OutOfMemory.cpp

```
#include <iostream>
#include <stdexcept>

int main() {
   try {
      int count = 0;
      for(;;) {
         char* megabyte = new char[1048576]; *megabyte = 0;
         std::cout << " " << ++count << std::flush;
      }
   } catch(std::bad_alloc) {
      std::cout << " ... Game over!" << std::endl;
   }
} // main
```

- Ausnahmenbehandlungen sind eine mächtige (und recht aufwendige!) Kontrollstruktur zur Behandlung von Fehlern.

```
theon$ ulimit -d 8192  # limits max size of heap (in kb)
theon$ ./OutOfMemory
 1 2 3 4 5 6 7 ... Game over!
theon$
```

- Ausnahmenbehandlungen erlauben das Schreiben robuster Software, die wohldefiniert im Falle von Fehlern reagiert.

Crash.cpp

```
#include <iostream>

int main() {
   int count = 0;
   for(;;) {
      char* megabyte = new char[1048576]; *megabyte = 0;
      std::cout << " " << ++count << std::flush;
   }
} // main
```

- Ausnahmen, die nicht abgefangen werden, führen zum Aufruf von *std*::*terminate*(), das voreinstellungsgemäß *abort()* aufruft.
- Unter UNIX führt *abort()* zu einer Terminierung des Prozesses mitsamt einem Core-Dump.

```
theon$ ulimit -d 8192
theon$ ./crash 2>&1 | fold -w 60
 1 2 3 4 5 6 7terminate called after throwing an instance of
 'std::bad_alloc'
  what():  std::bad_alloc
theon$
```

- Dies ist akzeptabel für kleine Programme oder Tests. Viele Anwendungen benötigen jedoch eine robustere Behandlung von Fehlern.

Ausnahmen können als Verletzungen von Verträgen zwischen Klienten und Implementierungen im Falle von Methodenaufrufen betrachtet werden, wo

- ein Klient all die Vorbedingungen zu erfüllen hat und umgekehrt
- die Implementierung die Nachbedingung zu erfüllen hat (falls die Vorbedingung tatsächlich erfüllt gewesen ist).

Es gibt jedoch Fälle, bei denen eine der beiden Seiten den Vertrag nicht halten kann.

Gegeben sei das Beispiel einer Matrixinvertierung:

- Vorbedingung: Die Eingabe-Matrix ist regulär.
- Nachbedingung: Die Ausgabe-Matrix ist die invertierte Matrix der Eingabe-Matrix.

Problem: Wie kann festgestellt werden, dass eine Matrix regulär ist? Dies ist in manchen Fällen fast so aufwendig wie die Invertierung selbst.

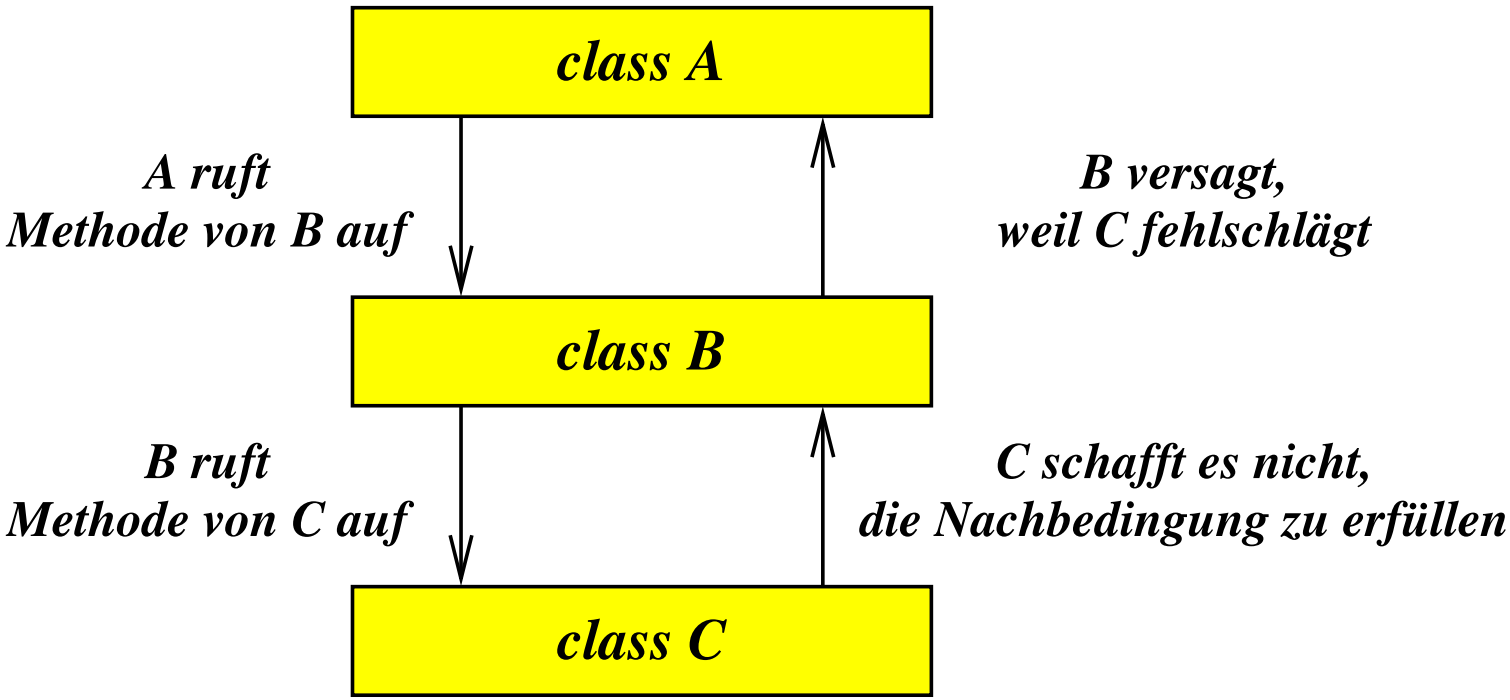
Beispiel: Übersetzer für C++:

- Vorbedingung: Die Eingabedatei ist ein wohldefiniertes Programm in C++.
- Nachbedingung: Die Ausgabedatei enthält eine korrekte Übersetzung des Programms in eine Maschinensprache.

Problem: Wie kann im Voraus sichergestellt werden, dass die Eingabedatei wohldefiniert für C++ ist?

Die Einhaltung der Nachbedingungen kann aus vielerlei Gründen versagt bleiben:

- Laufzeitfehler:
  - Programmierfehler wie z.B. ein Index, der außerhalb des zulässigen Bereiches liegt.
  - Arithmetische Fehler wie Überläufe oder das Teilen durch 0.
- Ausfälle der Systemumgebung wie etwa zu wenig Hauptspeicher, unzureichender Plattenplatz, Hardware-Probleme und unterbrochene Netzwerkverbindungen.


class A
A ruft
Methode von B auf
B versagt,
weil C fehlschlägt
class B
B ruft
Methode von C auf
C schafft es nicht,
die Nachbedingung zu erfüllen
class C

- Eine Software-Komponente ist **robust**, wenn sie nicht nur korrekt ist (d.h. die Nachbedingung wird eingehalten, wenn die Verbedingung erfüllt ist), sondern sie auch Verletzungen der Vorbedingung erkennen und signalisieren kann. Ferner sollte eine robuste Software-Komponente in der Lage sein, alle anderen Probleme zu erkennen und zu signalisieren, die sie daran hindern, die Nachbedingung zu erfüllen.
- Solche Verletzungen oder Nichterfüllungen werden **Ausnahmen** (*exceptions*) genannt.
- Die Signalisierung einer Ausnahme ist so zu verstehen: »Verzeihung, ich muss aufgeben, weil ich dieses Problem nicht selbst weiter lösen kann.«

- Wer ist für die Behandlung einer Ausnahme verantwortlich?
- Welche Informationen sind hierfür weiterzuleiten?
- Welche Optionen stehen einem Ausnahmenbehandler zur Verfügung?

- Es gibt hierfür eine Vielzahl an Konzepten, den zuständigen Ausnahmenbehandler (*exception handler*) zu lokalisieren. Dies hängt jeweils von der Programmiersprache bzw. der verwendeten Bibliothek ab.
- Es wird vielfach gerne gesehen, wenn die Ausnahmenbehandlung vom normalen Programmtext getrennt werden kann, damit der Programmtext nicht mit Überprüfungen nach jedem Methodenaufruf unübersichtlich wird.
- In C++ (und ebenso nicht wenigen anderen Programmiersprachen) liegt die Verantwortung beim Klienten. Wenn kein zuständiger Ausnahmenbehandler definiert ist, dann wird die Ausnahme automatisch durch die Aufrufkette weitergeleitet und dabei der Stack abgebaut. Wenn am Ende nirgends ein Ausnahmenbehandler gefunden wird, terminiert der Prozess mit einem Core-Dump.
- Alternativ gibt es den Ansatz, Ausnahmenbehandler für Objekte zu definieren. Dies ist auch bei C++ möglich, wird aber nicht direkt von der Sprache unterstützt.

Wie können Informationen über eine Ausnahme weitergeleitet werden?

VerboseOutOfMemory.cpp

```
#include <iostream>
#include <stdexcept>

int main() {
   try {
      int count = 0;
      for(;;) {
         char* megabyte = new char[1048576]; *megabyte = 0;
         std::cout << " " << ++count << std::flush;
      }
   } catch(std::bad_alloc& e) {
      std::cout << " ... Game over!" << std::endl;
      std::cout << "This hit me: " << e.what() << std::endl;
   }
} // main
```

- C++ hat einen recht einfachen und gleichzeitig mächtigen Ansatz: Beliebige Instanzen einer Klasse können verwendet werden, um das Problem zu beschreiben.

Wie können Informationen über eine Ausnahme weitergeleitet werden?


```
theon$ ulimit -d 8192
theon$ ./VerboseOutOfMemory
 1 2 3 4 5 6 7 ... Game over!
This hit me: std::bad_alloc
theon$
```

- Alle Ausnahmen, die von der ISO-C++-Standardbibliothek ausgelöst werden, verwenden Erweiterungen der **class** *exception*, die eine virtuelle Methode *what()* anbietet, die eine Zeichenkette für Fehlermeldungen liefert.

exception

```
namespace std {

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

- Hier ist zu beachten, dass Klassen, die für Ausnahmen verwendet werden, einen Kopierkonstruktor anbieten müssen, da dieser implizit bei der Ausnahmenbehandlung verwendet wird.
- Die Signatur einer Funktion oder Methode kann spezifizieren, welche Ausnahmen ausgelöst werden können. **noexcept** bedeutet, dass keinerlei Ausnahmen ausgelöst werden.

⟨noexcept-specifier⟩ ⟶ **noexcept** „(“ ⟨constant-expression⟩ „)“
⟶ **noexcept**
⟶ **throw** „(“ „)“

- Früher war mit **throw** eine Aufzählung der denkbaren Ausnahmen möglich – dies wurde inzwischen abgeschafft.
- Mit **noexcept** wird zugesichert, dass keine Ausnahme auftritt. Dies lässt Optimierungen zu.
- Wenn ein ⟨constant-expression⟩ angegeben wird, dann hängt dies von dem zur Übersetzzeit erfolgten Bewertung des Ausdrucks ab: **true** entspricht der Zusicherung, **false** heißt, dass die Zusicherung nicht gegeben ist.
- Wenn kein ⟨noexcept-specifier⟩ gegeben ist, entspricht dies **noexcept**(**false**).
- Wenn trotz einer **noexcept**-Zusicherung eine Methode oder Funktion direkt oder indirekt Ausnahmen auslöst, wird *std*::*unexpected* bzw. *std*::*terminate* aufgerufen.

ArrayedStack.hpp

```
template<class Item, std::size_t SIZE = 4>
class ArrayedStack {
   public:
      ArrayedStack() noexcept : index(0) {};

      bool empty() const noexcept { return index == 0; }
      bool full() const noexcept { return index == SIZE; }
      const Item& top() const { /* ... */ }
      void push(const Item& item) { /* ... */ }
      void pop() { /* ... */ }

   private:
      std::size_t index;
      Item items[SIZE];
}; // class ArrayedStack
```

- Wenn zugesichert werden kann, das keine Ausnahmen direkt oder indirekt ausgelöst werden, dann sollte **noexcept** in die Signatur aufgenommen werden.

Definition von Ausnahmen

StackExceptions.hpp

```
#include <exception>

class StackException : public std::exception {};

class FullStack : public StackException {
   public:
      virtual const char* what() const noexcept {
         return "stack is full";
      };
}; // class FullStack

class EmptyStack : public StackException {
   public:
      virtual const char* what() const noexcept {
         return "stack is empty";
      };
}; // class EmptyStack
```

- Klassen für Ausnahmen sollten hierarchisch organisiert werden.
- Eine **catch**-Anweisung für *StackException* erlaubt das Abfangen der Ausnahmen *FullStack*, *EmptyStack* und aller anderen Erweiterungen von *StackException*.

ArrayedStack.hpp

```
const Item& top() const {
   // PRE: not empty()
   if (index > 0) {
      return items[index-1];
   } else {
      throw EmptyStack();
   }
}

void push(const Item& item) {
   // PRE: not full()
   if (index < SIZE) {
      items[index] = item;
      index += 1;
   } else {
      throw FullStack();
   }
}
```

⟨throw-expression⟩ ⟶ **throw** [⟨assignment-expression⟩]

- Ein ⟨throw-expression⟩ löst eine Ausnahmenbehandlung aus, die durch das angebene Objekt repräsentiert wird, das das Problem beschreiben sollte.
- Innerhalb einer Ausnahmenbehandlung ist auch ein ⟨throw-expression⟩ ohne einen Ausdruck sinnvoll. In diesem Fall wird die Ausnahme an den Aufrufer weitergeleitet.
- Es ist hierbei erlaubt, temporäre Objekte zu verwenden, da diese bei Bedarf implizit kopiert werden.

- Nach dem Auflösen werden entsprechend der Aufrufverschachtelung sukzessive Blöcke abgebaut, bis ein passende Ausnahmenbehandlung gefunden wird.
- Bei jedem abgebauten Block werden alle zugehörigen Variablen dekonstruiert.
- Wird keine passende Ausnahmenbehandlung gefunden, wird *std::terminate* aufgerufen.

⟨statement⟩ ⟶ [⟨attribute-specifier-seq⟩] ⟨try-block⟩

⟨try-block⟩ ⟶ **try** ⟨compound-statement⟩ ⟨handler-seq⟩

⟨handler-seq⟩ ⟶ ⟨handler⟩ [⟨handler-seq⟩]

⟨handler⟩ ⟶ **catch** „(“ ⟨exception-declaration⟩ „)“ ⟨compound-statement⟩

⟨exception-declaration⟩ ⟶ [⟨attribute-specifier-seq⟩] ⟨type-specifier-seq⟩ ⟨declarator⟩

⟶ [⟨attribute-specifier-seq⟩] ⟨type-specifier-seq⟩ [⟨abstract-declarator⟩]

⟶ „...“

⟨function-body⟩ ⟶ [⟨ctor-initializer⟩] ⟨compound-statement⟩
⟶ ⟨function-try-block⟩
⟶ „=“ **default** „;“
⟶ „=“ **delete** „;“
⟨function-try-block⟩ ⟶ **try** [⟨ctor-initializer⟩]
⟨compound-statement⟩ ⟨handler-seq⟩

- Auch innerhalb eines ⟨ctor-initializer⟩ bei einem Konstruktor kann es zum Auslösen von Ausnahmen kommen.
- In diesem Falle käme eine Ausnahmenbehandlung innerhalb des regulären ⟨compound-statement⟩ zu spät.
- Daher kann dies beides durch einen ⟨function-try-block⟩ ersetzt werden.

LeakingObject.cpp

```
struct A {
   A(int i) : i(i) {
      if (i < 0) {
         throw i;
      }
   }
   int i;
};
```

- Prinzipiell können Konstruktoren auch Ausnahmen auslösen und das kann auch sehr sinnvoll sein, da kaum andere Wege der Fehlerbehandlung zur Verfügung stehen. Das erscheint vorteilhafter als der Umgang mit „Zombie-Objekten“, die scheinbar fertig konstruiert sind, aber die gewünschte Funktionalität nicht erfüllen.

LeakingObject.cpp

```
struct B {
   B(int i, int j) : p1(new A(i)), p2(new A(j)) { }
   ~B() { delete p1; delete p2; }
   A* p1; A* p2;
};
```

- Was passiert, wenn ein Konstruktor von einer Ausnahme betroffen ist?
  - Falls es bei **new** passierte, wird der Speicher umgehend freigegeben.
  - Dann werden alle bereits konstruierten Unterobjekte in umgekehrter Reihenfolge abgebaut.
  - Der *destructor* wird nie aufgerufen.
- Was passiert, wenn **new** $A(j)$ schiefgeht? Dann wird der hierfür belegte Speicher freigegeben und der Zeiger *p1* abgebaut. Das Abbauen eines Zeigers ist aber nicht mit einer Aktion verbunden. Wir hätten also ein Speicherleck.
- Deswegen sollte innerhalb eines ⟨ctor-initializer⟩ nie mehr als ein **new** vorkommen, um Lecks zu vermeiden.

CatchLeakingObject.cpp

```
struct B {
   B(int i, int j) try : p1(new A(i)), p2(new A(j)) {
   } catch(int val) {
      // what are our options here?
   }
   ~B() {
      delete p1; delete p2;
   }
   A* p1; A* p2;
};
```

- Gab es da nicht die Option, so etwas abzufangen?
- Ja, aber das ändert nichts daran, dass das Objekt nach dem unterbrochenen Abarbeiten des ⟨ctor-initializer⟩ nicht mehr erfolgreich konstruiert werden kann.
- Das bedeutet, dass es in jedem Fall mit einer Ausnahme weitergeht. Entweder die gleiche oder eine andere.
- Somit ist das nur in Ausnahmesituationen sinnvoll, wenn etwa eine zusätzliche Aktion für den korrekten Abbau notwendig ist.

CatchLeakingObject.cpp

```
struct B {
   B(int i, int j) try : p1((++count, new A(i))), p2((++count, new A(j))) {
   } catch(int val) {
      if (count == 2) delete p1;
   }
   ~B() {
      delete p1; delete p2;
   }
   int count = 0;
   A* p1; A* p2;
};
```

- Prinzipiell wäre es denkbar, fehlende Abbau-Aktionen wie etwa die fehlende Freigabe nachzuholen.
- Erstrebenswert oder lesbar ist das nicht.

Calculator.hpp

```
template<typename Value, typename Stack>
class Calculator {
   public:
      class Exception : public std::exception {};
      // ...

      Value calculate(const std::string& expr) {
         // PRE: expr in RPN (reversed polish notation) syntax
         // ...
      }

   private:
      Stack opstack;
}; // class Calculator
```

- Diese Klasse bietet einen Rechner an, der Ausdrücke in der umgekehrten polnischen Notation (UPN) akzeptiert.
- Beispiele für gültige Ausdrücke: „1 2 +“, „1 2 3 * +“.
- UPN-Rechner können recht einfach mit Hilfe eines Stacks implementiert werden.

Ausnahmen des Taschenrechners

Calculator.hpp

```
template<typename Value, typename Stack>
class Calculator {
   public:
      class Exception : public std::exception {};
      class SyntaxError : public Exception {
         public:
            virtual const char* what() const noexcept {
               return "syntax error";
            };
      }; // class SyntaxError
      class BadExpr : public Exception {
         public:
            virtual const char* what() const noexcept {
               return "invalid expression";
            };
      }; // class BadExpr
      class StackFailure : public Exception {
         public:
            virtual const char* what() const noexcept {
               return "stack failure";
            };
      }; // class StackFailure
   // ...
}; // class Calculator
```

- Die Ausnahmen für *Calculator* sollten entsprechend der Abstraktionsebene dieser Klasse verständlich sein.
- Aus diesem Grunde wird hier die Ausnahme *StackFailure* hinzugefügt, die für den Fall vorgesehen ist, dass der zur Verfügung stehende Stack seine Aufgabe (z.B. wegen mangelnder Kapazität) nicht erfüllt.

Calculator.hpp

```
Value calculate(const std::string& expr) {
   // PRE: expr in RPN (reversed polish notation) syntax
   std::istringstream in(expr);
   Value result; // return value
   try {
      std::string token;
      while (in >> token) {
         // ...
      }
      result = opstack.top(); opstack.pop();
      if (!opstack.empty()) {
         throw BadExpr();
      }
   } catch(FullStack) {
      throw StackFailure();
   } catch(EmptyStack) {
      throw BadExpr();
   }
   return result;
}
```

- Zu beachten ist hier, wie Ausnahmen der *Stack*-Klasse in solche der *Calculator*-Klasse konvertiert werden.

Calculator.hpp

```
while (in >> token) {
   if (token == "+" || token == "-" ||
         token == "*" || token == "/") {
      Value op2{opstack.top()}; opstack.pop();
      Value op1{opstack.top()}; opstack.pop();
      Value result;
      if (token == "+") { result = op1 + op2;
      } else if (token == "-") { result = op1 - op2;
      } else if (token == "*") { result = op1 * op2;
      } else { result = op1 / op2;
      }
      opstack.push(result);
   } else {
      std::istringstream vin{token};
      Value value;
      if (vin >> value) {
         opstack.push(value);
      } else {
         throw SyntaxError();
      }
   }
}
result = opstack.top(); opstack.pop();
```

TestCalculator.cpp

```
int main() {
   std::string expr;
   while (std::cout << ": " && std::getline(std::cin, expr)) {
      try {
         Calculator<double, ArrayedStack<double>> calc;
         std::cout << calc.calculate(expr) << std::endl;
      } catch(std::exception& exc) {
         std::cerr << exc.what() << std::endl;
      }
   }
} // main
```

- Ausnahmen werden hier innerhalb der **while**-Schleife abgefangen, so dass ein Weiterarbeiten nach einem Fehler möglich ist.

```
theon$ ./TestCalculator
: 1 2 +
3
: 1 2 3 * +
7
: 1 2 3 4 5 + + + +
stack failure
: 1
1
: 1 2
invalid expression
: +
invalid expression
: x
syntax error
: theon$
```

- Zu beachten ist hier, dass die Implementierung des *ArrayedStack* nur vier Elemente unterstützt.
- „1 2“ ist unzulässig, da der Stack am Ende nach dem Entfernen des obersten Elements nicht leer ist.

Ausnahmenbehandlungen brechen Abstraktionsgrenzen und können eine regelrechte Verwüstung hinterlassen, da ganze Ketten von Funktionsaufrufen abgeräumt werden können. Je nach Umfang des Schutzes lassen sich verschiedene Grade voneinander unterscheiden:

- Gar kein Schutz.
- Elementarer Schutz gegen Speicherlecks und das Hinterlassen offener Ressourcen. Dieser Schutz basiert auf der konsequenten Anwendung von RAII-Objekten.
- Transaktionsbasierter Schutz: Entweder ist die Operation erfolgreich oder sie schlägt fehl mit einer Ausnahmenbehandlung. Im letzteren Fall bleibt der Stand vor dem Aufruf der Operation erhalten.
- Zusicherung von **noexcept**.

- Bereits 2000 propagierten Andrei Alexandrescu und Petru Marginean in Ihrem Paper die Idee, dass spezielle RAII-Objekte dazu genutzt werden könnten, um ein *rollback* im Falle einer abgebrochenen Transaktion durchzuführen.
- Die speziellen RAII-Objekte wurden *scope guards* genannt, die die Aufgabe hatten, die Rollback-Aktionen zu übernehmen, wenn die Transaktion abgebrochen wird.
- Die Realisierung ist recht einfach: Ein *scope guard* führt den Rollback beim Abbau aus, es sei denn, es wurde zuvor die Methode *commit* aufgerufen.

```
template<typename T>
bool add(Database<T>& db1, Database<T>& db2, T object) {
   try {
      db1.add(object);
      auto guard = make_guard([&]() { db1.remove(object); });
      db2.add(object);
      guard.commit();
      return true;
   } catch (...) {
      return false;
   }
}
```

- Die Funktion *add* soll transaktionsbasiert ein Objekt in zwei Datenbanken einfügen.
- Entsprechend der Transaktion soll entweder beide Operationen durchgeführt werden oder keine davon.
- Die Funktion *make_guard* erzeugt hier einen *scope guard*, der den angegebenen Lambda-Ausdruck ausführt, wenn nicht zuvor *commit* aufgerufen wurde.

Einfache Implementierung eines *scope guards* 355

ScopeGuard.hpp

```
template<typename T>
class ScopeGuard {
   public:
      ScopeGuard(T rollback) : rollback(std::move(rollback)) {
      }
      ~ScopeGuard() {
         if (!committed) rollback();
      }
      void commit() {
         committed = true;
      }
   private:
      T rollback;
      bool committed = false;
};
```

- Der Template-Typparameter T ist der des Funktionsobjekts, der beim Abbau aufzurufen ist, wenn nicht zuvor *commit* aufgerufen wurde.
- In dieser Form ist es unhandlich zu benutzen. Deswegen kommt noch eine Template-Funktion *make_guard* hinzu...

ScopeGuard.hpp

```
#include <utility>

/* ... */

template<typename T>
ScopeGuard<T> make_guard(T&& rollback) {
   return ScopeGuard<T>(std::forward<T>(rollback));
}
```

- Um auf das unhandliche Spezifizieren des Typparameters verzichten zu können, hilft es, eine Template-Funktion hinzuzufügen, die den Typ automatisch ableitet.
- Wenn *T&&* für einen Template-Typparameter *T* spezifiziert wird, werden automatisch beide Fälle unterstützt: *lvalue reference* und *rvalue reference*.
- Um einen Parameter bedingt per *std*::*move* weiterzureichen, falls es sich um eine *rvalue reference* handelte, gibt es *std*::*forward*. Diese Technik nennt sich *perfect forwarding*, d.h. entweder geben wir die *lvalue reference* weiter oder Verschieben es mit *std*::*move*.

ScopeGuard.hpp

```
template<typename T>
class ScopeGuard {
   public:
      ScopeGuard(T rollback) : rollback(std::move(rollback)) {
      }
      ~ScopeGuard() {
         if (std::uncaught_exception()) rollback();
      }
   private:
      T rollback;
};
```

- Lässt sich feststellen, ob der Stack gerade im Rahmen einer Ausnahmenbehandlung abgebaut wird?
- Ja, das geht mit der Funktion *std::uncaught_exception*, die mit C++11 eingeführt wurde.
- Idee: Auf *commit* verzichten und *rollback* nur ausführen, wenn gerade eine Ausnahmenbehandlung läuft.

Bessere Lösung (ab C++17)

ScopeGuard.hpp

```
template<typename T>
class ScopeGuard {
   public:
      ScopeGuard(T rollback) :
         rollback(std::move(rollback)),
         exceptions(std::uncaught_exceptions()) {
      }
      ~ScopeGuard() {
         if (std::uncaught_exceptions() > exceptions) rollback();
      }
   private:
      T rollback;
      int exceptions;
};
```

- Problem: Auch während einer laufenden Ausnahmenbehandlung können Objekte (wie *scope guards*) regulär angelegt und abgebaut werden.
- Beginnend ab C++17 gibt es *std::uncaught_exceptions*, die die Zahl der gerade laufenden Stack-Abbauten liefert.
- Jetzt wird *rollback* nur dann aufgerufen, wenn der *scope guard* in Folge einer Ausnahmenbehandlung abgebaut wird.