Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 6

6. Ausnahmebehandlung (Exception Handling)

6.1. Allgemeines

6.2. Werfen und Fangen von Exceptions

6.3. Beispiele

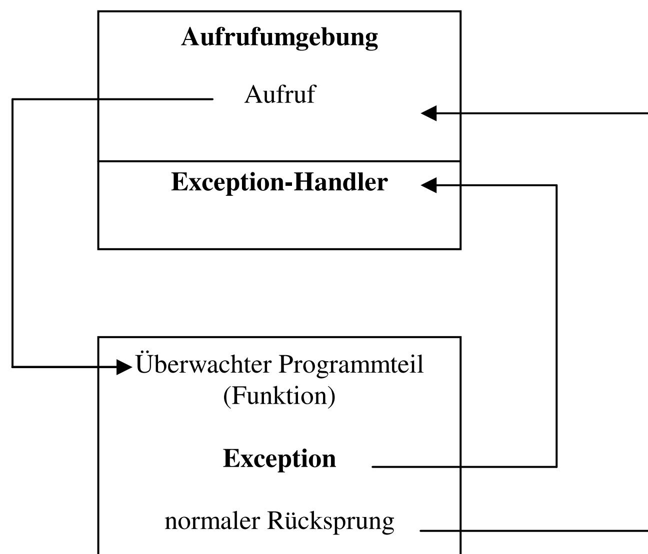
Ausnahmebehandlung in C++ - Allgemeines

- **Ausnahmesituationen**

C++ stellt einen speziellen Mechanismus zur Behandlung von Ausnahmesituationen (Fehlerfällen, Exceptions) zur Verfügung. ⇒ **Exception Handling**.
Ausnahmesituationen in diesem Sinne sind **Fehler** oder sonstige unerwünschte **Sonderfälle** (z.B. Dateizugriffsfehler, Fehler bei der dynamischen Speicherallokation, Bereichsfehler usw.), die im **normalen Programmablauf** nicht auftreten sollten aber auftreten können.
Sie werden vom Programmierer festgelegt.
→ Es handelt sich **nicht** um einen Mechanismus zur Behandlung von externen oder internen **Interrupts**.

- **Grundprinzip :**

Das in C++ implementierte Exception-Handling beruht auf der **Trennung** von der im normalen Programmablauf auftretenden **Fehlererkennung** und der **Fehlerbehandlung** :
Der Programmteil (i. a. eine Funktion), der einen Problemfall (Fehlerfall) entdeckt, bearbeitet diesen nicht selbst, sondern "**wirft**" **eine Exception** ("**throws an exception"**).
In der Aufrufumgebung dieses Programmteils sollte eine Bearbeitungsroutine für diese Exception (→ **Exception-Handler**) vorhanden sein. Der Exception Handler "**fängt**" die Exception und behandelt den Problemfall.



- **Exception-Klassen**

Eine derartige Exception wird durch einen Ausdruck beschrieben, der prinzipiell einen beliebigen Wert ergeben kann.
Im einfachsten Fall kann dies der Wert eines Standard-Datentyps oder ein char-Pointer (C-String) sein, meist wird es sich dabei aber um ein Klassen-Objekt handeln.
Durch den Typ des erzeugten Werts bzw Objekts lassen sich verschiedene **Exception-Arten** unterscheiden.
Zweckmäßigerweise definiert man für die verschiedenen Exceptions spezielle **Exception-Klassen** (Ausnahme-Klassen, Fehler-Klassen, → **Exception-Typ**).
Beim Auftritt einer Exception wird dann ein Objekt der entsprechenden Klasse erzeugt und geworfen.
⇒ die **Exception** wird als **Objekt** behandelt.

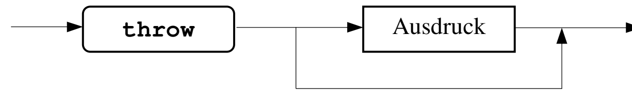
Wenn eine Exception-Klasse allein zur Anzeige der Exception-Art dienen soll, muß sie keine Komponenten besitzen.
I.a. wird sie jedoch Komponenten haben, die der Fehlerbehandlung genauere Informationen über die Fehlerursache zur Verfügung stellen.

Ausnahmebehandlung in C++ - "Werfen" einer Exception

• `throw`–Ausdruck

◇ Das **"Werfen" einer Exception** erfolgt durch einen `throw`–Ausdruck. Dieser wird mit dem **unären `throw`–Operator** gebildet, dessen Präzedenz zwischen Komma-Operator und den Zuweisungsoperatoren liegt.

◇ Der **Typ** eines `throw`–Ausdrucks ist **`void`** → ein `throw`–Ausdruck ist nur in der Form einer Ausdrucksanweisung möglich.



◇ Der throw-Ausdruck initialisiert ein **temporäres Objekt** seines Operanden-Typs (=**Exception-Typ**) mit dem Wert seines Operanden (→ "geworfene" Exception), **sucht** den passenden **Exception-Handler**, initialisiert gegebenenfalls dessen Parameter mit diesem Wert und übergibt die Programmfortsetzung an diesen.
→ Der Handler hat die Exception "**gefangen**".
Dabei findet eine **Stackbereinigung** ("*stack unwinding*") statt: Sämtliche `auto`–Objekte (und einfache `auto`–Variable), die seit Eintritt in die zum Exception-Handler gehörende Aufrufumgebung angelegt worden sind, werden entfernt.

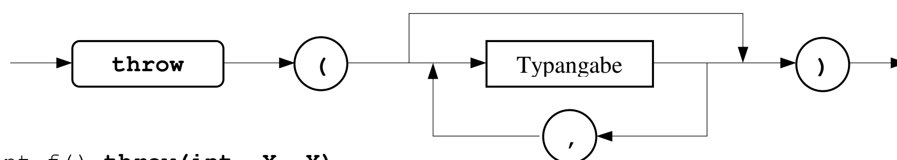
◇ Das vom `throw`–Ausdruck angelegte temporäre Objekt existiert solange, solange ein Exception-Handler für diese Exception ausgeführt wird. Erst nach Beendigung des (letzten) Exception-Handlers für diese Exception wird das Objekt zerstört.

◇ Die Form des `throw`–Ausdrucks **ohne Operanden** ist nur **innerhalb eines Exception-Handlers** (bzw innerhalb einer von einem Exception-Handler aufgerufenen Funktion) zulässig .
Ein derartiger `throw`–Ausdruck bewirkt, daß eine weitere Exception mit dem vorhandenen temporären Objekt geworfen wird, d.h. es wird versucht, die Programmfortsetzung an einen weiteren Exception-Handler zu übergeben (→ "***rethrow***" *the exception*).

◇ Kann durch den `throw`–Ausdruck kein passender Exception-Handler gefunden werden, so wird die (Standardbibliotheks-)Funktion **`terminate()`** aufgerufen.

• Exception-Specification

◇ Ergänzung einer **Funktionsdeklaration** bzw des Funktionskopfes einer **Funktionsdefinition** um eine **Auflistung der Exception-Typen**, die von der **Funktion** – direkt oder indirekt – **geworfen** werden können bzw dürfen.
Eine Exception-Specification kann auch in einer **Funktionspointer-Vereinbarung** enthalten sein.



◇ **Beispiele** :

```
int f() throw(int, X, Y)
{
..// Funktion darf Exceptions vom Typ int, X und Y werfen
}

void (*pf)() throw(A);
```

◇ Enthält eine Vereinbarung (Deklaration oder Definition) einer Funktion eine Exception-Specification, so müssen alle anderen Vereinbarungen derselben Funktion ebenfalls eine Exception-Specification mit derselben Typ-Liste enthalten.

◇ Eine Exception-Specification mit **leerer Typ-Liste** bedeutet, daß die entsprechende Funktion **keine Exceptions** werfen darf. Eine **ohne Exception-Specification** vereinbarte Funktion darf **beliebige Exceptions** werfen.

◇ Wenn eine Typ-Liste den Typ `X` enthält, so darf die betreffende Funktion neben Exceptions dieses Typs auch Exceptions aller Typen, die sich von `X` öffentlich und eindeutig ableiten lassen, werfen

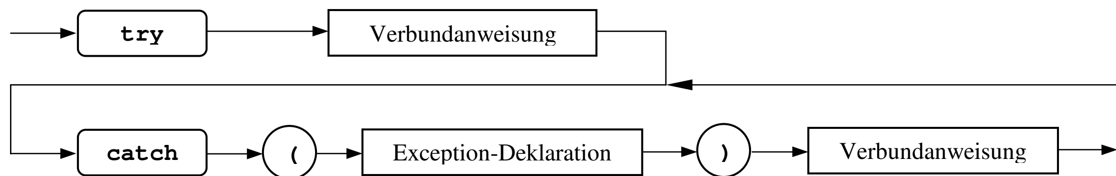
◇ Wird während der Abarbeitung einer Funktion eine Exception geworfen, deren Typ nicht in der Typ-Liste enthalten ist, so wird die Standardbibliotheks-Funktion **`unexpected()`** aufgerufen.

◇ **Anmerkung** : Die Exception-Specification hat bei **Visual-C++ keine Wirkung**.

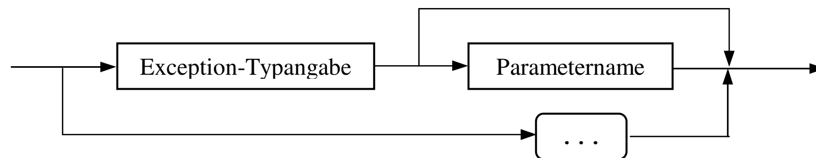
Ausnahmebehandlung in C++ - "Fangen" von Exceptions (1)

- **`try`-Anweisung**

  ◇ Sie legt die Aufrufumgebung und damit den Gültigkeitsbereich einer Ausnahmebehandlung und die dazugehörigen Exception-Handler fest.
  ⇒ Sie besteht aus
  - einer **Verbundanweisung** (**`try`-Block**), die aus den Anweisungen besteht, in denen die Fehlererkennung wirksam ist (Häufig sind Aufrufe von Funktionen enthalten, in denen die eigentliche Fehlererkennung stattfindet).
  - den **Exception-Handlern** (**`catch`-Blöcken**), die den verschiedenen jeweils fangbaren Exception-Typen zugeordnet sind. Für jeden Exception-Typ ist ein eigener Handler vorzusehen.



Exception-Deklaration :



  ◇ **`try`-Anweisungen** können **geschachtelt** werden.

  ◇ Der in der Exception-Deklaration eines Handlers angegebene Typ wird als der **Typ des Handlers** bezeichnet.

- **"Fangen" von Exceptions**

  ◇ Der `throw`-Ausdruck bewirkt eine **Suche nach einem Handler**, der die geworfene Exception "fangen", d. h. bearbeiten kann. Die Suche erfolgt in der Reihenfolge der `catch`-Blöcke.
  Dem **ersten Handler**, der die Exception bearbeiten kann, wird das geworfene **Fehlerobjekt übergeben**.
  Die Übergabe erfolgt **analog zur Parameterübergabe** bei Funktionen, allerdings finden **keine impliziten Typkonvertierungen** für **Standard-Datentypen** statt.

  ◇ Ein Handler kann eine **Exception vom Typ `E`** bearbeiten,
  - wenn der **Typ des Handlers** der Typ **`T`**, oder **`T&`** (einschließlich einer eventuellen Qualifikation mit **`const`** und/oder **`volatile`**) ist und
    - **`E`** und **`T`** der **gleiche Typ** sind (ohne Berücksichtigung der Qualifikation mit `const` bzw `volatile`) oder
    - **`T`** zugreifbare und eindeutige **Basisklasse von `E`** ist
  - oder wenn der **Typ des Handlers** ein **Pointer-Typ** (einschließlich einer eventuellen Qualifikation mit **`const`** und/oder **`volatile`**) ist und **`E`** ebenfalls ein **Pointer-Typ** ist, der mittels Standardkonversion in den Typ des Handlers umgewandelt werden kann (Pointer auf abgeleitete Klasse → Pointer auf Basisklasse).

  ◇ Ein Handler, in dessen Exception-Deklaration statt eines Typs drei Punkte (**`...`**) angegeben ist, kann **jede beliebige Exception** fangen. Falls vorhanden, muß er der **letzte Handler** einer try-Anweisung sein.

  ◇ Wird kein passender Handler gefunden, wird die Suche in der nächsten umfassenden `try`-Anweisung – sofern vorhanden – fortgesetzt usw.
  Wird auch auf diese Weise **kein passender Handler** gefunden, wird die (Standardbibliotheks-) Funktion **`terminate()`** aufgerufen.

  ◇ **Nach** erfolgter **Abarbeitung eines Handlers** wird – sofern der Handler nicht das Programm beendet oder erneut eine Exception geworfen hat – das **Programm** mit der Anweisung, die **auf die `try`-Anweisung folgt**, zu der der Handler gehört, **fortgesetzt**. Es findet also **keine Rückkehr** zu der **Stelle**, an der die **Exception geworfen** wurde, statt.

Ausnahmebehandlung in C++ - "Fangen" von Exceptions (2)

- **Funktions-`try`-Blöcke**

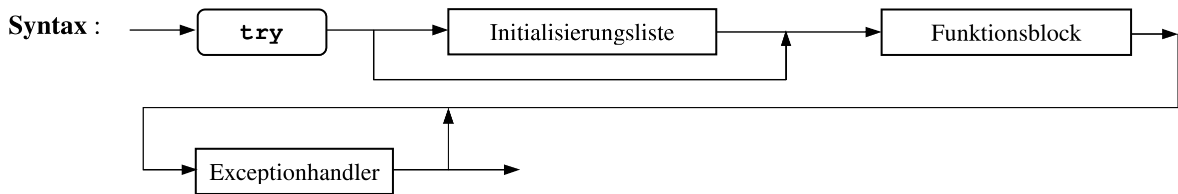
◇ Normalerweise treten **`try`**-Blöcke **innerhalb** von Funktionsblöcken auf. Bei Konstruktoren bedeutet dies, dass die Initialisierung von Datenkomponenten und/oder Basisklassen-Teilobjekten (über die **Initialisierungsliste** bzw. **implizite Defaultkonstruktoraufrufe**) nicht überwacht werden kann.

```
class IntVector
{
        int m_iSize;
        int *m_pVec;
 public:
        IntVector(int val=0) : m_iSize(val), m_pVec(0)
        {
           if (m_iSize<1)
              throw m_iSize;
           m_pVec=new int[m_iSize];
        }
        virtual ~IntVector(void)          { delete [] m_pVec; }
};

class NewIntVector : public IntVector
{
 public:
        NewIntVector(int val) : IntVector(val)  /* wird so nicht ueberwacht */
        {
           try
           {
               // weitere notwendige Initialisierungen
           }
           catch(int ex)
           {  cerr << "Exception in NewIntVector constructor. "
                       "Alloc size: " << ex << endl;
           }
        }
};
```

◇ Ein **Funktions-`try`-Block** ersetzt in einer Funktions-Definition den eine Verbundanweisung darstellenden Funktions-block (= Funktionsrumpf). Er fasst eine – optionale – **Initialisierungsliste** und den eigentlichen **Funktionsblock** zusammen.

Syntax :



Damit wird bei **Konstruktoren** die Überwachung des Codes auf die Initialisierung aller Datenkomponenten und Basisklassenobjekte (welche explizit über Angabe einer **Initialisierungsliste** oder implizit über **Defaultkonstruktoraufrufe** durchgeführt wird) erweitert.

```
class NewIntVector1 : public IntVector
{
 public:
        NewIntVector1(int val)
          try : IntVector(val) /* Code der Initialisierungsliste wird ueberwacht */
          {
              // weitere notwendige Initialisierungen
          }
          catch(int ex)
          {   cerr << "Exception in NewIntVector1 constructor. "
                      "Alloc size: " << ex << endl;
          }
};
```

Ausnahmebehandlung in C++ - "Fangen" von Exceptions (3)

- **Anmerkungen zu Funktions-`try`–Blöcken**

◇ Funktions-`try`-Blöcke können nicht nur auf Konstruktoren, sondern auf **alle Klassenmethoden und freie Funktionen** angewendet werden.

Beispiel:

```
int main()
try
{
       NewIntVector1 a(-3);
       return 0;
}
catch(...)
{      cerr << "Exception!" << endl;
       return 1;
}
```

◇ Die Sichtbarkeit und die Lebensdauer von **Funktionsparametern** werden bei Funktions-`try`-Blöcken auf alle Exceptionhandler (`catch`-Blöcke) erweitert.

◇ In Funktionen, die einen Funktions-`try`-Block nutzen, muss **jeder** dazugehörende `catch`-Block einen dem Funktionstyp entsprechenden Funktionswert zurückgeben.
`return`-Anweisungen in `catch`-Blöcken eines **Konstruktors** sind jedoch **nicht erlaubt**.

◇ Am Ende eines Exceptionhandlers von Funktions-`try`–Blöcken für **Konstruktoren oder Destruktoren** wird die Exception **automatisch erneut geworfen** ("rethrow").
D. h. eine Exception, die in einem Konstruktor oder Destruktor über einen Funktions-`try`-Block gefangen wird, kann dort nicht gelöst werden.

◇ Wird in einem Konstruktor eine Exception geworfen, so werden die bereits vollständig konstruierten Teilobjekte **vor** dem Eintritt in den entsprechenden `catch`-Block zerstört.
Damit führt der Zugriff auf **nicht-statische Basisklassenobjekte** oder **nicht-statische Datenkomponenten** innerhalb eines Exceptionhandlers eines Funktions-`try`–Block zu undefiniertem Verhalten im Programmablauf.

◇ Werden in Destruktoren von Objekten mit statischer Lebensdauer oder in Konstruktoren von Objekten innerhalb eines `namespace`-Bereiches Exceptions geworfen, können diese **nicht** im Funktions-`try`–Block der Funktion `main()` gefangen werden.

◇ **Funktions-`try`-Blöcke** werden von **einigen C++-Compilern** (u. a. Visual Studio 6.0) **nicht unterstützt!**

Einfaches Demonstrationsprogramm zur Ausnahmebehandlung in C++

- **Demonstrationsprogramm `exhdemo`** :

```
//----------------------------------------------------------------------------
// Programm EXHDEMO
// ---------------------------------------------------------------------------
// Einfaches Demonstrationsprogramm zum Exception Handling
// ---------------------------------------------------------------------------

#include <iostream>
using namespace std;

class IntVec
{ public:
    IntVec(int);
    ~IntVec() {  delete[] ip;  }
    int& operator[](int);
    //  ...
  private:
    int *ip;
    int len;
};

class RangeError                                 // Fehlerklasse
{ public:
    RangeError(int i) : ind(i) { }
    operator int() { return ind; }
  private:
    int ind;                                     // fehlerhafter Index
};

IntVec::IntVec(int i)
{  ip=new int[i];  len=i;  for(i=0; i<len; i++)  ip[i]=i;  }

int& IntVec::operator[](int i)
{ if (i<0 || i>=len)
    throw RangeError(i);                         // "Werfen" der Exception
  else
    return ip[i];
};

int main(void)
{ IntVec iv(20);
  try                                            // try-Block
  { cout << "\niv[10] : " << iv[10] << '\n';
    cout <<   "iv[30] : " << iv[30] << '\n';
    cout << "Ende try-Block\n";
  }
  catch (RangeError& err)                        // catch-Block
  { cout << "Index [" << int(err) << "] out of range\n";
  }
  cout << "Programmende\n";
  return 0;
}
```

- **Ausgabe des Programms** :

```
iv[10] : 10
Index [30] out of range
Programmende
```

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (1)

- **Erzeugung und Behandlung von Exceptions bei Zugriff zu einem Stack**

◊ Definition der folgenden **Exception-Klassen** :

- **`MyException`** als Basisklasse für die weiteren Exception-Klassen.
  Dem `protected` Konstruktor ist ein konstanter String, der die Exception-Ursache beschreiben sollte, zu übergeben. Dieser String wird nicht in einen neu allokierten Speicherbereich kopiert, sondern es wird nur seine Adresse in einer Datenkomponente gespeichert.
  Um ein Kopieren von `MyException`–Objekten zu vermeiden, sind Copy-Konstruktor und Zuweisungsoperator als `private` deklariert.
  Mittels der `public` Memberfunktion `getReason()` kann die Adresse des Ursachen-Strings ermittelt werden.
- **`StackEmptyException`**
- **`StackFullException`** (Klassen-Template)
  Template-Parameter ist der jeweilige Typ der Stack-Elemente
  Dem Konstruktor ist ein Wert vom Typ der Stack-Elemente als Parameter zu übergeben. Es sollte sich um den Wert, bei dessen Ablage-Versuch die Exception aufgetreten ist, handeln. Der übergebende Parameter wird in einer Datenkomponente gespeichert
  . Mittels der `public` Memberfunktion `getValue()` kann zu diesem Wert lesend zugegriffen werden.
- **`HeapFullException`**

◊ Realisierung eines **"allgemeinen" Stacks** durch ein Klassen-Template **`Stack<T>`**
Template-Parameter ist der Typ der jeweiligen Stack-Elemente.
Dem Konstruktor ist die (Anfangs-)Kapazität (Größe) des Stacks als Parameter zu übergeben.
Mittels der `public` Memberfunktion `incCapacity()` läßt sich diese Kapazität um einen bestimmten Bruchteil der aktuellen Kapazität erhöhen. Der Erhöhungs-Bruchteil ist als Parameter zu übergeben
Sowohl der **Konstruktor** als auch **`incCapacity()`** werfen die Exception **`HeapFullException`**, wenn der benötigte Speicher nicht allokiert werden kann.
Die `public` Memberfunktion **`push()`** wirft die Exception **`StackFullException`**, wenn der Stack voll ist, die `public` Memberfunktion **`pop()`** wirft die Exception **`StackEmptyException`**, wenn sich kein Element im Stack befindet
Weiterhin ist die `public` Memberfunktion **`outStack()`** zur Ausgabe des Stackinhalts nach `cout` definiert.

◊ Demonstration des Exception Handling mittels eines kleinen **Testprogramms** :
Die Funktion `main()` ruft eine Funktion **`testStack()`** auf.
In `testStack()` wird ein **`char`–Stack** untersucht.
In einer `do`–Schleife wird eine **`try`–Anweisung** ausgeführt, in der nach Ausgabe des jeweils aktuellen Stack-Inhalts die Eingabe der jeweils nächsten auszuführenden Operation angefordert und diese dann ausgeführt wird
Bei Eingabe von `'q'` wird die `do`–Schleife beendet.
Die `try`–Anweisung definiert **zwei Exception-Handler** : einen, der **`StackFullException`**s fängt, einen zweiten, der **`StackEmptyException`**s fängt.
Der Handler für `StackEmptyException`s gibt lediglich die Exception-Ursache aus, anschließend wird die `do`–Schleife fortgesetzt.
Der Handler für `StackFullException`s versucht durch Aufruf von `incCapacity()` den Stack zu vergrößern, um dann das Element, das zunächst keinen Platz hatte, im vergrößerten Stack abzulegen. Bei Erfolg wird die `do`–Schleife fortgesetzt.
Kann der Stack nicht vergrößert werden, so wird durch `incCapacity()` die Exception **`HeapFullException`** geworfen, die von einem Exception-Handler in einer übergeordneten `try`–anweisung gefangen werden muß.
Diese **`try`–Anweisung** ist in **`main()`** enthalten . Sie umschließt den Aufruf von `testStack()`.
Der durch sie definierte **Exception-Handler** fängt Exceptions vom Typ **`MyException`**, damit auch Exceptions vom Typ **`HeapFullException`**.
Nach Ausgabe der Exception-Ursache beendet dieser Handler die Funktion `main()` und damit das Programm.

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (2)

- **Definition der Exception-Klassen** (Header-Datei "Exceptbsp.h")

```
// --------------------------------------------------------------------------------
// Header-Datei Exceptbsp.h
// --------------------------------------------------------------------------------
// Definition von Exception-Klassen
// --------------------------------------------------------------------------------
// für Programm ExceptBsp -
// Beispiel- u. Demonstrationsprogramm zum Exception-Handling
// --------------------------------------------------------------------------------

#ifndef EXCEPTBSP_H
#define EXCEPTBSP_H


class MyException
{
  public :
    const char* getReason() const { return m_cpReason; }
  protected :
    explicit MyException(const char *str) { m_cpReason=str;}
  private :
    const char* m_cpReason;                        // Exception-Ursache
    MyException& operator=(const MyException&);   // privater Zuweisungsoperator
};


class HeapFullException : public MyException
{
  public :
    HeapFullException() : MyException("Heap is full") { }
};


class StackEmptyException : public MyException
{
  public :
    StackEmptyException() : MyException("Stack is empty") { }
};


template<class T> class StackFullException : public MyException
{
  public :
    StackFullException(T val) :
      MyException("Stack is full"), m_tValue(val) { }
    T getValue() const { return m_tValue; }
  private :
    T m_tValue;                                  // nicht mehr speicherbarer Wert
  };


#endif
```

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (3)

- **Definition und Implementierung des Klassen-Templates stack<T>** (Header-Datei "Stack.h")

```
#include <iostream>
using namespace std;
#include "Exceptbsp.h"

#define DEF_INC   0.2
#define DEF_SIZE  1000

template <class T> class Stack
{
  public :
    explicit Stack(int=DEF_SIZE);
    ~Stack() { delete[] m_tContent; }
    void push(T);
    T pop();
    void incCapacity(double=DEF_INC);
    // ...
    void outStack();
  private :
    T* m_tContent;    // Pointer auf Speicherplatz des Stacks
    int m_iTos;       // Index des ersten freien Speicherplatzes ("Top of Stack")
    int m_iCap;       // Kapazität des Stacks
};

// -----------------------------------------------------------------------------

template <class T> Stack<T>::Stack(int cap)
{
  m_iTos=0;
  m_iCap=cap;
  if ((m_tContent=new T[cap])==0)
    throw HeapFullException();
}

template <class T> void Stack<T>::push(T val)
{
  if (m_iTos>=m_iCap)
    throw StackFullException<T>(val);
  else
    m_tContent[m_iTos++]=val;
}

template <class T> T Stack<T>::pop()
{
  if (m_iTos<=0)
    throw StackEmptyException();
  else
    return m_tContent[--m_iTos];
}

template <class T> void Stack<T>::incCapacity(double inc)
{
  T* temp=new T[m_iCap=(unsigned)((1+inc)*m_iCap)+1];
  if (temp==0)
    throw HeapFullException();
  else
  { for (int i=0; i<m_iTos; i++)
      temp[i]=m_tContent[i];
    delete [] m_tContent;
    m_tContent=temp;
  }
}

template <class T> void Stack<T>::outStack()
{
  for (int i=m_iTos-1; i>=0; i--)
    cout << m_tContent[i] << " ";
}
```

Beispiel- und Demonstrationsprogramm zur Ausnahmebehandlung in C++ (4)

- **Implementierung des Testprogramms** (C++-Quell-Datei "Stacktest.cpp")

```
// --------------------------------------------------------------------------------
// Programm EXCEPTBSP
// --------------------------------------------------------------------------------
// Beispiel- u. Demonstrationsprogramm zum Exception-Handling
// --------------------------------------------------------------------------------

#include "Exceptbsp.h"
#include "Stack.h"

void testStack()
{
  Stack<char> cStack(5);
  char c;
  do
  { try
    {
      cout << "\nStackinhalt : ";
      cStack.outStack();
      cout << "\npop(-) oder push(+) oder quit(q) : ";
      cin >> c;
      if (c=='-')
        cout << cStack.pop() << endl;
      else
        if (c=='+')
        {
          char cNeu;
          cout << "Zeichen ? ";
          cin >> cNeu;
          cStack.push(cNeu);
        }
    }
    catch (StackFullException<char>& ex)
    {
      cout << ex.getReason() << endl;
      cStack.incCapacity();
      cout << "Stack capacity increased\n";
      cStack.push(ex.getValue());
    }
    catch (StackEmptyException& ex)
    {
      cout << ex.getReason() << endl << endl;
    }
  } while (c!='q');
  cout << endl;
}

int main()
{
  try
  {
    testStack();
    return 0;
  }
  catch (MyException& ex)
  {
    cout << ex.getReason() << endl << endl;
    return 1;
  }
}
```