OutOfMemory.cpp

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

int main() {
   try {
      int count = 0;
      for(;;) {
         char* megabyte = new char[1048576];
         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.

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

- 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];
      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.

```
thales$ ulimit -d 8192
thales$ ./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
thales$
```

- 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.

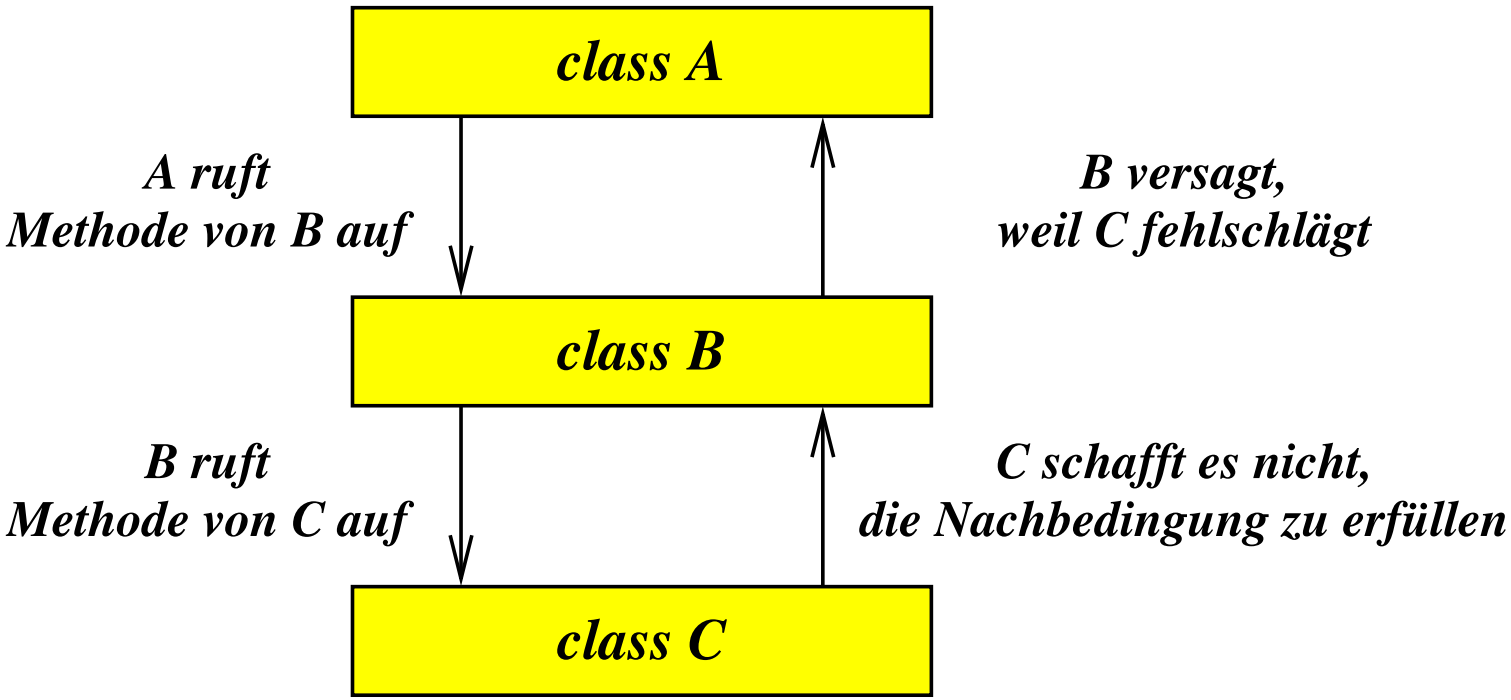
Beispiel: Übersetzer für C++:

- 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];
         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?


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

- 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.

⟨exception-specification⟩ ⟶ ⟨dynamic-exception-specification⟩
⟶ ⟨noexcept-specification⟩
⟨dynamic-exception-specification⟩ ⟶ **throw** „(“ [⟨type-id-ist⟩] „)“
⟨type-id-list⟩ ⟶ ⟨type-id⟩ [„...“]
⟶ ⟨type-id-list⟩ „,“ ⟨type-id⟩ [„...“]
⟨noexcept-specification⟩ ⟶ **noexcept** „(“
⟨constant-expression⟩ „)“
⟶ **noexcept**

- Wenn keine ⟨exception-specification⟩ angegeben wird, entspricht dies **noexcept**(**false**), d.h. alle Ausnahmen sind denkbar.
- **noexcept** ist äquivalent zu **noexcept**(**true**) und bedeutet, d.h. die Methode oder Funktion weder direkt noch indirekt Ausnahmen auslöst. Geschieht es dennoch, wird *std*::*unexpected* bzw. *std*::*terminate* aufgerufen.

ArrayedStack.hpp

```
template<typename Item, int SIZE = 4>
class ArrayedStack {
   public:
      ArrayedStack() noexcept;

      bool empty() const noexcept;
      bool full() const noexcept;
      const Item& top() const throw(EmptyStack);

      void push(const Item& item) throw(FullStack);
      void pop() throw(EmptyStack);

   private:
      // ...
}; // class ArrayedStack
```

- Die Menge der potentiell ausgelösten Ausnahmen kann und sollte in eine Signatur aufgenommen werden. Wenn dies erfolgt, dürfen andere Ausnahmen von der Funktion bzw. Methode nicht ausgelöst 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.tpp

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

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

- **throw** erhält ein Objekt, das die Ausnahme repräsentiert und initiiert die Ausnahmenbehandlung.
- Das Objekt sollte das Problem beschreiben.
- Es ist hierbei erlaubt, temporäre Objekte zu verwenden, da diese bei Bedarf implizit kopiert werden.
- Zu beachten ist, dass alle lokalen Variablen einer Funktion oder Methode, die eine Ausnahme auslöst, jedoch diese nicht abfängt, vollautomatisch im Falle einer Ausnahmenbehandlung dekonstruiert werden.
- Das automatische Dekonstruieren erstreckt sich auf alle lokalen Variablen der Funktionen und Methoden in der Aufrufkette, die zwischen dem Auslöser und dem Ausnahmenbehandler stehen.

Calculator.hpp

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

      Value calculate(const std::string& expr) throw(Exception) {
         // 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.

Ausnahmenbehandlung des Taschenrechners 278

Calculator.hpp

```
Value calculate(const std::string& expr) throw(Exception) {
   // PRE: expr in RPN (reversed polish notation) syntax
   std::istringstream in(expr);
   std::string token;
   Value result;
   try {
      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.

Innenbereich der Schleife des Taschenrechners

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

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

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

- 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**.