Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

8. November 2019

Exceptions

Inhalt

- Exceptions
- Modifizierung und Weiterwerfen
- `noexcept`
- Weitere Exception-Themen

C++ Exceptions

- Exceptions werfen mittels `throw` Keyword
- Jeder Typ ist erlaubt als Exception
- Vordefinierte Typen in `<exception>`-Header

```
1  void RuntimeThrow() {
2    throw std::runtime_error("example");
3  }
4
5    try {
6      RuntimeThrow();
7    } catch (const std::runtime_error& e) {
8      std::cout << "std::runtime_error: " << e.what() << std::endl;
9    }
```

Best-Practice

- Exceptions nur by-value werfen (automatic lifetime)
- Exceptions als const-ref fangen
- Nur Typen abgeleitet von `std::exception` werfen

Beispiel: Exceptions

```
1  void RuntimeThrow() { throw std::runtime_error("example"); }
2  void BadThrow()     { throw std::bad_exception(); }
3  void IntThrow()     { throw 42; }
4
5  int main() {
6    try {
7      RuntimeThrow();      // erster catch Block
8      BadThrow();          // zweiter catch Block
9      IntThrow();          // dritter catch Block
10   } catch (const std::runtime_error& e) {
11     std::cout << "std::runtime_error: " << e.what() << std::endl;
12   } catch (const std::exception& e) {       // Referenz auf Basis-Klasse
13     std::cout << "std::exception: " << e.what() << std::endl;
14   } catch (...) {                           // default catch block
15     std::cout << "unknown exception" << std::endl;
16   }
17 }
```

Modifizierung und Weiterwerfen

```
1  struct MutableException : std::exception {
2    explicit MutableException(const std::string& message) : message_(message) {
3    }
4    MutableException& operator+=(const std::string& additional_text) {
5      message_ += additional_text;
6      return *this;
7    }
8    const char* what() const noexcept override { return message_.c_str(); }
9   private:
10   std::string message_;
11 };
12
13 int main() {
14   try {
15     throw MutableException("info");
16   } catch (MutableException& e) {
17     e += ": additional info";
18     throw;       // aktuelle Exception weiterwerfen
19 } }
```

noexcept

- Funktion oder Lambda wird nie Exception werfen
- Compiler kann Handling-Code eliminieren
- Performance-Optimierung
- Kan mittels noexcept-Operator getestet werden

```
1  void RuntimeThrow() { throw std::runtime_error("example"); }
2  void NoThrow() noexcept {}
3
4  int main() {
5    std::cout << "RuntimeThrow is noexcept: " << noexcept(RuntimeThrow()) << std::endl;
6    std::cout << "NoThrow is noexcept: " << noexcept(NoThrow()) << std::endl;
7  }
```

Weitere Exception-Themen

- Exception-Handling ist teuer
- Move-Semantik benötigt `noexcept`
  - Wenn eine Move-Operation fehlschlagen könnte, wäre einerseits das neue als auch das alte Objekt in invalidem Zustand.
  - Siehe: `std::move_if_noexcept`
- Behandeln von Exception in Default Catch Block
  - Möglich mittels `std::current_exception`, `std::exception_ptr` und `std::rethrow_exception`
- Konstruktoren können Fehlschlag nur über Exceptions kommunizieren
  - Während dem Exception Handling werden evtl. Destruktoren von Attributen aufgerufen
  - Destruktoren dürfen nie Exceptions werfen