

Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Ausnahmebehandlung

Laufzeitfehler

Weiterreichen von Ausnahmen

Mehrfache Ausnahmen

Ausnahmen als Klassen

Einschränkung von Ausnahmen

Fehlertoleranz

Zusicherungen

Zusammenfassung

Vorlesung 10. Ausnahmebehandlung

Ausnahmebehandlung

Laufzeitfehler

Weiterreichen von Ausnahmen

Mehrfache Ausnahmen

Ausnahmen als Klassen

Einschränkung von Ausnahmen

Fehlertoleranz

Zusicherungen

Zusammenfassung

10. Ausnahmebehandlung

Vorige Vorlesung

- ▶ Parametrische Polymorphie
- ▶ Template-Funktionen
- ▶ Template-Klassen
- ▶ Beispiele Stack und binärer Suchbaum

Heutige Vorlesung

- ▶ Laufzeitfehler
- ▶ Ausnahmebehandlung
- ▶ Fehlertoleranz
- ▶ Zusicherungen

Lernziele dieser Vorlesung

- ▶ Kenntnis des Ausnahmebehandlungsmechanismus in C++
- ▶ Verständnis der mehrstufigen Fehlerbehandlung
- ▶ Kenntnis von Zusicherungen und Fehlertoleranz

Laufzeitfehler und Ausnahmebehandlung

- ▶ Ein Programm, das in der Praxis eingesetzt wird, muss die **Behandlung möglicher Laufzeitfehler** vorsehen.
- ▶ Typische Laufzeitfehler treten auf, wenn Speicherplatz alloziert wird und nicht genügend Speicher vorhanden ist, oder wenn eine Division durch 0 erfolgen soll.
- ▶ C++ enthält einen speziellen Mechanismus zur Behandlung von Laufzeitfehlern (**Exception-handling**).
- ▶ Exception-handling ermöglicht dem Benutzer einer Klasse, beim Aufruf einer Member-Funktion, auftretende **Fehler** im Anwendungsprogramm sinnvoll **abzufangen**

try, catch und throw...

Die Ausnahmebehandlung in C++ beruht auf drei Konstrukten

try:

Programmteile (Program-statements), die vom **Ausnahmemechanismus** (**Exception-handler**) überwacht werden sollen, werden in einem **try**-Block zusammengefasst.

throw:

Wenn ein Laufzeitfehler in einem **try**-Block auftritt, wird eine **Ausnahme** (**Exception**) mittels **throw** ausgegeben.

catch:

Die Ausnahme wird mittels **catch** **aufgefangen und behandelt**.

...try, catch und throw...

```
try {  
    anweisungen  
}
```

```
catch(typ1 parameter) {  
    anweisungen  
}
```

```
catch(typ2 parameter) {  
    anweisungen  
}
```

...

```
catch(typN parameter) {  
    anweisungen  
}
```


...try, catch und throw

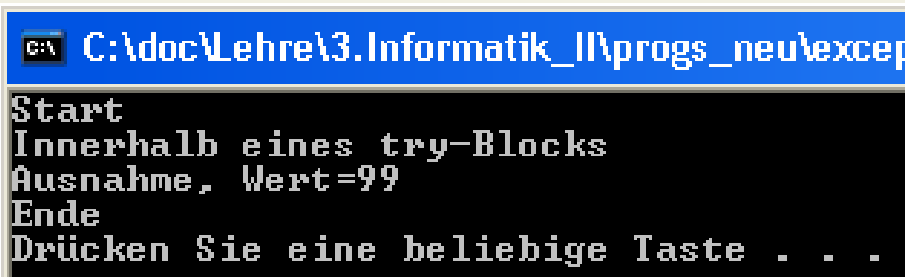
- ▶ Es können mehrere `catch`-Anweisungen zu einem `try`-Block gehören.
- ▶ Der Typ der Ausnahme bestimmt, welche `catch`-Anweisung ausgeführt wird.
- ▶ Soll das gesamte Programm mittels Exception-handler überwacht werden, muss die Funktion `main ()` innerhalb eines `try`-Blocks stehen.

Exception: Funktionsweise...

```
#include <iostream>
using namespace std;
int main ()
{
    cout << "Start" << endl;
    try { // Beginn des try-Blocks
        cout << "Innerhalb_eines_try-Blocks\n";
        throw 99; // Ausgabe einer Ausnahme
        cout << "Dies_wird_nicht_ausgefuehrt";
    } // Ende des try-Blocks
    catch (int i) { // Fange eine Ausnahme ab
        cout << "Ausnahme ,_Wert=";
        cout << i << endl;
    } // Ende catch
    cout << "Ende" << endl;
    return (EXIT_SUCCESS);
}
```

...Exception: Funktionsweise...

Programmausgabe



```
C:\doc\Lehre\3.Informatik_II\progs_neu\excep
Start
Innerhalb eines try-Blocks
Ausnahme, Wert=99
Ende
Drücken Sie eine beliebige Taste . . .
```

...Exception: Funktionsweise...

- ▶ Die `throw`-Anweisung führt die Programmsteuerung zum `catch`-Anweisungsblock.
- ▶ Der `catch`-Block wird **nicht aufgerufen**, sondern die Programmausführung wird dorthin **übertragen**.
- ▶ **Nach Ausführung** des `catch`-Anweisungsblocks **fährt** die Programmausführung mit der unmittelbar auf den `catch`-Anweisungsblock **folgenden Anweisung fort**.

...Exception: Funktionsweise...

`throw;`

`throw ausdruck;`

- ▶ Der Datentyp von *ausdruck* muss mit dem Datentyp einer `catch`-Anweisung **übereinstimmen**.
- ▶ Eine `throw`-Anweisung ohne folgenden *ausdruck* reicht eine Ausnahme weiter (**rethrow**); wird benutzt, wenn ein weiterer Exception-handler zur Fehlerbehandlung eingesetzt werden soll.
- ▶ Bei **geschachtelten** `try`-Blöcken wird der **innerste** `try`-Block, in dem eine Ausnahme mittels `throw` gegeben wird, zur Auswahl des **passenden** `catch`-Anweisungsblocks gewählt.

...Exception: Funktionsweise...

- ▶ Der durch `throw` gegebene *ausdruck* ist ein **statisches temporäres Objekt**, das so lange existiert, bis der zugehörige Exception-handler (`catch`) verlassen wird.
- ▶ Der Exception-handler kann diesen *ausdruck* benutzen.

```
void foo ()
{
    int i;
    // ...
    throw i;
}
```

```
int main ()
{
    try {
        foo ();
    }
    catch (int n) { /*...*/ }
    // ...
}
```

...Exception: Funktionsweise...

- ▶ Bei Ausnahmen in **geschachtelten Funktionen** wird der Prozess-Stack soweit zurückgenommen, bis ein Exception-handler gefunden wird.
- ▶ Damit werden beim **Verlassen** eines lokalen Prozesses (innere Funktion) alle zugehörigen Objekte (Variablen, Instanzen) **zerstört**.

...Exception: Funktionsweise

```
void foo ()
{
    int i, j;
    // ...
    throw i;
    // ...
}

void call_foo ()
{
    int k;
    // ...
    foo ();
    // ...
}
```

```
int main ()
{
    try {
        call_foo ();
        // foo () beendet,
        // i, j zerstört
    }
    catch (int n) {...}
}
```


Weiterreichen von Ausnahmen

- ▶ Ausnahmen werden durch eine `throw/catch`-Sequenz behandelt.
- ▶ Solche Ausnahmen können auch nach außerhalb einer `throw/catch`-Sequenz **weitergereicht** werden.
- ▶ Es können **mehrfache Ausnahmebehandlungen** vorgenommen werden, das heißt, es kann verschiedene Exception-handler für verschiedene Aspekte einer Ausnahme geben.
- ▶ Weiterreichen von Ausnahmen kann **nur innerhalb eines catch-Anweisungsblocks** geschehen.

Beispiel Weiterreichen von Ausnahmen...

```
#include <iostream>
using namespace std;

void Xhandler ()
{
    try {
        throw (22);
    }
    catch (int i) {
        cerr << "Ausnahme int ";
        cerr << "innerhalb von Xhandler. ";
        cerr << "Wert: ";
        cerr << i << endl;
        throw; // Weiterreichen von int
    }
}
```

...Beispiel Weiterreichen von Ausnahmen...

```
int main ()
{
    cout << "Start" << endl;
    try {
        Xhandler ();
    }
    catch (char *s) {
        cerr << "Ausnahme_int_";
        cerr << "innerhalb_von_main._Wert:_";
        cerr << i << endl;
    }
    cout << "Ende" << endl;

    return (EXIT_SUCCESS);
}
```

...Beispiel Weiterreichen von Ausnahmen

Programmausgabe

```
C:\> C:\doc\Lehre\3.Informatik_II\progs_neu\exception2.exe
```

```
Start  
Xhandler start  
Ausnahme int innerhalb von Xhandler. Wert: 22  
Ausnahme int innerhalb von main. Wert: 22  
Ende  
Drücken Sie eine beliebige Taste . . . _
```

Mehrfache Ausnahmen

- ▶ Es können **mehrere catch**-Anweisungsblöcke zu **einem try**-Block implementiert werden
- ▶ Die **catch**-Anweisungen müssen jedoch mit **verschiedenen Datentypen** verbunden sein
- ▶ Es gibt eine **typunabhängige catch**-Anweisung, mit der beliebige mit **throw** gegebene Ausnahmen behandelt werden können (**allgemeines catch**)

```
catch(...) {  
    anweisungen  
}
```

Beispiel Mehrfache Ausnahmen...

```
vektor::vektor(int n)
{
    if (n < 1)
        throw (n);
    p = new int[n];
    if (p == 0)
        throw ("Kein_Speicher_mehr");
}

void g ()
{
    try {
        vektor a(n), b(n);
        // ...
    }
    catch (int n) {...} // Behandelt inkorrekte Groesse
    catch (char *error) {...} // Behandelt Speicherefehler
}
```

...Programmabbruch...

- ▶ Gewöhnlich wird ein `catch`-Anweisungsblock mit der C++ Standard-Bibliotheksfunktion `exit ()` oder `abort ()` beendet (Ausnahmebehandlung von **katastrophalen Fehlern**).
- ▶ `void abort ()`
 - ▶ bewirkt die **sofortige Beendigung** des Programms
 - ▶ kein Rückgabewert an das Betriebssystem, schließt geöffnete Dateien nicht
- ▶ `void exit (int status)`
 - ▶ **geordnete Beendigung** des Programms
 - ▶ schließt alle geöffneten Dateien
 - ▶ beendet das Programm mit einem Rückgabewert an das Betriebssystem

...Beispiel Mehrfache Ausnahmen...

Standardaktion für Mehrfachausnahmen

```
catch (const char *meldung) // Fange String meldung
{
    cerr << meldung << endl;
    exit (EXIT_FAILURE); // definiert in stdlib.h
}

catch (...) // Standardaktion,
             //falls niemand anders faengt
{
    cerr << "Hasta_la_vista,_baby!"
         << endl;
    abort ();
}
```


...Beispiel Mehrfache Ausnahmen...

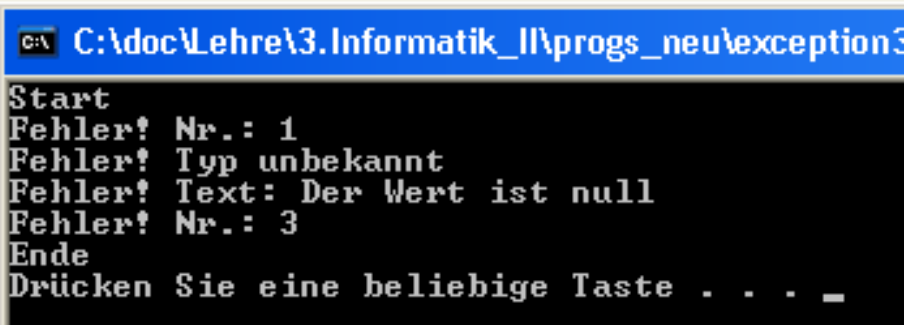
```
void Xhandler (int test)
{
    try {
        if (test==0) throw "Der_Wert_ist_null";
        if (test==1 || test==3) throw test;
        if (test==2) throw 22.89;
        else throw "Der_Wert_ist_zu_gross";
    }
    catch (int i) { // Fange int
        cerr << "Fehler!_Nr.:_" << i << endl;
    }
    catch (const char *str) { // Fange String
        cerr << "Fehler!_Text:_" << str << endl;
    }
    catch (...) { // Fange andere Typen
        cerr << "Fehler!_Typ_unbekannt" << endl;
    }
}
```

...Beispiel Mehrfache Ausnahmen...

```
int main ()
{
    cout << "Start" << endl;
    Xhandler (1); // wirft int
    Xhandler (2); // wirft float
    Xhandler (0); // wirft String
    Xhandler (3); // wirft int
    cout << "Ende" << endl;
    return (EXIT_SUCCESS);
}
```

...Beispiel Mehrfache Ausnahmen

Programmausgabe



```
C:\doc\Lehre\3.Informatik_II\progs_neu\exception3
Start
Fehler! Nr.: 1
Fehler! Typ unbekannt
Fehler! Text: Der Wert ist null
Fehler! Nr.: 3
Ende
Drücken Sie eine beliebige Taste . . . _
```

Ausnahmen als Klassen

- ▶ Für Ausnahmebehandlung von **komplexen Fehlern** sollten **Fehlerklassen** implementiert werden.
- ▶ Die entsprechenden Objekte enthalten dann Informationen über die **Umgebung** (Objektzustände, Variablenwerte, etc.) des **Fehlerzustands**.

```
enum error { grenze, speicher, andere }
class vekt_error {
private:
    error e_typ;
    int og, index, groesse;
public:
    vekt_error (error, int, int); // Grenzen
    vekt_error (error, int); // out of memory
    // ...
};
// ...
throw vekt_error (grenze, index, obergrenze);
// ...
```

Matching von throw und catch

- ▶ Ein mit `throw` gegebener Ausdruck passt auf einen Parameter von `catch`, falls
 - ▶ die Signatur genau übereinstimmt,
 - ▶ der `catch`-Parameter eine öffentliche Basisklasse des `throw`-Ausdrucks ist,
 - ▶ der `throw`-Parameter ein Zeiger ist, der konvertierbar zu einem Zeigertyp des `catch`-Parameters ist.
- ▶ **Vorsicht bei der Reihenfolge der catch-Anweisungsblöcke**

```
catch(void*) // auch char* passt
catch(char*) // wird nie benutzt
catch(Basistyp&) // passt immer auf Abgeleitet
catch(Abgeleitet&) // wird nie benutzt
```

Einschränkung von Ausnahmen. . .

```
typ funktionsbezeichner (parameterliste) throw (typliste)
{
    anweisungen
}
```

- ▶ Funktionen, die innerhalb von `try`-Blöcken Ausnahmen ausgeben, können bzgl. der **Datentypen** dieser Ausnahmen **eingeschränkt** werden.
- ▶ `typliste` gibt an, welche Datentypen für ein `throw` innerhalb von `funktionsbezeichner` benutzt werden dürfen.

...Einschränkung von Ausnahmen

- ▶ Eine *leere typliste verbietet* es einer Funktion, Ausnahmen zu geben.
- ▶ Ist innerhalb einer Funktion ein `try`-Block, so gelten die Einschränkungen für diese Funktion nicht, wenn *innerhalb* der Funktion die mit `throw` gegebenen Ausnahmen auch *innerhalb* der Funktion durch `catch`-Anweisungsblöcke abgefangen werden.

Beispiel Stack...

Ausnahmen eines Stack: Bereichsüberschreitungen

- ▶ Pop auf leeren Stack
- ▶ Push auf vollen Stack

Klassenstruktur Stack

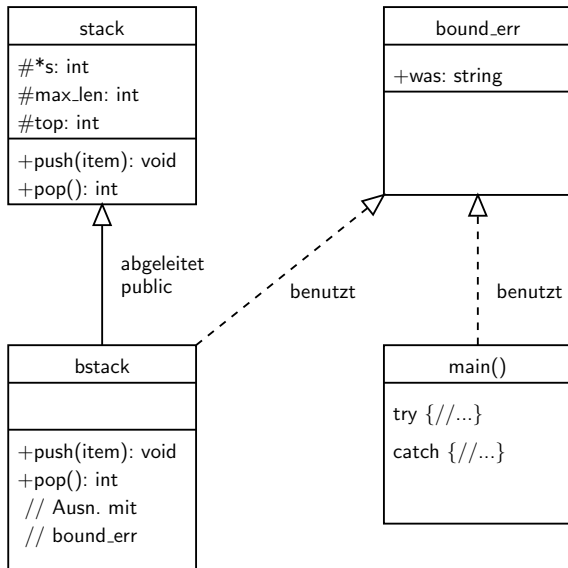
Klasse `stack` als Basisklasse für eine Klasse `b_stack`, die mit Ausnahmebehandlung Bereichsüberschreitungen abfängt.

...Beispiel Stack

Implementierungsprinzip:

- ▶ Standardmäßige Klasse `stack` (ohne Ausnahmebehandlung)
- ▶ Abgeleitete Klasse, die die relevanten Member-Funktionen (`push ()` und `pop ()`) um Exception-handling erweitert
- ▶ Fehlerklasse `bound_err`
- ▶ Benutzung von Ausnahmeeinschränkung

Beispiel Stack Klassendiagramm



Beispiel Stack err.h

```
#include <cstring>
const int WAS_MAX=80; // Max Laenge der Fehlermeldung

class bound_err {
public:
    char was[WAS_MAX]; // Was war der Fehler

    bound_err (char *_was) {
        if (strlen (_was) < (WAS_MAX - 1)) {
            strcpy (was, _was);
        } else {
            strcpy (was, "Interner Fehler: _was ist zu lang");
        }
    }
};
```

Beispiel Stack stack.h

```
#include "err.h"
class stack {
protected:
    enum { EMPTY = -1};
    int *s, max_len, top;
public:
    stack ();
    stack (int size);
    ~stack ();
    void reset ();
    void push (int c);
    int pop ();
    int top_of ();
    bool empty ();
    bool full ();
};
```

Beispiel Stack bstack.h

- ▶ Klasse stack ohne Ausnahmebehandlung
- ▶ Klasse b_stack abgeleitet von stack
- ▶ Ausnahmerelevante Member-Funktionen push (), pop ()
- ▶ Einschränkung der Ausnahmen auf Fehlerklasse bound_err

```
class b_stack: public stack {  
public:  
    void push (int c) throw (bound_err);  
    int pop () throw (bound_err);  
};
```

Beispiel Stack bstack.cpp...

```
#include "stack.h"
void b_stack::push (int c) throw (bound_err)
{ // Ausnahme-Einschraenkung
  if (full ()) {
    bound_err overflow ("Push_nicht_moeglich,_Stack_voll");
    throw overflow;
  }
  stack::push (c);
}

int b_stack::pop () throw (bound_err)
{ // Ausnahme-Einschraenkung
  if (empty ()) {
    throw bound_err ("Pop_nicht_moeglich,_Stack_leer");
  }
  return (stack::pop ());
}
```

...Beispiel Stack stack.cpp...

```
#include "stack.h"

stack::stack ()
  : max_len (1000) {
  s=new int [1000]; top=EMPTY;
}

stack::stack(int size)
  : max_len (size) {
  s=new int [size]; top=EMPTY;
}

stack::~~stack () {
  delete [] s;
}

void stack::reset () {
  top=EMPTY;
}
```

...Beispiel Stack stack.cpp

```
void stack::push (int c) {
    s[++top]=c;
}
int stack::pop () {
    return (s[top--]);
}
int stack::top_of () {
    return (s[top]);
}
bool stack::empty () {
    return (bool(top==EMPTY));
}
bool stack::full () {
    return (bool(top==max_len-1));
}
```


Beispiel Stack main.cpp...

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

b_stack test_stack;

void push_viel (int viel) {
    for (int i=0; i<viel; i++) {
        test_stack.push (i);
        cout << test_stack.top_of () << endl;
    }
}

void pop_viel(int viel) {
    for (int i=0; i<viel; i++) {
        cout << test_stack.top_of () << endl;
        test_stack.pop ();
    }
}
```

...Beispiel Stack main.cpp...

```
int main ()
{
    int po_viel, pu_viel;

    cout << "Wie_viel_push?_";
    cin >> pu_viel;

    cout << "wie_viel_pop?_";
    cin >> po_viel;
```

...Beispiel Stack main.cpp

```
try {
    push_viel (pu_viel);
    pop_viel (po_viel);
}
catch (bound_err err) {
    cerr << "Fehler:␣";
    cerr << "Stack-Grenzen␣ueberschritten" << endl;
    cerr << "Grund:␣" << err.was << endl;
    exit (EXIT_FAILURE);
}
catch (...) {
    cerr << "Fehler:␣";
    cerr << "unerwartete␣Ausnahmesituation" << endl;
    exit (EXIT_FAILURE);
}
return (EXIT_SUCCESS);
}
```

Exception: Beispiel Stack

Programmausgabe

```
C:\> C:\doc\Lehre\3.Informatik_II\progs_neu\except\Stack.exe
Wie viel push? 2
wie viel pop? 4
0
1
1
0
524571
Fehler: Stack-Grenzen ueberschritten
Grund: Pop nicht moeglich, Stack ist leer
Druecken Sie eine beliebige Taste . . .
```

Fehlertoleranz

- ▶ **Exception-handler** eignen sich gut zum implementieren fehlertoleranter Programme.
- ▶ **Fehlertoleranz** bedeutet, dass ein Programm auch bei **Laufzeitfehlern** weiterhin **funktionsstüchtig** bleibt.
- ▶ Eine wichtige Anwendung: **Korrektur von illegalen Werten**.
- ▶ Vorteil gegenüber direkter Korrektur im entsprechenden Programm-Code: Klare **Unterscheidung** zwischen einem **Fehler** und dessen **Behandlung**.

Beispiel Fehlertoleranz...

- ▶ Exception-handler ersetzt einen illegalen Wert durch einen standardmäßigen legalen Wert.
- ▶ Dieses Vorgehen ist besonders sinnvoll bei der Systementwicklung und beim Integrationstest.

```
vektor::vektor (int n)
{
    if (n < 1)
        throw (n); // Falsche Groesse
    p = new int[n];
    if (p == 0)
        throw ("Kein_Speicher_mehr");
}
```

...Beispiel Fehlertoleranz

```
void g(int m)
{
    try {
        vektor a(m);
        // ...
    }
    catch (int n)
    {
        cerr << "Groessenfehler_" << n << endl;
        g (10); // Versuch mit legalem Wert
    }
    catch (const char *error)
    {
        cerr << error << endl;
        abort ();
    }
}
```

Fehlertoleranz und Klassenkonstruktoren...

- ▶ Allgemeines Schema zur Implementierung fehlertoleranter Klassen.
- ▶ Der Objekt-Konstruktor enthält Ausnahmen für illegale Zustände.
- ▶ Es gibt eine hierarchische Struktur von Fehlerklassen.
- ▶ Der `try`-Block benutzt die Information zur Reparatur oder zum Abbruch des nicht-korrekten Programm-Codes.

...Fehlertoleranz und Klassenkonstruktoren

```
Objekt::Objekt(parameter)
{
    if (illegalparameter1)
        throw ausdruck1;
    if (illegalparameter2)
        throw ausdruck2;
    ...
    // Konstruktion
    ...
}
```

```
try {
    // Fehlertoleranter Programm-Code
}
catch(deklaration1) { /* Reparatur 1 */}
catch(deklaration2) { /* Reparatur 2 */}
// ...
// Werte sind jetzt legal
```

Fehlerklassen

```
class objektfehler {
public:
    objektfehler(parameter);
    // Members mit Daten des Fehlers
    virtual void reparatur()
    { cerr << "Reparatur nicht moeglich"; endl; abort(); }
};
```

```
class objektfehler1 : public objektfehler {
public:
    objektfehler1(parameter);
    // Weitere Members mit Daten des Fehlers
    void reparatur();
    // Passende Reparatur
};
// Andere abgeleitete Fehlerklassen
```

Philosophie von Error-recovery

- ▶ Eine Methode, Fehlertoleranz zu erreichen ist **Error-recovery**.
- ▶ Error-recovery bedeutet die **Reparatur** fehlerhafter Zustände im Programmlauf.
- ▶ Erreichbar ist dies durch **Eingriff in den Programmsteuerfluss** (Ausnahmebehandlung).
- ▶ Zu viel Eingriff in den Steuerfluss führt zu Chaos.
- ▶ Meist ist die Reparatur nur bei Echtzeit-Programmen sinnvoll.
- ▶ In den meisten Fällen ist es sinnvoll, bei Fehlern eine **Diagnose** auszugeben und das Programm **sicher** zu beenden.

Zusicherungen...

- ▶ **Programmkorrektheit** bedeutet, dass auf **korrekte Eingabe** eine **korrekte Ausgabe** erfolgt.
- ▶ Der **aufrufende Teil** hat die Verantwortung für eine **korrekte Eingabe**.
- ▶ Der **aufgerufene Teil** hat die Verantwortung für eine **korrekte Ausgabe**.
- ▶ **Programming-by-contract**: Regelung der Verantwortlichkeiten, typisch für objektorientierte Programmierung.
- ▶ In **Member-Funktionen** von Klassen sollte die **Erfüllung der Pflichten geprüft** werden.

...Zusicherungen...

- ▶ Die Pflichten aus den Verantwortlichkeiten werden durch **Zusicherungen** gegeben.
- ▶ Eine **Zusicherung (Assertion)** ist
 - ▶ **Vorbedingung** (Precondition), d. h. korrekte Eingabe,
 - ▶ **Nachbedingung** (Postcondition), d. h. korrekte Ausgabe,
 - ▶ **Invariante** (Invariant), d. h. das, was sich nicht ändern darf.
- ▶ C++ bietet eine Möglichkeit, Zusicherungen zu implementieren und zu prüfen.

...Zusicherungen...

- ▶ Die Standard-Bibliothek `assert.h` enthält ein **Makro**

```
assert (int expression);
```

- ▶ Wirkungsweise:
 - ▶ Wird `expression` zu **falsch** ausgewertet, so wird die **Programmausführung** mit einer Diagnose-Ausgabe **abgebrochen**.
 - ▶ Wird `expression` zu **wahr** ausgewertet, so wird mit der **Programmausführung fortgefahren**.

...Zusicherungen

- ▶ **Direkt** zu implementieren sind in C++ nur **Vorbedingungen** und **Nachbedingungen**.

```
vekt::vekt(int n)
{
    assert (n > 0); //Vorbedingung
    groesse = n;
    p = new int[groesse];
    assert (p != 0); //Nachbedingung
}
```

- ▶ **Invarianten** sollten **gesondert implementiert** werden.

Beispiel String...

```
#include <string.h>
#include <assert.h>
class String{
    char *Buffer;
    int Buflen;
    void Invariant ();
public:
    String (char *str);
    ~String ();
    String &operator= (char *str);
};
```


...Beispiel String...

```
inline void String::Invariant () {
    assert (strlen (Buffer) = BufLen);
};

String::String (char *str) {
    assert (str != NULL);
    BufLen = strlen (str) + 1
    Buffer = new char [BufLen];
    assert (Buffer != NULL);
    strcpy (Buffer, str);
    Invariant ();
}
```

...Beispiel String

```
String::~~String () {
    Invariant ();
    delete [] Buffer;
}

String &String::operator= (char *str) {
    Invariant ();
    assert (str != NULL);
    if (strlen(str) >= Buflen) {
        delete [] Buffer;
        Buflen = strlen (str) + 1;
        Buffer = new char[Buflen];
        assert (Buffer != NULL);
    }
    strcpy (Buffer, str);
    return (*this);
}
```

Eine mögliche assert()-Definition

```
#include <stdio.h>
#include <stdlib.h>
#if defined(NDEBUG)
    #define assert(ignore) ((void) 0) /*ignorieren */
#else
    #define assert(expr) \
        if (!(expr)) { \
            printf("\n%s%s\n%s%s\n%s%d\n\n", \
                "Assertion_ failed:", #expr, \
                "in_ file_", __FILE__, \
                "at_ line_", __LINE__); \
            abort(); \
        }
#endif
```

Ausnahmebehandlung

- ▶ **Laufzeitfehler** können in C++ mit `throw`, `try` und `catch` behandelt werden.
- ▶ Ausnahmen können über **mehrere Ebenen** weitergereicht werden.
- ▶ Ausnahmebehandlung kann an einen **Typ gebunden** werden (mehrfache Ausnahmen).
- ▶ Komplexe Laufzeitfehler können in **Fehlerklassen** behandelt werden.
- ▶ Ausnahmebehandlung eignet sich gut zur Implementierung von **Fehlertoleranz**.
- ▶ **Zusicherungen** können bei der Programmentwicklung zur **Qualitätskontrolle** eingesetzt werden.