soll ein Anwendungsprogramm Ausnahmen behandeln können, die in einer Bibliothek entdeckt werden, so muß es dazu an irgend einer Stelle Funktionen hinterlegen. Das verkompliziert die Schnittstelle zur Bibliothek und wird deshalb auch selten genutzt.

Ist ein Fehler eine wirkliche Ausnahme, das heißt, tritt er selten und kaum vorhersehbar auf, sind die obigen Fehlerbehandlungsarten nicht adäquat.

Das gilt insbesondere, wenn ein Teilsystem oder eine Bibliothek entwickelt wird, die in verschiedenen Kontexten verwendet wird, die nicht einmal bei der Entwicklung bekannt sind. Den (unbekannten) Anwendungsprogrammen sollte dennoch die Möglichkeit gegeben werden können, auf Ausnahmen zu reagieren.

Ausnahmebehandlung in C++

Für die Ausnahmebehandlung bietet C++ einen Mechanismus an, von der Stelle in einer Funktion, in der eine Ausnahme auftritt, die Kontrolle und Informationen an einen unbestimmten Aufrufer zu übergeben, der seine Bereitschaft erklärt hat, Ausnahmen eines bestimmten Typs zu behandeln.

Die Ausnahmebehandlung wird im folgenden an einigen kleinen Beispielen eingeführt. Auch dieser Artikel steht in dem Konflikt, daß ein Konzept vorgestellt wird, das erst für große Softwaresysteme interessant wird, auf ein paar Seiten aber nur kleine, einfache Beispiele durchgesprochen werden können.

Als Beispielklasse, in der Ausnahmen erkannt werden, wird ein Auszug der Klasse *Vector* vorgestellt, die ein eindimensionales Feld mit Indexüberprüfung bereitstellt. Der Elementtyp ist zunächst mit *double* festgelegt:

```
typedef double ITEM;

class Vector
{
public:
    Vector ( int size );
    ITEM & operator[] ( int );
```

```

    int    size() const;
    // ..
private:
    ITEM * _vector; // Feld
    int    _size;
           // Elementanzahl
};

```

Zunächst wird nur ein Ausnahmetyp, *RangeError*, eingeführt:

```

class RangeError {};
    // Bereichs-Fehler

```

Ein Ausnahmetyp wird durch eine ganz normale Klassendefinition festgelegt, der außer am Namen nicht angesehen werden kann, daß es ein Ausnahmetyp ist.

Ausnahmen melden

Wenn eine Ausnahme erkannt wird, muß sie **gemeldet** werden, in der Originalliteratur wie in [1] heißt es, eine Ausnahme wird **ausgeworfen**.

Im folgenden wird bei einer Indexunter- oder -überschreitung eine Ausnahme des Typs *RangeError* mit der Anweisung *throw RangeError()* gemeldet, wobei der Operand von *throw* wie ein Operand von *return* angesehen wird. In der Anweisung wird durch den Konstruktoraufruf *RangeError()* ein temporäres Objekt erzeugt.

```

ITEM &
Vector::operator[] ( int i )
{
    if ( 0 <= i && i < _size )
        return _vector[i];
    throw RangeError();
}

```

Wird die Klasse *Vector* sinnvoll benutzt, gibt es immer sinnvolle Resultate, und der Programmablauf ist normal:

```

int main()
{
    Vector v(10);
    for( int i=0; i < v.size();
        i++ )
    {
        v[i] = i;
    }
    return 0;
}

```

Schleicht sich ein Programmierfehler ein, den C++-Anfänger (ohne C-Vorkenntnisse) gerne machen:

```

int main()
{
    Vector v(10);
    for ( int i=0; i<=10; i++ )
    {
        v[i] = i;
    }
    return 0;
}

```

so wird bei *v[10]* eine Ausnahme gemeldet. Da sie vom aufrufenden Kontext, hier dem Hauptprogramm, nicht "abgefangen" wird, kommt die voreingestellte Ausnahmebehandlung zum Zuge: Der Laufzeitkeller wird abgeräumt (*unwinded*), und das Programm wird durch Aufruf der Funktion *terminate()*, die ihrerseits mit *abort()*

voreingestellt ist, abgebrochen.

Soweit haben wir nicht mehr erreicht als einen Programmabbruch bei Auftreten einer Ausnahme.

Ausnahmen abfangen

Die Idee der Ausnahmebehandlung in C++ besteht nun darin, daß der aufrufende Kontext Ausnahmen **abfangen** kann. Dazu muß der Kontext die Bereitschaft mit einer **try-Anweisung** angeben. Eine *try*-Anweisung besteht aus dem Schlüsselwort *try*, gefolgt von dem Block, für den Ausnahmen abgefangen werden sollen. Beispiel:

```
int main()
{
    Vector v(10);

    try
    {
        for (int i=0; i<=10; i++)
            v[i] = i;
    }
    catch ( RangeError )
    {
        cerr << "irgendwo in der "
             << "try-An\%wei\%sung ist "
             << "eine Ausnahme "
             << "aufgetreten"
             << endl;
        return 1;
    }
    return 0;
}
```

In diesem Fall wird die Ausnahme durch die direkt folgende *catch*-Anweisung abgefangen. Es wird eine Fehlermeldung ausgegeben und *return 1* ausgeführt. Die *catch*-Anweisung wird auch **Handler** der Ausnahme genannt, wenn sie, wie hier, die Ausnahme abfängt und behandelt. Eine *catch*-Anweisung beginnt mit dem Schlüsselwort *catch*, das von einer Ausnahmedeklaration in runden Klammern und einem Block gefolgt wird.

Würde keine Ausnahme gemeldet werden, z.B. wenn das Feld mit *Vector v(11)* deklariert wäre, würde die *catch*-Anweisung übersprungen, und das Programm endet normal.

Wenn in einer *try*-Anweisung eine Ausnahme auftritt, kann diese in einer der direkt folgenden *catch*-Anweisungen abgefangen werden, wenn dort eine passende *catch*-Anweisung gefunden wird. Eine *catch*-Anweisung ist passend, wenn der Typ des folgenden Ausdrucks in Klammern derselbe Typ wie die gemeldete Ausnahme oder ein Basistyp der gemeldeten Ausnahme ist.

Eine *try*-Anweisung mit allen direkt folgenden *catch*-Anweisungen wird als **try-Block** bezeichnet. Eine *throw*-Anweisung wird auch **Meldepunkt** (*throw-point*) einer Ausnahme genannt.

Gibt es keine passende *catch*-Anweisung in einem *try*-Block, wird die Ausnahme an den aufrufenden Kontext des *try*-Blocks gemeldet.

Datenübergabe an einen Handler

Wird eine Ausnahme gemeldet, so können an den abfangenden Handler auch Daten übergeben werden, die in der gemeldeten Ausnahme gekapselt sind. Als Beispiel ist es bei der Indexüberprüfung sinnvoll, den Wert des Index zu übergeben, der die Ausnahme verursacht. Dazu muß die Klassendefinition der Ausnahme erweitert

werden zu:

```
class RangeError
{
public:
    int index;
    RangeError(int i=0)
        : index(i) {}
};
```

Diese Daten werden bei der Meldung der Ausnahme bereitgestellt, indem der Konstruktor der Ausnahme mit den entsprechenden Parametern aufgerufen wird:

```
ITEM &
Vector::operator[] ( int i )
{
    if ( 0 <= i && i < _size )
        return _vector[i];
    throw RangeError ( i );
}
```

Diese Daten werden in der *catch*-Anweisung abgefangen durch:

```
catch ( RangeError & ex )
{
    cerr << "Indexfehler mit"
          << " Index = "
          << ex.index
          << endl;
    return 1;
}
```

Die Ausnahmedeklaration hinter *catch* ist wie die Deklaration eines Funktionsarguments aufgebaut. Der Variablen *ex* wird hier eine Referenz des beim Melden der Ausnahme erzeugten temporären Objekts zugewiesen. Es ist guter Stil, aber nicht zwingend, in der Ausnahmedeklaration eine Referenz anzugeben. Das erlaubt später auf einfache Art, Ausnahmeobjekte von abgeleiteten Typen zu behandeln.

Ein Handler kann, aber muß die Daten nicht benutzen, so daß die Hauptprogramme der vorigen Abschnitte auch mit der erweiterten Ausnahmedefinition korrekt sind.

Die Übergabe von Daten in einem Ausnahmeobjekt ist, neben dem unschönen Setzen von globalen Variablen, die einzige Möglichkeit, Daten vom Meldepunkt zum Abfangpunkt in einem Handler zu übergeben, da beim Abräumen des Kellers die Destruktoren aller automatischen Objekte aufgerufen werden.

Melden verschiedener Ausnahmen

Das Melden einer Ausnahme transferiert die Kontrolle zu einem Handler. Dabei bestimmt der Typ des übergebenen Ausnahmeobjekts, welcher Handler die Ausnahme abfängt.

Als Beispiel wird die Klasse *Vector* so erweitert, daß bei einer Konstruktion eines Objekts der Klasse *Vector* überprüft wird, ob vorgegebene Grenzen eingehalten werden und ob ein Objekt hinreichender Größe konstruiert werden kann. Dazu werden zunächst weitere Ausnahmetypen neben *RangeError* definiert:

```
class SizeError
{
    // falsche Groesse
public:
    int size;
    int maxsize;
    SizeError (int aSize = 0,
```

```

        int max    = 0)
    : size(aSize),
      maxsize(max) {}
};
class StorageError {};
        // Speicherplatz-
        // probleme

```

Hinter einer *try*-Anweisung kann eine Folge von *catch*-Anweisungen stehen, u.a. zusätzlich in der *Vector*-Anwendung:

```

catch (StorageError)
{ // keine Ahnung, was zu
  // tun ist. Vielleicht
  // weiss es eine andere.
  throw;
}

catch ( SizeError  &ex )
{
  cerr
    <<"Size error"
      ", size = "
    << ex.size
    << " maxsize = "
    << ex.maxsize
    << endl;
  exit ( 99 );
}

```

Nach dem Melden einer Ausnahme werden die *catch*-Anweisungen in der Reihenfolge ihres Vorkommens überprüft. Die erste Anweisung, deren Ausnahme paßt, wird ausgeführt und alle weiteren ignoriert, die direkt dahinterstehen.

Mit der Anweisung *throw* ohne Ausnahmetyp wird dieselbe Ausnahme noch einmal gemeldet, in der Hoffnung, daß der aufrufende Kontext des gesamten *try*-Blocks (einschließlich Handler) darauf reagieren kann. So werden Ausnahmen explizit an umschließende *try*-Blöcke weitergereicht. *try*-Blöcke können beliebig tief geschachtelt sein.

Konstruktoren und Destruktoren

Während die Kontrolle vom Meldepunkt zu einem Handler transferiert wird, werden die Destruktoren für alle automatischen Objekte aufgerufen, die seit Eintritt in den *try*-Block konstruiert worden sind.

Für ein Objekt, das teilweise konstruiert worden ist, werden nur die Destruktoren für die vollständig erzeugten Teilobjekte aufgerufen. Bei einem Feld werden auch nur die Destruktoren für die erzeugten Objekte aufgerufen.

Der Aufruf der Destruktoren erlaubt, Ressourcen wieder freizugeben, die mit einem Konstruktor angefordert worden sind. Im folgenden Beispiel soll eine bestimmte Sperre des Typs

```

class Lock
{
public:
  int  grab();
  int  release();
  // ...
};

```

beim Transfer der Kontrolle zu einem Handler wieder freigegeben werden. Dazu wird die Sperre gekapselt:

```

class GrabLock

```

```

{
  Lock & _lock;
public:
  GrabLock ( Lock & aLock )
  : _lock( aLock )
  {
    _lock.grab();
  }
  ~GrabLock()
  { _lock.release(); }
};

```

Sei jetzt irgendwo im Programm der Zugriff auf eine zentrale Ressource mit der Sperre

```
Lock wichtig;
```

synchronisiert. Dann wird durch *GrabLock* gewährleistet, daß die Sperre vom Destruktor von *GrabLock* freigegeben wird, auch wenn Ausnahmen auftreten:

```

GrabLock mein ( wichtig );
// ..
throw exception
("in gesperrter Umgebung");

```

Dies ist eine große Hilfe beim Vorbereiten einer Rekonstruktion für den Handler. Der sinnvollen Definition von Destruktoren in C++ kommt in Zusammenhang mit der Ausnahmebehandlung noch mehr Bedeutung als ohnehin schon zu.

Hierarchie von Ausnahmen

Es sollte nicht jede Klasse ihre eigene Menge von Ausnahmetypen definieren, wie es oben in der Einführung zur Demonstration getan wurde. Statt dessen sind die Teilsysteme, die beim Entwurf der Systemarchitektur eines Softwarepakets identifiziert werden, die Einheiten, für die jeweils ein Ausnahmetyp definiert wird, von dem gegebenenfalls spezielle Ausnahmen abgeleitet werden.

Als Beispiel gebe es eine mathematische Bibliothek, für die es den Ausnahmetyp *Matherror* und davon abgeleitete Typen gibt, siehe Abbildung 1.

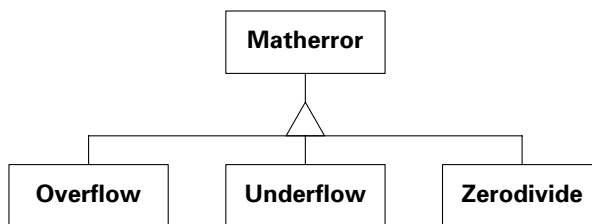


Abbildung 1: Hierarchie von Ausnahmen

Beim Melden einer Ausnahme wird nach wie vor exakt angegeben, welche Ausnahme aufgetreten ist. Die Handler können jetzt jedoch auf z.B. *Overflow* und *Underflow* getrennt oder für alle Ausnahmen vom Typ *Matherror* und deren abgeleitete Ausnahmen zusammen reagieren:

```

int i = 10000;
try
{
  g(i);
}
catch ( Overflow & oo )
{

```

```

    // versuch's kleiner
    g(i/2);
}
catch ( Matherror & mm )
{
    // alle anderen Matherror
    cerr
    << "Underflow, "
        "Zerodivide oder "
        "Matherror "
        "aufgetreten"
    << endl;
    throw;
}

```

Im obigen Beispiel behandelt der *Overflow*-Handler alle Ausnahmen von Typ *Overflow* und der *Matherror*-Handler alle Ausnahmen vom Typ *Matherror* und von einem öffentlich von *Matherror* abgeleiteten Typ, inklusive *Underflow* und *Zerodivide*.

Ein anderer Stil der Ausnahmebehandlung ist, nur die Basisklasse aufzufangen und sich auf virtuelle Funktionen der abgeleiteten Klassen zu verlassen. Denn zu einem Ausnahmetyp können wie zu jeder Klasse auch Funktionen beliebigen Typs gehören.

Es ist ein Fehler, einen Handler einer Basisklasse vor den Handler einer abgeleiteten Klasse anzuordnen, weil dann der Handler der abgeleiteten Klasse nie aufgerufen würde.

Ein ... in dem Ausnahmeausdruck einer *catch*-Anweisung steht für jede beliebige Ausnahme. Ein solcher Handler muß dann natürlich der letzte einer Handlerfolge sein. Beispiel:

```

catch ( ... ) { exit(99); }

```

Wenn kein passender Handler in einem Programm gefunden wird, wird die Funktion *terminate()* aufgerufen.

Ausnahmespezifikation

Die Ausnahmen, die eine Funktion nach außen melden kann, sind ein Teil ihrer Schnittstelle. Deshalb ist es sinnvoll, in der Spezifikation einer Funktion auch diese Ausnahmen anzugeben. Aus der Spezifikation kann dann eine Software-Entwicklerin erkennen, auf welche Ausnahmen überhaupt reagiert werden kann.

Als Beispiel erweitern wir den Selektoroperator [] der Klasse *Vector*:

```

ITEM & operator[] ( int )
    throw( RangeError );

```

Der Operator darf jetzt nur Ausnahmen des angegebenen Typs melden.

Ein Versuch einer Funktion, eine Ausnahme auszuwerfen, die nicht in ihrer Ausnahmespezifikation steht, führt zu einem Aufruf der Funktion *unexpected()*.

Eine Funktion ohne Ausnahmespezifikation kann jede beliebige Ausnahme melden.

```

int f();
    // f() wirft beliebige
    // Ausnahmen aus.
int g() throw();
    // g() wirft keine
    // Ausnahme aus.

```

Die Ausnahmespezifikation gehört nicht zum Typ einer Funktion.

terminate und unexpected

Die Ausnahmebehandlung setzt die Funktionen *terminate()* und *unexpected()* voraus, um Fehler der Ausnahmebehandlung selbst zu behandeln. Die Funktion *terminate()* wird aufgerufen, wenn

- die Ausnahmebehandlung keinen Handler für eine gemeldete Ausnahme findet,
- die Ausnahmebehandlung den Laufzeitkeller zerstört findet oder
- ein Destruktor, der während des Abräumens des Laufzeitkellers aufgrund einer Ausnahme aufgerufen wird, selbst wieder eine Ausnahme verursacht.

terminate() ruft die letzte Funktion auf, die als Argument an die Funktion *set_terminate()* gegeben wird. Der voreingestellte Wert von *terminate()* ist *abort()*.

Wenn eine Funktion mit einer Ausnahmespezifikation eine Ausnahme meldet, die nicht in der Spezifikation vorkommt, wird die Funktion *unexpected()* aufgerufen.

Diese Funktion ruft die Funktion auf, die als letztes der Funktion *set_unexpected()* übergeben wurde. Der voreingestellte Wert ist *terminate()*.

Damit ist voreingestellt, daß ein Anwendungsprogramm abgebrochen wird, wenn eine Ausnahme gemeldet wird und das Programm keine Vorkehrungen zum Abfangen getroffen hat. So werden spätestens dann vergeßliche Personen an die Möglichkeit von Ausnahmen erinnert.

Die Voreinstellung der Funktionen *terminate()* und *unexpected()* zu verändern, ist in den seltensten Fällen eine gute Idee. Nur dann, wenn ein Teilsystem in einem völlig neuen Kontext mit vielleicht einer völlig ausgetauschten Basisschicht laufen soll, für das es eigentlich nicht konzipiert war, kann das eine Übergangslösung sein.

Hinweise

In einem Software-Entwicklungsprojekt müssen einheitliche Richtlinien

- zur Abgrenzung von normalen Fehlern und Ausnahmen,
- zur einheitlichen Behandlung von normalen Fehlern und
- zur einheitlichen Behandlung von Ausnahmen

festgelegt werden. Für relativ unabhängige Subsysteme kann es Variationen geben, z.B. für disjunkte Bibliotheken. Weitere Tips sind [2] zu entnehmen.

In der C++-Standardbibliothek sind einige Ausnahmetypen definiert, die in Abbildung 2 skizziert sind.

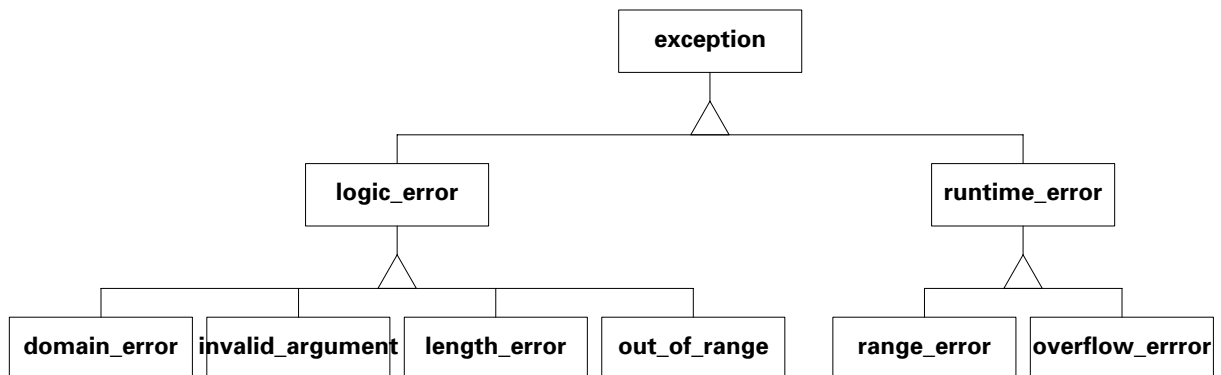


Abbildung 2: Hierarchie von Ausnahmen aus `<stdexcept>`

Für diese schon vordefinierten Ausnahmen sollten Sie sich keine neuen ausdenken. Auch kann es sehr wohl sinnvoll sein, von `exception` eine spezielle Ausnahme abzuleiten, die für Ihr Projekt fehlt.

Trotz der vielen Ausnahmen in diesem Artikel sollten sie in einem Programm nur sehr sparsam verwendet werden. Es sollte eben eine Ausnahme sein.

Zusammenfassung

Die Ausnahmebehandlung in C++ dient der Unterstützung der Fehlerbehandlung und der Implementierung von fehlertoleranten Systemen. Damit bietet C++ einen Mechanismus an, der die Programmiererinnen und Programmierer ermutigen soll, qualitativ bessere Software zu schreiben.

- Ausnahmen beliebigen Typs können gemeldet und abgefangen werden.
- Eine Funktion kann die Menge der Ausnahmen festlegen, die sie melden kann.
- Es wird ein vordefiniertes Terminierungsmodell bereitgestellt.
- Es liegt in der Entscheidung der Anwendung, welche Ausnahmen wie abgefangen werden.
- Durch geschachtelte *try*-Anweisungen kann an verschiedenen Stellen in einem Softwaresystem ein Netz zum Abfangen von Ausnahmen aufgespannt werden.
- Da die Ausnahmebehandlung relativ spät in die Sprache C++ aufgenommen und noch später von Übersetzern unterstützt worden ist, wird sie in momentan verfügbaren Implementierungen von Bibliotheken kaum benutzt. Das ändert sich hoffentlich.

Quellen

Die vollständigen Programme befinden sich auf dem FTP-Server *ftp.informatik.uni-osnabrueck.de* unter *pub/hanser/um*.

Literatur

- [1] B. Stroustrup *Die Programmiersprache C++ (2. Auflage)*, Addison-Wesley, Bonn, 1992.
- [2] R. B. Murray *C++ Strategies and Tactics*, Addison-Wesley, Reading, MA, 1993.