15. C++ vertieft (IV): Ausnahmen (Exceptions)

Was kann schon schiefgehen?

- Öffnen einer Datei zum Lesen oder Schreiben

```
std::ifstream input("myfile.txt");
```

- Parsing

```
int value = std::stoi("12−8");
```

- Speicherallokation

```
std::vector<double> data(ManyMillions);
```

- Invalide Daten

```
int a = b/x; // what if x is zero?
```

Möglichkeiten der Fehlerbehandlung

- Keine (inakzeptabel)
- Globale Fehlervariable (Flags)
- Funktionen, die Fehlercodes zurückgeben
- Objekte, die Fehlercodes speichern
- Ausnahmen

Globale Fehlervariablen

- Typisch für älteren C-Code
- Nebenläufigkeit ist ein Problem
- Fehlerbehandlung nach Belieben. Erfordert grosse Disziplin, sehr gute Dokumentation und übersäht den Code mit scheinbar unzusammenhängenden Checks.

Rückgabe von Fehlercodes

- Jeder Aufruf einer Funktion wird mit Ergebnis quittiert.
- Typisch für grosse APIs (OS Level). Dort oft mit globalen Fehlercodes kombiniert.[20]
- Der Aufrufer kann den Rückgabewert einer Funktion prüfen, um die korrekte Ausführung zu überwachen.

[20] Globaler error code thread-safety durch thread local storage.

Rückgabe von Fehlercodes

Beispiel

```
#include <errno.h>
...

pf = fopen ("notexisting.txt", "r+");
if (pf == NULL) {
  fprintf(stderr, "Error opening file: %s\n", strerror( errno ));
}
else { // ...
  fclose (pf);
}
```

Fehlerstatus im Objekt

- Fehlerzustand eines Objektes intern im Objekt gespeichert.

Beispiel

```
int i;
std::cin >> i;
if (std::cin.good()){// success, continue
        ...
}
```

Exceptions

- Exceptions unterbrechen den normalen Kontrollfluss
- Exceptions können geworfen (throw) und gefangen (catch) werden
- Exceptions können über Funktionengrenzen hinweg agieren.

Beispiel: Exception werfen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main()
{
  f(4);
  return 0;
}
```

Beispiel: Exception werfen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main()
{
  f(4);
  return 0;
}
```

```
terminate called after throwing an instance of 'MyException'
Aborted
```

Beispiel: Exception fangen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main(){
  try{
    f(4);
  }
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

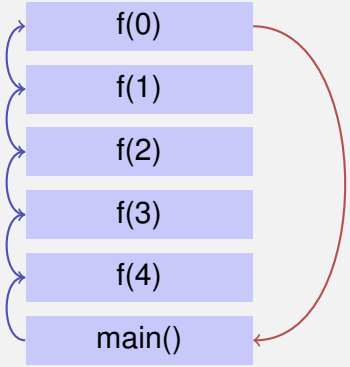
Beispiel: Exception fangen

```
class MyException{};

void f(int i){
  if (i==0) throw MyException();
  f(i-1);
}

int main(){
  try{
    f(4);
  }
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

exception caught



Resourcen werden geschlossen

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
  if (i==0) throw MyException();
  SomeResource x;
  f(i-1);
}
int main(){
  try{f(5);}
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

Resourcen werden geschlossen

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
  if (i==0) throw MyException();
  SomeResource x;
  f(i-1);
}
int main(){
  try{f(5);}
  catch (MyException e){
      std::cout << "exception caught\n";
  }
}
```

```
closed resource
closed resource
closed resource
closed resource
closed resource
exception caught
```

Wann Exceptions?

Exceptions werden für die *Behandlung von Fehlern* benutzt.

- Verwende `throw` nur, um einen Fehler zu signalisieren, welcher die Postcondition einer Funktion verletzt oder das Fortfahren des Codes unmöglich macht
- Verwende `catch` nur, wenn klar ist, wie man den Fehler behandeln kann (u.U. mit erneutem Werfen der Exception)
- Verwende `throw` *nicht* um einen Programmierfehler oder eine Verletzung von Invarianten anzuzeigen (benutze stattdessen `assert`)
- Verwende Exceptions *nicht* um den Kontrollfluss zu ändern. Throw ist *nicht* ein besseres return.

Warum Exceptions?

Das:

```
int ret = f();
if (ret == 0) {
  // ...
} else {
  // ...code that handles the error...
}
```

sieht auf den ersten Blick vielleicht besser / einfacher aus als das:

```
try {
  f();
  // ...
} catch (std::exception& e) {
  // ...code that handles the error...
}
```

Warum Exceptions?

Die Wahrheit ist, dass Einfachstbeispiele den Kern der Sache nicht immer treffen.

Return-Codes zur Fehlerbehandlung übersähen grössere Codestücke entweder mit Checks oder die Fehlerbehandlung bleibt auf der Strecke.

Darum Exceptions

Beispiel 1: Evaluation von Ausdrücken (Expression Parser, Vorlesung Informatik I), siehe
`http://codeboard.io/projects/46131`

Eingabe: `1 + (3 * 6 / (/ 7 ))`
Fehler tief in der Rekursionshierarchie. Wie kann ich eine vernünftige Fehlermeldung produzieren (und weiterfahren)? Müsste den Fehlercode über alle Rekursionsstufen zurückgeben. Das übersäht den Code mit checks.

Beispiel 2

Wertetyp mit Garantie: Werte im gegebenen Bereich.

```
template <typename T, T min, T max>
class Range{
public:
  Range(){}
  Range (const T& v) : value (v) {
    if (value < min) throw Underflow ();
    if (value > max) throw Overflow ();
  }
  operator const T& () const {return value;}
private:
  T value;
};
```

Fehlerbehandlung im Konstruktor!

Fehlertypen, hierarchisch

```
class RangeException {};
class Overflow : public RangeException {};
class Underflow : public RangeException {};
class DivisionByZero: public RangeException {};
class FormatError: public RangeException {};
```

Operatoren

```
template <typename T, T min, T max>
Range<T, min, max> operator/ (const Range<T, min, max>& a,
                              const Range<T, min, max>& b){
  if (b == 0) throw DivisionByZero();
  return T (a) * T(b);
}

template <typename T, T min, T max>
std::istream& operator >> (std::istream& is, Range<T, min, max>& a){
  T value;
  if (!(is >> value)) throw FormatError();
  a = value;
  return is;
}
```

Fehlerbehandlung im Operator!

Fehlerbehandlung (zentral)

```
Range<int,−10,10> a,b,c;
try{
  std::cin >> a;
  std::cin >> b;
  std::cin >> c;
  a = a / b + 4 ∗ (b − c);
  std::cout << a;
}
catch(FormatError& e){ std::cout << "Format error\n"; }
catch(Underflow& e){ std::cout << "Underflow\n"; }
catch(Overflow& e){ std::cout << "Overflow\n"; }
catch(DivisionByZero& e){ std::cout << "Divison By Zero\n"; }
```