

```

public:
    Set() { card = 0; }
    // Test auf Mitgliedschaft
    friend bool operator& (int,Set);
    // Gleichheit
    friend bool operator== (const Set&,
                           const Set&);

    // Ungleichheit
    friend bool operator!= (const Set&,
                           const Set&);

    // Schnitt
    friend Set operator* (const Set&,
                         const Set&);

    // Vereinigung
    friend Set operator+ (const Set&,
                         const Set&);

    // Mengenminus
    friend Set operator- (const Set&,
                         const Set&);

    // Teilmenge
    friend bool operator<= (const Set&,
                           const Set&);

    // Ausgabe
    friend ostream& operator<< (ostream&,
                                const Set&);

    // Eingabe
    friend istream& operator>> (istream&,
                                Set&);

    // Hinzufügen
    bool AddElem(int);
    // Herausnehmen
    bool RmvElem(int);
};

```

Die Maximalzahl der Elemente `maxCard` sei dabei wahlweise eine Konstante oder ein Templateparameter. Ergänzen Sie die Schnittstelle der Klasse so, dass die Operatoren ausgewogen sind, und testen Sie die Klasse an geeigneten Beispielen. Halten Sie diese Schnittstelle für gelungen?

4.6 Ausnahmebehandlung (Exceptions)

In Abschnitt 6.2 (ab Seite 386) sehen wir uns an, welche Möglichkeiten es gibt, Fehler zu machen, und welche Möglichkeiten der Entwickler hat, diese zu erkennen und zu beseitigen. Grundsätzlich muss man aber zwischen zwei Arten von Fehlern unterscheiden:

1. Fehler, die nach einem gründlichen Test nicht mehr auftreten sollten
2. Fehler, die im laufenden Betrieb jederzeit auftreten können

Es ist damit eigentlich irreführend, in beiden Fällen von »Fehlern« zu sprechen. Im ersten haben wir es mit tatsächlichen Programmierfehlern zu tun, im zweiten dagegen nur mit Störungen des Programmablaufs. Um einen Mechanismus der Sprache C++ zur Behandlung dieser Störungen soll es in diesem Abschnitt gehen.

4.6.1 Behandlung von Fehlersituationen

Zunächst wollen wir uns überlegen, wie wir auf solche Störungen reagieren könnten. Ich hatte Ihnen schon mehrfach geraten, möglichst *defensiv zu programmieren*, das heißt hinter allen Funktionsaufrufen Fehlschläge zu vermuten und die entsprechenden Rückgabewerte abzufragen. Solche Fehlschläge können unter anderem sein:

Arten von Fehlschlägen

- numerische Probleme (Division durch 0, Wurzel oder Logarithmus aus negativer Zahl etc.)
- Dateiprobleme (Datei nicht gefunden, zu wenige Datensätze, Medium beim Schreiben voll etc.)
- Probleme mit Arrays (zu kleine Dimensionierung, Bereichsüberschreitung eines Index etc.)
- Speicherprobleme (zu wenig Speicher bei Reservierung)

Es geht also um Ausnahmesituationen, die Sie beim Programmieren bereits voraussehen und entsprechende Gegenmaßnahmen treffen können. Von welcher Art könnten diese sein?

Maßnahmen bei Problemfällen

- Die erste Möglichkeit ist, das Programm bei einem solchen Problem ganz zu beenden. Das ist bei ganz kurzen Testprogrammen sicher die einfachste Variante, bei allen etwas längeren Programmen hingegen völlig inakzeptabel.
- Eine andere Möglichkeit besteht darin, das Problem einfach zu ignorieren und mit Vorgabewerten fortzufahren, etwa bei einer versuchten Division durch 0 einfach ebenfalls 0 als Wert zurückzuliefern. Das ist eine Vorgehensweise, die das Programm zwar stabil macht, das Ergebnis aber oft falsch und nicht mehr nachvollziehbar.
- Die von uns bislang meist verwendete Strategie liegt darin, den Rückgabewert einer Funktion beziehungsweise Methode als Fehlercode anzusehen. Ist er beispielsweise 0 oder `true`, ging alles glatt; ansonsten drückt er ein Problem oder sogar schon eine mögliche Ursache aus.

*Behandlung erst auf
höherer Ebene*

Diese letzte Möglichkeit wird auch in der Standardbibliothek oft verwendet. Sie ist aber leider nicht für alle Situationen auch die beste. Denn zuweilen kann das Problem als solches zwar bei seinem Auftreten entdeckt werden; es ist jedoch nicht möglich, in diesem Kontext die eigentliche Ursache sowie eine geeignete Reaktion zu erkennen. Erst der Aufrufer kann darauf angemessen reagieren und den Fehler gegebenenfalls beheben. Nun ist leider der »Aufrufer« nicht immer gleich in der nächst höheren Ebene, sondern kann bei einer komplexen Programmstruktur mehrere Ebenen darüber liegen. Wenn es sich um ein Problem handelt, von dem der Benutzer unterrichtet werden sollte, will dieser ebenfalls nicht mit einer Meldung in der Form »Ungültiger Wert 0 in Zugriff auf `m_Order[i]`« belästigt, sondern über die wahren Hintergründe aufgeklärt werden.

Eine weitere Schwierigkeit mit dieser Vorgehensweise ist, dass es C++ erlaubt, den Rückgabewert von Funktionen und Methoden zu ignorieren. Die Funktion, die das Problem feststellt, kann also nach dem Zurückmelden weder sicherstellen noch nachprüfen, ob ihrer Meldung auch Beachtung geschenkt wurde.

4.6.2 Exception Handling

Eine zusätzliche Alternative zu gerade aufgezeigten Möglichkeiten besteht darin, dass die Funktion, die das Problem erkennt, selbst eine andere Funktion zu dessen Behandlung aufruft und dieser eventuell auch noch entsprechende Kontextinformationen mitgibt. Prinzipiell bedeutet dies nichts anderes, als einen zweiten Weg zum Verlassen einer Funktion neben der Rückkehr mit `return` zu schaffen.

Ablauf

Zur Behandlung von vorhersehbaren Fehlern, die in einer Funktion auftreten und die der Aufrufer behandeln kann, stellt C++ den Mechanismus der *Ausnahmebehandlung* (besser bekannt unter dem englischen Ausdruck *exception handling*) zur Verfügung. Dabei ist der Ablauf typischerweise der folgende:

1. Eine Funktion versucht, eine andere aufzurufen. Dieser Versuch wird durch das Schlüsselwort `try` ausgedrückt.
2. Gibt es während der Abarbeitung des Aufrufs einen Fehler, wirft sie eine Ausnahme aus. Dieses Werfen erfolgt mit dem Schlüsselwort `throw`.
3. Die aufrufende Funktion kennt die Möglichkeit, dass eine solche Ausnahme auftreten könnte, und ist bereit, diese abzufangen. Das geschieht mittels `catch`.

Sowohl `try` als auch `catch` leiten dabei einen Block ein und stehen in der *aufrufenden* Funktion. Der Befehl `throw` befindet sich in der *aufgerufenen* Funktion.

Wenn eine Funktion eine Exception auswirft, ist dies ein alternativer Rücksprung. Alle Objekte, die dort vollständig erzeugt wurden (also deren Konstruktoren schon beendet sind), werden gelöscht. Es findet aber *kein* normaler Rücksprung mehr statt!



4.6.3 Allgemeine Syntax

Syntaktisch entspricht diese Beschreibung folgendem Schema:

```
void f1()
{
    try
    {
        // Versuche, eine andere Funktion
        // aufzurufen; diese könnte eine
        // Exception auslösen und dabei ein
        // Fehlerobjekt übergeben
        f2();
    }

    // Werte die möglichen Fehlerobjekte aus
    catch(Fehlerklasse1& _f)
    {
        // Fehlerbehandlung
    }
    catch(Fehlerklasse2& _f)
    {
        // Fehlerbehandlung
    }
    // ... ggf. weitere catch-Blöcke

    // Hier geht der Programmfluss weiter
}
```



Abbildung 4.6
In der Fehlersituation
wird eine Ausnahme
geworfen.

Ausnahmeobjekte

Beim Werfen einer Ausnahme darf ein Objekt eines beliebigen Typs übergeben werden, also sowohl ein Standardtyp als auch ein Objekt einer Klasse. Sie sollten aber immer darauf achten, dass Sie das Objekt selbst übergeben und keinen Zeiger darauf.

Wenn Sie etwa schreiben:

```
if (bad())
    throw MyException();
```

müssen Sie vorher einen Typ — zum Beispiel eine Klasse — namens `MyException` definiert haben. Diese darf sogar völlig leer sein; denn eventuell drücken Sie ja schon durch den Typ selbst die Art der Ausnahme aus. Wenn Sie hier nur den Namen der Klasse sehen, so bedeutet dies, dass beim `throw`-Kommando ein temporäres Objekt von diesem Typ angelegt wird; nach Ende des zugehörigen `catch`-Blocks wird es automatisch wieder freigegeben. Sie verstehen sicher, dass es bei diesem Ablauf höchst ungeschickt wäre (wenn auch leider nicht syntaktisch falsch), das Ausnahmeobjekt dynamisch zu erzeugen.

Über den Konstruktor kann dieses Objekt bei Bedarf noch mit Informationen über die Umstände der Ausnahme gefüllt werden.

```
if (bad())
    throw MyException("Operation failed",
                      status);
```

Natürlich sollte die Klasse dann auch über entsprechende öffentliche Methoden verfügen, so dass der Fänger der Ausnahme auf diese Informationen zugreifen kann.

4.6.4 Auffangen der Ausnahmen

Das Argument von `catch` kann sowohl als Objekt als auch als Referenz abgefangen werden. Dadurch lassen sich auch die enthaltenen Information auswerten, zum Beispiel:

```
catch(MyException& _exc)
{
    cerr << _exc.getMessage() << endl;
    // weitere Fehlerbehandlung
}
```

Manchmal kann man nicht genau wissen, welche Ausnahmen auf einen Programmabschnitt zukommen können. Aber auch dafür gibt es einen Ausweg: Wenn Sie nur drei Punkte als Argument angeben, so fangen Sie damit alle (vorher nicht behandelten) Ausnahmen ab:



Abbildung 4.7
Geworfene Ausnahmen
sollten auch
aufgefangen werden.

```
catch(...)
{
    cerr << "Unbekannter Fehler!" << endl;
}
```

Sie sollten in Ihren Programmen darauf achten, dass Sie alle möglichen Ausnahmen auch auffangen. Denn es gilt: Wird eine Ausnahme in der aufrufenden Funktion nicht abgefangen, wird sie an deren aufrufende Funktion weitergegeben. Das setzt sich fort bis hin zu `main()`. Ist auch dort keine Behandlung implementiert, bricht das Programm mit einer Fehlermeldung ab. Dann haben Sie in punkto Fehlerbehandlung und Stabilität Ihres Programms herzlich wenig erreicht.

*Nicht gefangene
Ausnahmen*

Weiterwerfen

Es ist aber auch möglich, eine Ausnahme bewusst an die nächst höhere Ebene weiterzugeben. Vielleicht wollen Sie die Abfolge innerhalb des Aufruf-Stacks nur dokumentieren und den Fehler nicht wirklich behandeln. Dann werfen Sie die aktuelle Exception mit einem schlichten `throw` einfach weiter, zum Beispiel:

```
catch(...)
{
    log("Unbekannter Fehler!");
    throw; // Weitergabe
}
```

Angabe von Ausnahmen in Schnittstellen

Wenn Sie eine Klasse verwenden, die ein anderer implementiert hat, möchten Sie gerne wissen, welche Ausnahmen deren Methoden werfen könnten. Da die Header-Datei mit der Schnittstelle Ihre primäre Informationsquelle ist, sollten auch die Ausnahmen dort vermerkt sein.

Genau dies unterstützt die Sprache C++. Um anzugeben, welche Exception eine Funktion auswirft, kann man sie in einer Liste hinter dem Funktionsprototyp deklarieren.

```
void func(int _arg) throw(RangeException,
                          DomainException);
```

Ist eine solche Angabe vorhanden, ist sie auch verbindlich. In diesem Fall darf die Funktion `func` ausschließlich die Ausnahmen `RangeException` und `DomainException` werfen; bei anderen meldet der Compiler einen Fehler. Als Entwickler müssen Sie beachten, dass die Liste bei der Funktionsdefinition wiederholt werden muss.

Steht hinter dem Prototyp nur `throw()`, so verspricht die Funktion damit, überhaupt keine Exceptions auszuwerfen.

Aus Gründen der Abwärtskompatibilität gilt aber leider auch: Fehlt eine Angabe von `throw` hinter einer Funktion, so darf sie beliebige Ausnahmen auswerfen. Eine Schnittstelle mit gutem Design sollte daher immer die vor ihr geworfenen Ausnahmen auflisten.

Exception-Objekte als Hierarchie

Die Klassen der Exception-Objekte können auch eine Hierarchie bilden. Eine `catch`-Anweisung, die als Argument ein Objekt der Basisklasse enthält, fängt dann auch alle davon abgeleiteten Exceptions ab. Damit Sie auch tatsächlich auf die Informationen der abgeleiteten Klasse zugreifen können, müssen Sie die Methoden der Basisklasse natürlich als `virtual` deklarieren.

Wenn Sie beispielsweise einen Kellerspeicher (*stack*) mit fester Größe implementieren, also einen Container für Objekte, bei dem das letzte Objekt, das Sie dort abgelegt haben, auch das erste ist, das Sie wieder herausbekommen, können Sie folgende Hierarchie bilden:

```
class StackError {};
class StackEmpty : public StackError {};
class StackFull : public StackError {};
```

In der Klasse sieht das dann folgendermaßen aus:

```
template <typename T>
class SimpleStack
{
public:
// ...
    T pop() throw(StackEmpty);
    void push(const T&) throw(StackFull);
};
```

Wie üblich bezeichnet dabei `push()` das Ablegen und `pop()` das Herunternehmen eines Objekts. In der Anwendung können Sie nun durch das Fangen von `StackError` beide Fälle behandeln:

```
int main()
{
    SimpleStack<int> stack;
```

```
try
{
    cout << stack.pop();
    for (int i=0;i<30; i++)
        stack.push(i);
}
catch(StackError)
{
    cerr << "So geht's nicht!" << endl;
}

//...
}
```

Da `StackError` über keinerlei Elemente verfügt, genügt in der `catch`-Anweisung die Angabe des Typs. Einen Argumentnamen benötigen wir nicht, da wir mit dem Objekt sowieso nichts anfangen können.

Hintergrund

Man kann Exceptions auch (miss-)brauchen, um den Programmfluss zu steuern. In folgendem Beispiel wirft die Funktion `get_user_input()` dann eine Ausnahme, wenn die Eingabe ungültig war:

*Steuerung des
Programmflusses*

```
bool do_repeat = true;
do
{
    try
    {
        get_user_input();
        do_repeat = false;
    }
    catch(InvalidInput)
    {
        cerr << "Input invalid!";
    }
}
while(do_repeat);
```

Eine solche Vorgehensweise ist aber nur in den seltensten Fällen eine gute Idee. Normalerweise sollten Sie andere Wege zur Kontrollflusssteuerung finden können und dafür keine Exceptions benötigen. Bewusst ausgelöste Exceptions sind ohnehin immer überflüssig.

4.6.5 Beispiel: Vektor

Die Technik der Ausnahmebehandlung wollen wir nun bei unserer bekannten Vektorklasse einsetzen, die wir zuletzt in Abschnitt 4.4 überarbeitet haben. Wir definieren zwei Ausnahmen:

- ❑ `VektorNoMem`, wenn kein Speicher mehr für das dynamische Anlegen des Feldes vorhanden ist
- ❑ `VektorOutOfRange`, wenn bei `at()` der Index den zulässigen Bereich überschreitet

Die Exception-Klassen

Beide leiten wir wie eben von einer Basisklasse ab, die auch gleich die Schnittstelle für den Zugriff auf die Informationen in abstrakter Weise definiert. Wir geben die Fehlermeldung einfach als Text zurück, den wir mittels eines `stringstream` erzeugt haben:

```
// Ausnahmebasisklasse
class VektorException
{
public:
    VektorException() throw() {}
    virtual ~VektorException();
    virtual const char* what() const = 0;
};

// Ausnahme bei Bereichüberschreitung
class VektorOutOfRange
{
private:
    unsigned int size, index;

public:
    VektorOutOfRange(unsigned int _size,
                    unsigned int _index) :
        size(_size), index(_index) {}
    virtual ~VektorOutOfRange() {};
    virtual const char* what() const
    {
        stringstream s;
        s << "Bereichsüberschreitung (Größe: "
          << size << ", Index: " << index
          << ")" << ends;
        return s.str();
    }
};
```

Die Klasse `VektorNoMem` ist ganz analog zu `VektorOutOfRange` aufgebaut, so dass ich sie hier wohl nicht abgedruckt muss.

Wenn wir nun alle möglichen Ausnahmen in die Schnittstelle der Klasse *Vektor* aufnehmen, erlangt diese folgende Form: *Die Schnittstelle*

```
// Deklaration der Klasse
template <typename T> class Vektor
{
private:
    unsigned int size;
    T* v;

public:
    Vektor() : size(0), v(0) {}
    Vektor(unsigned int _size) throw(VektorNoMem);
    Vektor(const Vektor& _vek) throw(VektorNoMem);
    ~Vektor() { if (v) delete[] v; }
    void resize(unsigned int _size)
        throw(VektorNoMem);
    const T& at(unsigned int _i) const
        throw(VektorOutOfRange);
    T& at(unsigned int _i)
        throw(VektorOutOfRange);
    // ...
};
```

Der Konstruktor mit Größenangabe sieht eigentlich aus wie bisher; nur der Fall, dass kein Speicher mehr zur Verfügung steht, wird mit einer Exception quittiert. *Konstruktor mit Größenangabe*

```
template <typename T>
Vektor<T>::Vektor(unsigned int _size)
    throw(VektorNoMem) : size(_size)
{
    v = new T[size];
    if (v == 0)
        throw VektorNoMem(size);
}
```

Beim Zugriff auf einzelne Elemente mittels der `at()`-Methode haben wir das `assert()` durch ein `throw` ausgetauscht. Das hat vor allem den Vorteil, dass nicht bei jeder Index-Verletzung das Programm gleich stehen bleibt, sondern man die Situation unter Umständen noch retten kann. *Zugriff auf einzelne Elemente*

```
template <typename T>
T& Vektor<T>::at(unsigned int _i)
    throw(VektorOutOfRange)
{
    // Vorbedingung: _i gültig und v vorhanden
    if (v == 0 || _i >= size)
```

```

        throw VektorOutOfRange(size, _i);

    return v[_i];
}

```

Durch die Übergabe der entscheidenden Größen `size` und `_i` ist der Aufrufer in der Lage, seine Anfrage gegebenenfalls zu überdenken und zu wiederholen.

Testprogramm

Wenn wir in unserem Testprogramm beispielsweise den Index um eins daneben platzieren, erhalten wir die Exception:

```

int main()
{
    unsigned int i=0;

    // Ganzzahlvektor
    Vektor<int> v(5);

    for(i=0; i<5; i++)
        v.at(i) = i+3;

    try
    {
        // einen Index zu viel
        for(i=0; i<=5; i++)
            cout << v.at(i) << " ";
    }
    catch (VektorOutOfRange& _e)
    {
        cerr << "Fehler: " << _e.what()
            << endl;
    }
}

```

Das Programm führt zur Ausgabe:

```

3 4 5 6 7 Fehler: Bereichsüberschreitung
(Größe: 5, Index: 5)

```

4.6.6 Exceptions und die Standardbibliothek

Als moderner Bestandteil von C++ sollte natürlich auch die Standardbibliothek — insbesondere die STL — Gebrauch von Exceptions machen. Allerdings ist die gegenwärtige Implementierung der GNU-STL noch nicht so weit fortgeschritten, dass sie überall mit Ausnahmen arbeitet. Nur ein kleiner Teil, das `Bitset`, unterstützt sie; doch dazu später mehr.

Die Basisklasse für die Exceptions wird jedoch immer mitgeliefert. Sie können daher auch Ihre eigenen Ausnahmen davon ableiten:

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

Wie Sie sehen, habe ich mich bei den Ausnahmen der Vektorklasse bereits daran orientiert. Diese Klasse ist in der Header-Datei `<exception>` zu finden. Es gibt davon noch eine Reihe von abgeleiteten Klassen, die verschiedene Arten von Fehlern ausdrücken sollen. Diese sind in `<stdexcept>` definiert:

- ❑ `logic_error` bezeichnet Fehler in der Programmlogik, die prinzipiell vermeidbar sind. Davon wiederum abgeleitet sind die Klassen `invalid_argument`, `length_error`, `out_of_range` sowie `domain_error`.
- ❑ `runtime_error` ist für Probleme, die zur Laufzeit von außerhalb des Programms herrühren. Hierunter gibt es `range_error` und `overflow_error`.
- ❑ Darüber hinaus gibt es noch drei Klassen, die direkt von `exception` erben. Diese stellen besondere Probleme im Programmablauf dar:
 - ❑ `bad_alloc` wird bei Speichermangel vom Operator `new` geworfen. Voraussetzung ist, dass Sie die Header-Datei `<new>` einbinden.
 - ❑ `bad_typeid` bezieht sich auf einen falschen Objekttyp bei der Bestimmung von Typinformationen zur Laufzeit. Darauf kann ich leider hier nicht näher eingehen.
 - ❑ `bad_cast` wird vom Operator `dynamic_cast<>` geworfen, falls Sie als Argument eine Referenz angeben, die nicht auf ein Objekt vom angegebenen Typ (oder eine Unterklasse davon) verweist (siehe Seite 296).

Ähnlich wie bei meiner Klasse `VektorException` erhalten Sie auch hier über die Methode `what()` eine textuelle Beschreibung des Fehlers.

4.6.7 Tipps und Hinweise

Exceptions haben zweifellos einige Vorteile. Neben der Eleganz sind dies vor allem:

Vorteile von Exceptions

- ❑ Exception Handling ist im Programmcode eindeutig erkennbar.
- ❑ Regulärer Code wird durch Exception Handling kaum belastet.

- ❑ *Jede* Exception muss behandelt werden: Es gibt keinen Ausweg!
- ❑ Durch Parametrisierung mit Typen und Objekten wird eine fein dosierte Reaktion auf unterschiedliche Fehlerarten möglich.
- ❑ Exception Handling führt als einheitliches Sprachmittel zu portablen Lösungen.

Sparsam einsetzen!

Allerdings ist dieses Konzept auch kein Allheilmittel. Verschiedene Autoren (zum Beispiel [COPLIEN 1992] oder [KERNIGHAN und PIKE 2000]) warnen zurecht davor, zu häufigen Gebrauch von Exceptions zu machen. Wenn sich etwa eine Datei nicht öffnen lässt, weil der Name falsch war, ist das keine so dramatische Situation, dass sie eine Exception rechtfertigen würde. Gehen Sie also sparsam damit um und versuchen Sie, weitgehend mit Rückgabewerten auszukommen. Nur bei komplex verschachtelten Strukturen und bei Fehlern in Konstruktoren (aber nicht in Kopierkonstruktoren!), die ja keine Rückgabewerte erlauben, sind Exceptions sinnvoll.

Noch ein Hinweis am Schluss: Werfen Sie niemals Exceptions in einem Destruktor! Sie bringen damit Ihr Programm vermutlich vollends durcheinander.

4.6.8 Zusammenfassung

Aus diesem Abschnitt sollten Sie sich folgende Gedanken einprägen:

- ❑ In fast jedem Programm können Ausnahmesituationen auftreten, die Sie aber schon voraussehen können und für die Sie entsprechende Gegenmaßnahmen treffen können. Die übliche Vorgehensweise, mit dem Rückgabewert einer Funktion ihren Fehlerstatus zu liefern, ist bei Problemen, die wegen fehlender Kontextinformationen erst auf höherer Ebene behandelt werden können, unzureichend.
- ❑ Ausnahmen stellen eine alternative Möglichkeit des Rücksprungs aus einer Funktion dar. Wird eine Ausnahme geworfen, findet kein üblicher Rücksprung mehr statt. Es werden dabei alle existierenden lokalen Objekte gelöscht.
- ❑ Eine Ausnahme wird mit dem Befehl `throw` geworfen. Als Argument darf dabei ein beliebiger Typ mitgegeben werden. Oftmals ist es ein selbst definiertes Objekt, das über seinen Konstruktor auch noch einige Kontextinformationen aufnehmen kann.
- ❑ Der Aufrufer einer Funktion, die eine Ausnahme werfen könnte, sollte seinen Aufruf in einen Block einschließen, vor dem ein `try` steht. Nach dem Block stehen einige oder mehrere weitere Blöcke, die mit

der `catch`-Anweisung beginnen. Für jede Klasse, von der ein Objekt mit der Ausnahme übergeben werden kann, darf eine `catch`-Anweisung stehen. Mit `catch(...)` werden alle Ausnahmen abgefangen.

- ❑ Mit einem weiteren `throw`-Kommando ohne weitere Argumente gibt man eine bereits gefangene `Exception` an die nächst höhere Ebene weiter.
- ❑ Um anzuzeigen, welche Ausnahmen eine Funktion werfen kann, können Sie diese hinter dem Funktionskopf auflisten. Die Liste beginnt dabei mit `throw(` und ist durch Kommas getrennt und wird mit einer runden Klammer geschlossen. Die Angabe ist verbindlich, das heißt, dass die Funktion auch nur diese Ausnahme werfen darf. Geben Sie ein `throw()` mit leerem Klammernpaar an, darf die Funktion überhaupt keine Ausnahme werfen.

4.6.9 Übungsaufgaben

1. Beantworten Sie folgende Fragen:
 - ❑ Wann sind Fehlermeldungen über Rückgabewerte nicht sinnvoll?
 - ❑ Was, glauben Sie, ist der Grund, dass man für Ausnahmen die Metapher des Werfens und Fangens gewählt hat?
 - ❑ Was ist der typische Aufbau eines Programmteils, der von den Funktionen, die er aufruft, erwartet, dass sie eine Ausnahme werfen?
 - ❑ Wie wirft man eine bereits gefangene Ausnahme weiter?
 - ❑ Ist es korrekt, die Liste der möglichen auftretenden Ausnahmen nur im Prototyp einer Funktion anzugeben und im Funktionskopf bei der Implementierung wegzulassen?
 - ❑ Warum ist es nicht sinnvoll, in einem Kopierkonstruktor oder einem Destruktor eine Ausnahme zu werfen?
2. Schreiben Sie eine Klasse mit Namen `Char`, die bei einem bevorstehenden Unter- beziehungsweise Überlauf eine Ausnahme auswirft, sich aber ansonsten wie der Datentyp `char` verhält.
3. Vervollständigen Sie die Klasse `SimpleStack` von Seite 330 und testen Sie sie an geeigneten Beispielen.