# Ausnahmebehandlung

- Was ist eine Ausnahme,
  und warum kümmere ich mich darum?
- Benutzen von Ausnahmen
- Vordefinierte Ausnahmeklassen
- Eigene Ausnahmeklassen

- Besondere Fehlerbehandlungsfunktionen
- Durch Ausnahmen verursachte Speicherlecks vermeiden
- Speicherbeschaffung mit `new`

- Zusammenfassung

# Ausnahmebehandlung

## Was ist eine Ausnahme, und warum kümmere ich mich darum?

Ein einfaches C++-Programm:

```
int main()
{
  int i = 1, j = 0;

  return i / j;
}
```

Übersetzen und Starten des Programms führt z.B. im Borland C++ Builder zu folgendem Ergebnis:

```
Project e1.exe raised exception class
EDivByZero with message
'Division by zero'.
Process stopped. Use Step or Run to continue.
```

# Ausnahmebehandlung

## Was ist eine Ausnahme, und warum kümmere ich mich darum?

Ein einfaches C++-Programm:

```
int main()
{
  int *ip = 0;

  return *ip;
}
```

Übersetzen und Starten des Programms führt z.B. im Borland C++ Builder zu folgendem Ergebnis:

```
Project e1.exe raised exception class
EAccessViolation with message
'Access violation at address 0040115c.
Read of address 00000000'.
Process stopped. Use Step or Run to continue.
```

# Ausnahmebehandlung

## Was ist eine Ausnahme, und warum kümmere ich mich darum?

Der Begriff *Ausnahme* ist eine Abkürzung für "außergewöhnliches Ereignis".

> Definition:
>
> Eine *Ausnahme* ist ein Ereignis, das während der Ausführung eines Programms eintritt und den normalen Anweisungsablauf stört.

Wenn in einer C++-Funktion eine Ausnahme auftritt, erzeugt die Funktion ein *Ausnahmeobjekt* und übergibt dieses an das C++-Laufzeitsystem.

Das Laufzeitsystem ist dann verantwortlich für das Auffinden von Programmteilen, welche die Ausnahme behandeln.

Die Verwendung von Ausnahmeobjekten (kurz: Ausnahmen) zum Umgang mit Fehlern hat die folgenden Vorteile:

- Trennen von Fehlerbehandlung und "regulärem" Ablauf
- Fortpflanzen von Fehlern durch den Aufrufstapel
- Gruppieren und Differenzieren von Fehlertypen

## Trennung von Fehlerbehandlung und "regulärem" Ablauf

Beispiel: ohne Fehlerbehandlung

```
readFile
{
  open the file;
  determine its size;
  allocate that much memory;
  read the file into memory;
  close the file;
}
```

# Trennung von Fehlerbehandlung und "regulärem" Ablauf

Beispiel (Fortsetzung): mit "klassischer" Fehlerbehandlung

```
errorCodeType readFile
{
  initialize errorCode = 0;
  open the file;
  if (theFileIsOpen)
  {
    determine the length of the file;
    if (gotTheFileLength)
    {
      allocate that much memory;
      if (gotEnoughMemory)
      {
        read the file into memory;
        if (readFailed)
          errorCode = -1;
      }
      else
        errorCode = -2;
    }
    else
      errorCode = -3;
    close the file;
    if (theFileDidntClose && errorCode == 0)
      errorCode = -4;
    else
      errorCode = errorCode and -4;
  }
  else
    errorCode = -5;
  return errorCode;
}
```

# Trennung von Fehlerbehandlung und "regulärem" Ablauf

## Beispiel (Fortsetzung): mit "klassischer" Fehlerbehandlung

```
errorCodeType readFile
{
  initialize errorCode = 0;
  open the file;
  if (theFileIsOpen)
  {
    determine the length of the file;
    if (gotTheFileLength)
    {
      allocate that much memory;
      if (gotEnoughMemory)
      {
        read the file into memory;
        if (readFailed)
          errorCode = -1;
      }
      else
        errorCode = -2;
    }
    else
      errorCode = -3;
    close the file;
    if (theFileDidntClose && errorCode == 0)
      errorCode = -4;
    else
      errorCode = errorCode and -4;
  }
  else
    errorCode = -5;
  return errorCode;
}
```

# Trennung von Fehlerbehandlung und "regulärem" Ablauf

Beispiel (Fortsetzung): mit Ausnahmebehandlung

```
readFile
{
  try
  {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
  }
  catch (fileOpenFailed)
  {
    doSomething;
  }
  catch (sizeDeterminationFailed)
  {
    doSomething;
  }
  catch (memoryAllocationFailed)
  {
    doSomething;
  }
  catch (readFailed)
  {
    doSomething;
  }
  catch (fileCloseFailed)
  {
    doSomething;
  }
}
```

# Was ist eine Ausnahme, und warum kümmere ich mich darum?

## Fortpflanzen von Fehlern durch den Aufrufstapel

Beispiel: ohne Fehlerbehandlung

```
method1
{
  call method2;
}

method2
{
  call method3;
}

method3
{
  call readFile;
}
```

# Fortpflanzen von Fehlern durch den Aufrufstapel

Beispiel (Fortsetzung): mit "klassischer" Fehlerbehandlung

```
method1
{
  errorCodeType error;
  error = call method2;
  if (error)
    doErrorProcessing;
  else
    proceed;
}

errorCodeType method2
{
  errorCodeType error;
  error = call method3;
  if (error)
    return error;
  else
    proceed;
}

errorCodeType method3
{
  errorCodeType error;
  error = call readFile;
  if (error)
    return error;
  else
    proceed;
}
```

# Fortpflanzen von Fehlern durch den Aufrufstapel

Beispiel (Fortsetzung): mit Ausnahmebehandlung

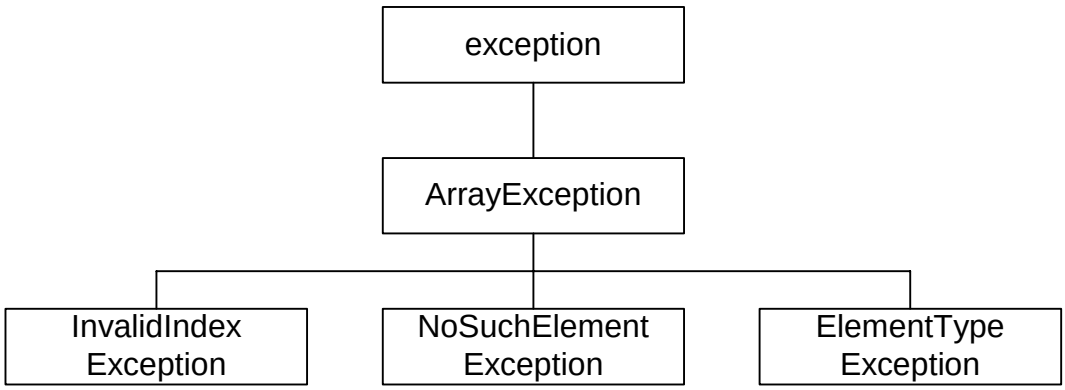
```
method1
{
  try
  {
    call method2;
  }
  catch (exception)
  {
    doErrorProcessing;
  }
}

method2 throw(exception)
{
  call method3;
}

method3 throw(exception)
{
  call readFile;
}
```

# Was ist eine Ausnahme, und warum kümmere ich mich darum?

## Gruppieren und Differenzieren von Fehlertypen

Beispiel:

Eine Gruppe von Ausnahmen, von denen jede einen spezifischen Fehlertyp darstellt, der bei Feldveränderung eintreten kann.




Abfangen von Ausnahmen wegen ungültigem Index:

```
catch (InvalidIndexException iie) { .. }
```

Abfangen *aller* Feldausnahmen:

```
catch (ArrayException ae) { .. }
```

# Benutzen von Ausnahmen

## Ablauf der Ausnahmebehandlung:

1. Ein Block von Anweisungen versucht (englisch ***try***) die Erledigung einer Aufgabe.

2. Wenn ein Fehler festgestellt wird, der nicht behoben werden kann, wirft (englisch ***throw***) der Block eine Ausnahme (englisch *exception*) aus.

3. Die Ausnahme wird von einem Ausnahmebehandler aufgefangen (englisch ***catch***), der den Fehler bearbeitet.

# Benutzen von Ausnahmen

## syntaktische Struktur der Ausnahmebehandlung:

```
try
{
  // Normalerweise läuft dieser Code ohne Probleme
  // vom Anfang des Blocks bis zum Ende durch.
  // Manchmal können aber auch Ausnahmen ausgelöst
  // werden.
  ..
  if (..)
    throw SomeException();
  ..
  if (..)
    throw AnotherException();
  ..
}

catch (SomeException se)
{
  // Behandle ein Ausnahme-Objekt se vom Typ
  // SomeException oder einer Unterklasse
  // dieses Typs.
}

catch (AnotherException ae)
{
  ..
}
```

# Benutzen von Ausnahmen

## Auslösen von Ausnahmen

implizit:

- das Programm führt etwas Unzulässiges aus

  Beispiele:

  - 
  - 
  - 

explizit:

- durch die `throw`-Anweisung:

  `throw` *`Ausnahmeobjekt`*`;`

Ein Ausnahmeobjekt ist ein Objekt eines beliebigen Datentyps. Ausgenommen ist nur der Typ `void`.

# Benutzen von Ausnahmen

## Abfangen und Behandeln von Ausnahmen

Ausnahmebehandler (`catch`-Blöcke) werden durch *Typen* unterschieden.

Die Reihenfolge der `catch`-Blöcke ist wichtig. Es wird immer **der erste passende** `catch`-Block ausgeführt.

Es wird der ***erste*** Ausnahmebehandler ausgeführt,

- dessen Typ genau mit dem Typ der signalisierten Ausnahme (`throw`-Anweisung) übereinstimmt
- oder dessen Typ eine Basisklasse der signalisierten Ausnahme ist
- oder dessen Typ ein Zeiger auf eine Klasse ist und das ausgeworfene Fehlerobjekt ein Zeiger auf ein Objekt einer abgeleiteten Klasse.

- `catch (void *)` fängt alle Ausnahmen auf, die Zeiger sind.
- `catch (...)` fängt ***alle*** Ausnahmen auf.

Nichtangabe eines Datentyps durch `...` bedeutet „beliebige Anzahl beliebiger Datentypen“.

# Benutzen von Ausnahmen

## Abfangen und Behandeln von Ausnahmen

Beispiel:

```
try {
  someReallyExceptionalMethod();
} catch (invalid_argument &) {
  ..
} catch (logic_error &) {
  ..
} catch (MyFirstException) {
  ..
} catch (exception &) {
  ..
} catch (int) {
  ..
} catch (char *) {
  ..

} catch (void *) {  // faengt alle Ausnahmen auf,
  ..                // die Zeiger sind

} catch (...) {     // faengt alle Ausnahmen auf

  throw;            // Weitergabe der Ausnahme
}                   // an die naechsthoehere Instanz
```

# Benutzen von Ausnahmen

## Abfangen und Behandeln von Ausnahmen

Beispiel: Einlesen von Zahlen

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

class DateiEnde {};   // Hilfsklasse

int liesZahl(istream& ein)
  throw(DateiEnde, const char*)
{
  int i;
  ein >> i;
  if (ein.eof())
    throw DateiEnde();
  if (ein.fail())
    throw "Syntaxfehler";
  if (ein.bad())
    throw;            // nicht behebbarer Fehler
                      // Aufruf von terminate()
  return i;
}

void Zahlen_lesen_und_ausgeben();  // Funktionsprototyp

int main()
{
  Zahlen_lesen_und_ausgeben();
}
```

# Benutzen von Ausnahmen

Beispiel: Einlesen von Zahlen (Fortsetzung)

```
void Zahlen_lesen_und_ausgeben()
{
  int zahl;
  while (true)
  {
    cout << "Zahl eingeben:";

    bool erfolgreich = true;
    // Versuchsblock
    try
    {
      //  Zahl von der Standardeingabe lesen:
      zahl = liesZahl(cin);
    }

    // Fehlerbehandlung
    catch (DateiEnde) // Objektname hier nicht notwendig
    {
      cout << "Ende der Datei erreicht!" << endl;
      cin.clear();    // Fehlerbits ruecksetzen
      break;          // Schleife verlassen
    }

    catch (const char* z)
    {
      cerr << z << endl;
      erfolgreich = false;
      cin.clear();    // Fehlerbits ruecksetzen
      cin.get();      // fehlerhaftes Zeichen entfernen
    }

    // Fortsetzung des Prog. nach der Fehlerbehandlung
    if (erfolgreich)
      cout << "Zahl =  " << zahl << endl;
  }
}
```

# Benutzen von Ausnahmen

## Abfangen und Behandeln von Ausnahmen

geschachtelte Ausnahmebehandlung:

- Falls für eine ausgelöste Ausnahme in diesem Anweisungsblock kein Ausnahmebehandler vorhanden ist, so wird die Ausnahme in den umfassenden Block weitergegeben
- Falls in dieser Methode kein umfassender Block mehr vorhanden ist, so wird im Block der Aufrufstelle nach einem Ausnahmebehandler gesucht.
- usw.

Falls kein Ausnahmebehandler gefunden wird, wird das C++-Programm abgebrochen.

# Benutzen von Ausnahmen

## Abfangen und Behandeln von Ausnahmen

### Beispiel: geschachtelte Ausnahmebehandlung

```
class Exception1 {};
class Exception2 {};
class Exception3 {};

class X
{
   public : void xx() throw (Exception3)
   {
      try
      {
         try { throw Exception3(); }
         catch (Exception1 e1)
         {
            // ..
         }
      }
      catch (Exception2 e2)
      {
         // ..
      }
   }
};

class Y
{
   public : void yy()
   {
      X xobj = X();
      try
      {
         xobj.xx();
      }
      catch (Exception3 e3)
      {
         // ..
      }
   }
};
```

# Benutzen von Ausnahmen

## Abfangen und Behandeln von Ausnahmen

### Ausnahmen

- entweder **abfangen**
- oder **weiterleiten**

Methoden können durch den **`throw`**-Zusatz eine Liste der Ausnahmen angeben, die sie nach "außen" auslösen können. Dabei werden drei Fälle unterschieden:

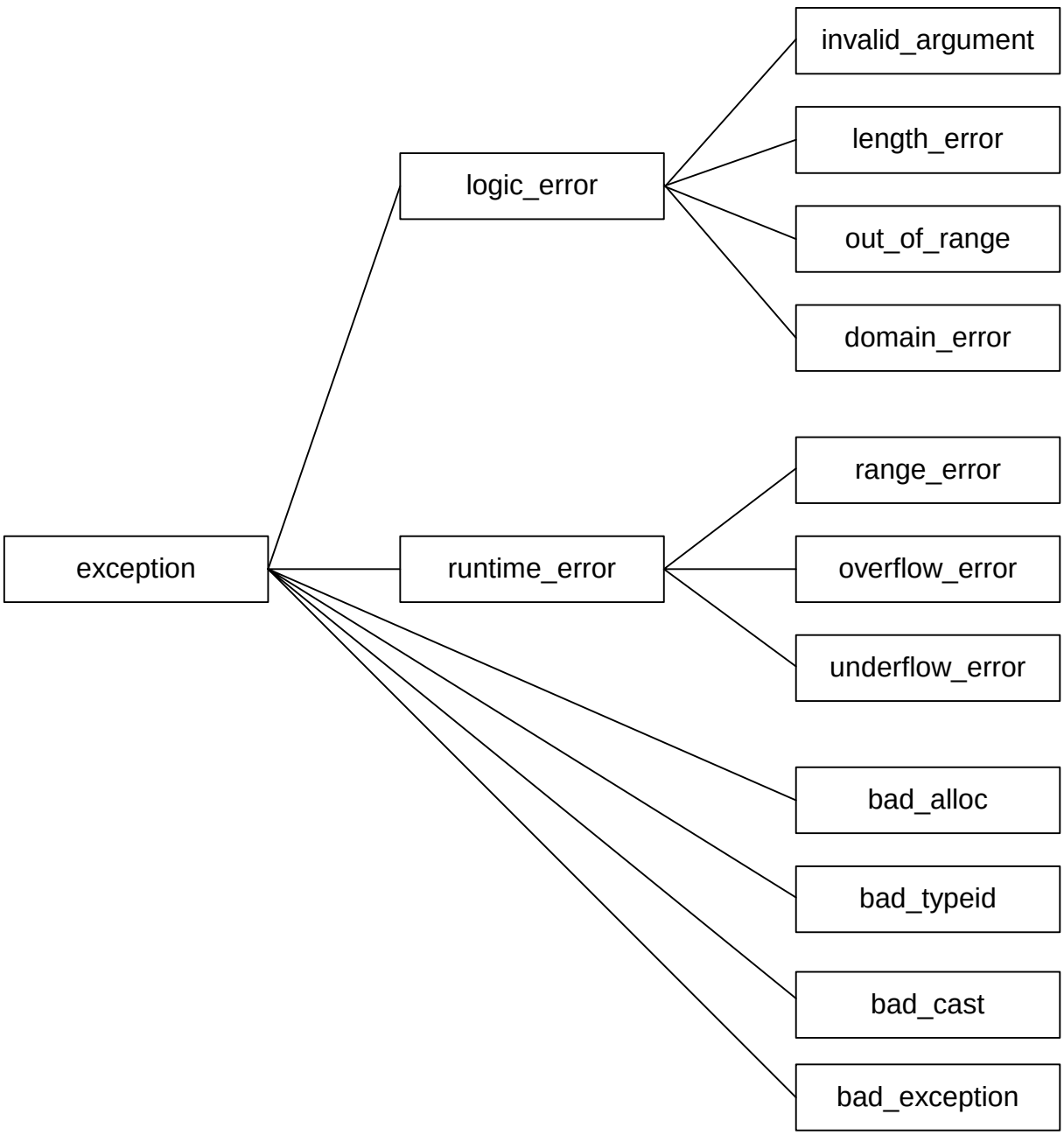
1. `void func();`
   kann beliebige Ausnahmen auswerfen.
2. `void func()` **`throw();`**
   verspricht, keine Ausnahmen auszuwerfen.
3. `void func()` **`throw(DateiEnde, const char*)`**`;`
   verspricht, nur die deklarierten Ausnahmen auszuwerfen.

Beispiele:

```
void open_file() throw(IOException)
{  ..  }

void myfunc(int arg)
  throw(int, char*, invalid_argument&)
{  ..  }
```

# Vordefinierte Ausnahmeklassen

C++ stellt eine Reihe vordefinierter Ausnahmeklassen zur Verfügung. Dabei erben alle speziellen Ausnahmeklassen von der Basisklasse **`exception`**.



# Vordefinierte Ausnahmeklassen

Die Klasse `exception` hat die folgende öffentliche Schnittstelle:

```
class exception {
  public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&)
      throw();
    virtual ~exception () throw();
    virtual const char* what () const throw();
  private:
    // ..
};
```

`throw()` nach den Deklarationen bedeutet, dass Methoden der Klasse selbst keine Ausnahmen werfen, weil es sonst eine unendliche Folge von Ausnahmen geben könnte.

Die Methode `what()` gibt einen Zeiger auf `char` zurück, der auf eine Fehlermeldung verweist.

# Vordefinierte Ausnahmeklassen

Die Klasse `logic_error` übernimmt durch Vererbung die Schnittstelle der Klasse `exception`:

```
class logic_error : public exception
{
  public:
    logic_error (const string& what_arg) throw()
      : _what(what_arg) {}
    virtual ~logic_error () throw;
    virtual const char * what () const throw()
    {
      return _what.c_str();
    }
  private:
    string _what;
};
```

# Vordefinierte Ausnahmeklassen

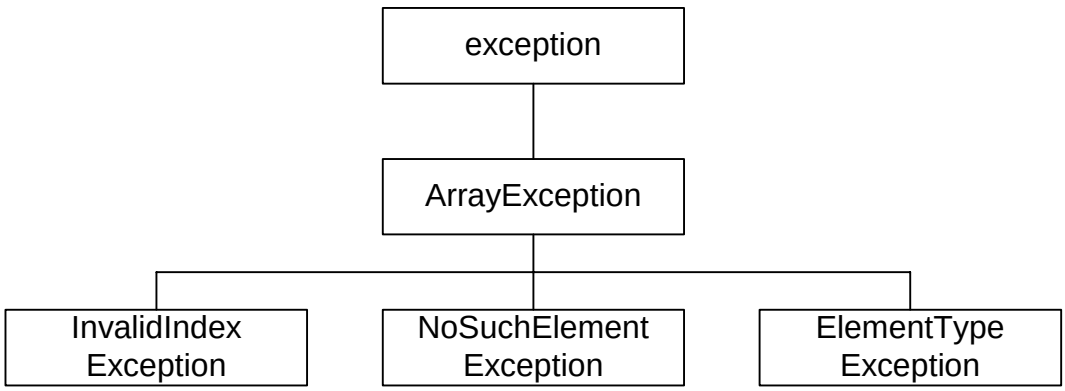
| Klasse | Bedeutung | Header |
|---|---|---|
| `exception` | Basisklasse | `<exception>` |
| `logic_error` | theoretisch vermeidbare Fehler, die noch vor der Ausführung des Programms entdeckt werden können, z.B. Verletzung von logischen Vorbedingungen | `<stdexcept>` |
| `invalid_argument` | Fehler in Funktionen der STL bei ungültigen Argumenten | `<stdexcept>` |
| `length_error` | Fehler in Funktionen der STL, wenn ein Objekt erzeugt werden soll, das die maximale Größe für dieses Objekt überschreitet | `<stdexcept>` |
| `out_of_range` | Bereichsüberschreitungsfehler in Funktionen der STL | `<stdexcept>` |
| `domain_error` | anderer Fehler des Anwendungsbereichs | `<stdexcept>` |
| `runtime_error` | nicht vorhersehbare Fehler, deren Gründe außerhalb des Programms liegen, z.B. datenabhängige Fehler | `<stdexcept>` |
| `range_error` | Bereichsüberschreitung zur Laufzeit | `<stdexcept>` |
| `overflow_error` | arithmetischer Überlauf | `<stdexcept>` |
| `underflow_error` | arithmetischer Unterlauf | `<stdexcept>` |
| `bad_alloc` | Speicherzuweisungsfehler | `<new>` |
| `bad_typeid` | falscher Objekttyp | `<typeinfo>` |
| `bad_cast` | Typumwandlungsfehler | `<typeinfo>` |
| `bad_exception` | Verletzung der Exception-Spezifikation | `<exception>` |

# Eigene Ausnahmeklassen

Gruppieren und Differenzieren von Fehlertypen durch hierarchische Anordnung von Ausnahmeklassen

Beispiel:

Fehler, die beim Zugriff auf ein Feld auftreten können




Abfangen von Ausnahmen wegen ungültigem Index:

```
catch (InvalidIndexException iie) { .. }
```

Abfangen *aller* Feldausnahmen:

```
catch (ArrayException ae) { .. }
```

Eigene Ausnahmeklassen können durch Vererbung die Schnittstelle der Klasse `exception` übernehmen (so wie z.B. die Klasse `logic_error`):

```
#include <exception>
class ArrayException : public exception
{ .. };
```

# Besondere Fehlerbehandlungsfunktionen

Es kann sein, dass in einer Fehlerbehandlung selbst wieder Fehler auftreten. Die dann aufgerufenen Funktionen werden in diesem Abschnitt beschrieben.

`terminate()`

Die Funktion `void terminate()` wird u.a. aufgerufen, wenn

- der Ausnahme-Mechanismus keine Möglichkeit zur Bearbeitung einer geworfenen Ausnahme findet;
- der Destruktor während des Aufräumens eine Ausnahme wirft oder wenn
- ein statisches (nicht lokales) Objekt während der Konstruktion oder Zerstörung eine Ausnahme wirft.

Die Standard-Implementierung von `terminate()` beendet das Programm durch Aufruf der Funktion `abort()`.

`unexpected()`

Die Funktion `void unexpected()` wird aufgerufen, wenn in einer Funktion eine Ausnahme geworfen wird, deren Typ in der Ausnahme-Spezifikation (`throw`-Liste) nicht aufgeführt wird.

Die Standard-Implementierung von `unexpected()` ruft die Funktion `terminate()` auf.

# Besondere Fehlerbehandlungsfunktionen

Die Funktionen `terminate()` und `unexpected()` können bei Bedarf selbst definiert werden, um die vorgegebenen zu ersetzen. Dazu sind standardmäßig zwei Typen für Funktionszeiger definiert:

```
typedef void (*terminate_handler)();
typedef void (*unexpected_handler)();
```

Dazu passend gibt es zwei Funktionen, denen die Zeiger auf die selbst definierten Funktionen dieses Typs übergeben werden, um die vorgegebenen zu ersetzen.

```
terminate_handler set_terminate
  (terminate_handler f) throw();
unexpected_handler set_unexpected
  (unexpected_handler f) throw();
```

Beispiel:

```
#include <exception>

void feierabend()
{
  terminate();     // hier wie Voreinstellung
}

set_terminate(feierabend);
```

# Durch Ausnahmen verursachte Speicherlecks vermeiden

In C++ kann ein Problem auftreten, wenn Speicher in einem Block dynamisch zugewiesen wird:

```
void func()
{
  Datum heute;
  Datum *pD = new Datum;
  f(heute);
  f(*pD);
  delete pD;
}
```

Wenn die Funktion `f()` eine Ausnahme auswirft, wird der Destruktor der Klasse `Datum` für das Objekt `heute` aufgerufen. Das Objekt, auf das `pD` verweist, wird jedoch nicht freigegeben, weil `delete` nicht mehr erreicht wird und `pD` außerhalb des Blocks unbekannt ist.

```
int main()
{
  try
  {
    func();
  }
  catch(...)
  {
    // pD ist hier unbekannt
  }
}
```

# Durch Ausnahmen verursachte Speicherlecks vermeiden

Abhilfe können wir z.B. dadurch schaffen, dass wir `func()` um eine Ausnahmebehandlung erweitern:

```
void func()
{
  Datum *pD;
  try
  {
    Datum heute;
    pD = new Datum;
    f(heute);
    f(*pD);
    delete pD;
  }
  catch(...)    // alle Fehler auffangen
  {
    delete pD;  // Speicher freigeben
    throw;      // Ausnahme-Objekt an den
                // Aufrufer weitergeben
  }
}
```

# Durch Ausnahmen verursachte Speicherlecks vermeiden

Besser ist die Verwendung der sogenannten `auto_ptr` aus der C++ Standardbibliothek.

Prinzip: Ersetzung von Zeigern durch automatisch angelegte Objekte, die sich wie Zeiger verhalten.

```
void func()
{
  Datum heute;
  auto_ptr<Datum> pD(new Datum);
  f(heute);
  f(*pD);
}
```

Beispiel für eine - unvollständige - Definition der Template-Klasse `auto_ptr`:

```
template <class T> class auto_ptr
{
  public:
    auto_ptr(T *p = 0) : ptr(p) {}
    ~auto_ptr() { delete ptr; }
  private:
    T *ptr;
}
```

# Speicherbeschaffung mit new

Bei Fehlschlagen der Speicherbeschaffung mit new wird eine Ausnahme ausgeworfen, nämlich ein Objekt vom Typ bad_alloc.

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

int main()
{
  char *cp;
  long total_allocated = 0;

  try
  {
    while(true)
    {
      cp = new char [1000000];
      total_allocated += 1000000;
      cout << total_allocated
           << " Byte beschafft." << endl;
    }
  }
  catch(bad_alloc & X)
  {
    cerr << total_allocated
         << " Byte beschafft." << endl;
    cerr << "bad_alloc ausgeworfen! Grund: "
         << X.what() << endl;
  }

  return 0;
}
```

# Zusammenfassung

Eine Ausnahme ist ein Signal, das ein außergewöhnliches Ereignis anzeigt.

- Durch `throw` wird ein solches Ereignis signalisiert:

```
throw MyException();
```

- Durch `catch` wird eine Ausnahme behandelt:
  `try`/`catch`-Anweisung

```
try { .. }
catch (MyException m) { .. }
catch (AnotherException) { .. }
```

- Funktionen können eventuelle Ausnahmen behandeln oder an den Aufrufer weiterreichen.
  Der Typ unbehandelter Ausnahmen *kann* in der Schnittstelle der Funktion spezifiziert werden:

```
void myfunc(int arg)
  throw(MyException1, MyException2) { .. }
```

- Die Definition eigener Ausnahmeklassen ist möglich:

```
class MyException : public exception {
  public : MyException() : exception() {}
};
```

# Zusammenfassung

Die Ausnahmebehandlung in C++ erlaubt eine klare Trennung der Programmteile für den Normalfall und den Ausnahmefall.

Die Ausnahmebehandlung ermöglicht die

- **Erkennung** und die
- **Behandlung**

von Fehlern.

Die Fehlererkennung ist in der Regel einfach, die Behandlung schwierig und häufig genug unmöglich.

Die Art der Fehlerbehandlung hängt auch davon ab, wie sicherheitskritisch der Einsatz der Software ist.

Beispiel: Die Fehlermeldung

```
Index-Fehler im Array controlParameters,
Index = 2198, Max = 84
```

- in einem Textverarbeitungsprogramm
- in der Software eines in der Luft befindlichen Flugzeugs