

12 Ausnahmebehandlung

Die Behandlung von Laufzeitfehlern bei der Erstellung von Bibliotheken stellt ein schwieriges Problem dar. C++ bietet zusätzlich zu den üblichen Methoden der Fehlerbehandlung das Konzept der Ausnahmen, das in diesem Kapitel beschrieben wird.

12.1 Einführung

Die Behandlung von Laufzeitfehlern bei der Erstellung von Bibliotheken ist problematisch: Der Entwickler der Bibliothek kann solche Fehler leicht entdecken, weiß aber in der Regel nicht, wie mit dem Fehler umgegangen werden soll. Der Anwender der Bibliothek weiß zwar, was er tun muß, wenn ein Fehler auftritt, hat aber keine direkte Möglichkeit den Fehler zu entdecken. Traditionell existieren verschiedene Strategien, dieses Problem zu lösen:

1. Entdeckt eine Bibliotheksroutine einen Laufzeitfehler, so wird das Programm abgebrochen.
2. Die Funktion, die den Fehler entdeckt liefert einen Wert zurück, der den Fehlerzustand anzeigt. Diese Methode funktioniert natürlich nur, wenn es einen Rückgabewert gibt, der ausschließlich für die Anzeige des Fehlerzustands reserviert ist.
3. Der Fehler wird einfach ignoriert.
4. Aufruf einer für den Fehlerfall vorgesehenen Funktion.

Wie in jeder anderen Sprache, so ist es natürlich auch in C++ möglich, die Strategien 1–3 zu implementieren. Darüber hinaus unterstützt C++ die 4. Strategie durch spezielle Sprachelemente.

Die Idee ist die, daß eine Funktion in der ein Fehler auftritt, den sie selbst nicht lösen kann, eine Ausnahme auswirft, die von einem Ausnahme-Handler aufgefangen werden kann. Kann auch der Handler, der sich in dem die Funktion aufrufenden Kontext befindet, die Ausnahme nicht behandeln, so wird das Programm normalerweise abgebrochen.

12.2 Ausnahmebehandlung in C++

Das Auswerfen einer Ausnahme geschieht durch den Befehl `throw`, der als Parameter ein Objekt erhält, das ausgeworfen wird. Das Auffangen einer Ausnahme geschieht hinter einem sogenannten `try`-Block mit der `catch`-Anweisung. Der `try`-Block enthält die Anweisung, die eventuell das Auswerfen einer Ausnahme bewirkt. Direkt hinter dem `try`-Block muß die Ausnahmebehandlungsroutine stehen, die durch `catch` eingeleitet wird. Folgendes Beispiel soll dies verdeutlichen:

```
class array {
    int *start;
    int count;
public:
    // Der Konstruktor
    array(int i) { count=i; start=new int[i]; }
    // Die auszuwerfende Objektklasse
    class Range {
    public:
        int index;           // Genauere Info für
                           // Ausnahmebehandlung
        Range(int i): index(i){} // Konstruktor für Range
    };
    // Indexoperator für Arrayzugriff
    int & operator[] (int i) {
        if( i<0 || i>=count ) // Unerlaubtes Index?
            throw Range(i);   // ja⇒ Ausnahme
        else                  // sonst
            return start[i];  // Element liefern
    }
    // Liefern der Array-Größe
    int size() {return count;}
};

int main() {
    array a(10);              // 10-Element int-Array
    int a_value;
    try {
        a_value=a[a.size()+4711]; // Bereichsfehler
        a_value=a[a.size()-4711]; // Bereichsfehler
    }
}
```

```

    catch( array::Range r ) {
        // Ausnahme-Handler für array::Range-Ausnahmen
        // An diese Stelle gelangt man nur, wenn im
        // vorangehenden try-Block eine Anweisung steht,
        // die den Auswurf einer Range-Ausnahme bewirkt.
        // Überprüfen des Index
        if(r.index < 0)
            std::cerr << "index underflow" << r.index;
        else std::cerr << "index overflow" << r.index;
    } // catch
} // main

```

In diesem Beispiel wirft der `[]`-Operator der Klasse `array` immer dann eine Ausnahme aus, wenn der Index `i` außerhalb der für diese `array`-Instanz gültigen Grenzen liegt. `throw Range(i)` bewirkt, daß die Instanz, die durch den Konstruktor `Range()` erzeugt wird, an einen Ausnahme-Handler geschickt wird, der Ausnahmen dieses Typs bearbeitet. Das ausgeworfene Objekt kann dabei dazu genutzt werden, weitere Informationen an den Handler zu übertragen. Im Beispiel enthält die Klasse `Range` eine `int`-Variable in der der Index gespeichert wird, der die Ausnahme ausgelöst hat.

C++-Code, der auf Bereichsüberschreitungen hin überwacht werden soll, wird in einen `try`-Block gefaßt, wie die beiden Zeilen in `main()` aus dem letzten Beispiel. Tritt eine Bereichsüberschreitung auf (was im Beispiel provoziert wurde), so „fängt“ der Ausnahme-Handler, also der für das ausgeworfene Objekt zuständige `catch`-Block das ausgeworfene Objekt auf, und alle Anweisungen innerhalb dieses `catch`-Blocks werden durchlaufen. Der Typ der Ausnahme, für den ein `catch`-Block zuständig ist, kann als Argument an `catch` angegeben werden. Zusätzlich kann hier das ausgeworfene Objekt benannt werden, wodurch ein Zugriff auf die Objektdaten möglich ist. Soll ein `catch`-Block alle auftretenden Ausnahmen behandeln, kann `...` als Argument übergeben werden. Um Ausnahmen unterschiedlicher Art, d.h. unterschiedlichen Typs bearbeiten zu können, dürfen mehrere `catch`-Blöcke direkt hintereinander stehen, die jeweils für Ausnahmen eines bestimmten Typs zuständig sind.

Stellt der aufgerufene `catch`-Block fest, daß er selbst diese Ausnahme nicht endgültig bearbeiten kann, hat er die Möglichkeit mittels eines Aufrufs von `throw()` (ohne Parameter) die Ausnahme an den nächsten Ausnahmehandler weiterzuleiten. Existiert kein weiterer Handler für diese Ausnahme wird schließlich die Funktion `terminate()` aufgerufen, die das Programm abbricht. Tritt keine Ausnahme auf, so wird der `catch`-Block übersprungen. Der `catch`-Block muß immer unmittelbar hinter einem `try`-Block oder unmittelbar hinter einem weiteren `catch`-Block stehen.

Die Unterscheidung von mittels `throw` ausgeworfener Ausnahmen kann einerseits durch im ausgeworfenen Objekt enthaltene Informationen (wie `Range::index`) erfolgen oder aber dadurch, daß Objekte verschiedener Klassen mit `throw` ausgeworfen werden, die durch unterschiedliche Handler aufgefangen werden.

12.3 Deklaration von Ausnahmen

C++ bietet die Möglichkeit eine Funktion, die evtl. eine Ausnahme auswirft entsprechend zu deklarieren. Dies hat den Vorteil, daß im Quellcode an der Stelle der Deklaration der Funktion bereits ersichtlich ist, welche Ausnahmen von dieser Funktion oder einer Funktion, die aus ihr heraus aufgerufen wird, ausgeworfen werden können. Die Angabe der Ausnahmen, die eine Funktion auswerfen „darf“ lautet wie folgt:

```
class array{
    // ...
    class Range { /* ... */ };
    // ...
    int & operator[] (int i) throw( Range );
};
```

Diese Deklaration sagt aus, daß der `operator[]` nur die Ausnahme `Range` oder davon abgeleitete Ausnahme-Klassen auswerfen kann. Eine Deklaration der Form `void f() throw()` besagt, daß die so deklarierte Funktion `f()` keine Ausnahmen auswerfen darf. Mehrere Ausnahmen dürfen als durch Kommata getrennte Liste angegeben werden wie z.B. `int x() throw(a1, a2);`. Wird bei einer Deklaration einer Funktion eine Ausnahmeliste angegeben, muß diese auch bei der Definition der Funktion mit angegeben werden. Falls die Funktion eine Ausnahme auswirft, die nicht in der Liste der erwarteten Ausnahmen enthalten ist (z.B. wenn der oben deklarierte `operator[]` eine Ausnahme des Typs `RangeUnderflow` auswerfen würde), wird automatisch die Funktion `unexpected()` aufgerufen, die das Programm normalerweise beendet. Diese Funktion kann mit Hilfe von `set_unexpected` auf eine benutzerdefinierte Funktion abgebildet werden:

```
void myUnexpected() { /* ... */ }
set_unexpected(myUnexpected);
```

`unexpected()` darf nicht zurückkehren, kann aber weitere Ausnahmen auswerfen, die in der Liste der Ausnahmen der ursprünglichen Funktion enthalten sein müssen. Ist dies nicht der Fall, wird die Ausnahme durch `bad_exception` ersetzt. Ist auch diese Ausnahme nicht in der Liste der Ausnahmen der Funktion enthalten, wird `terminate()` aufgerufen. Ansonsten wird nach einem Handler für `bad_exception` gesucht und dieser aufgerufen.

Auf eine ähnliche Weise wie `unexpected()` kann die Funktion `terminate()` auf eine benutzerdefinierte Funktion abgebildet werden:

```
void myTerminate() { /* ... */ }
typedef void(*terminate_handler)();
terminate_handler old = set_terminate(myTerminate);
set_terminate(myTerminate);
```

Das Ergebnis der Funktion `set_terminate` ist ein Zeiger auf die bisherige Handler-Funktion.

12.4 Ausnahmen der Standardbibliothek

Die C++-Standardbibliothek erzeugt entsprechend der Definition des kommenden ISO-Standards bei verschiedenen Fehlern Ausnahmen, die von der Anwendung aufgefangen werden sollten. Dabei muß beachtet werden, daß die Elemente der Standardbibliothek in dem Namensraum `std` liegen. Alle Ausnahmen der Standardbibliothek sind von der Klasse `exception` abgeleitet.

Die wichtigsten Ausnahmen, die ausgeworfen werden können, sind die folgenden: `bad_alloc` wird ausgeworfen, wenn der Versuch Speicher mittels `new` zu allozieren fehlschlägt. Die zweite wichtige Ausnahme ist `bad_cast`, die beim Fehlschlagen einer Anwendung des `dynamic_cast`-Operators ausgeworfen wird. `bad_typeid` wird ausgeworfen, wenn der Parameter eines `typeid`-Aufrufs zur Feststellung des Typs eines Objekts ein Null-Zeiger ist. Die Ausnahme `range_error` wird ausgeworfen, wenn bei internen Berechnungen Bereichsfehler auftreten.

Das Auffangen dieser Fehler kann, neben der Ausnahmebehandlung in anderen Teilen einer Anwendung, in der Funktion `main` erfolgen. Hierdurch kann sichergestellt werden, daß alle auftretenden Ausnahmen aufgefangen werden. Der Programmtext hierzu kann wie folgt aussehen:

```
#include <iostream>

int main(){
    try{
        // Der gesamte Anwendungscode
        // ...
    }
    catch(std::range_error){
        std::cerr << "range_error\n"; exit(1);
    }
    catch(std::bad_alloc){
        std::cerr << "Kein Speicher verfügbar\n"; exit(1);
    }
    catch(std::bad_cast){
        std::cerr << "Illegaler dynamic_cast\n"; exit(1);
    }
    catch(std::bad_typeid){
        std::cerr << "Illegaler typeid-Aufruf\n"; exit(1);
    }
    // Alle anderen Ausnahmen auffangen
    catch(...){
        std::cerr << "Abbruch aufgrund von Ausnahme\n";
        exit(1);
    }
}
```