Modernes C++

...für Programmierer

Fehlerbehandlung

by

Dr. Günter Kolousek

31. Juli 2019

Fehlerarten

- Benutzerfehler
  - menschlicher Benutzer
  - jede Eingabe überprüfen!!!
  - d.h. eigentlich kein Fehler…
- Systemfehler
  - Aufruf eines ”system call”
  - können nicht vorhergesagt werden
- Programmierfehler
  - z.B. memory leak
  - z.B. Aufruf einer Funktion mit ungültigem Argument
    - Parameter in Vorbedingung erfasst, dann muss Argument nicht überprüft werden $\rightarrow$ *undefined behaviour* (UB)
    - Parameter nicht in Vorbedingung erfasst, dann dies als Fehler melden $\rightarrow$ *defined behaviour*

Fehler ”finden”

- Fehlermeldungen ausgeben
  - Ausgaben auf `stdout` oder `stderr`
  - `assert()`, d.h. Debug-Mode
    - in Abhängigkeit von Makro NDEBUG
  - Loggingausgaben, auch in Datei oder über Netzwerk
- Backtrace
- Debugger
- automatisierte Tests, wie z.B. Unit-Tests

Fehler behandeln

- Ort der Fehlerbehandlung
  - direkt wo Fehler entsteht
  - an anderer Stelle
    - z.B. wo sinnvolle Behandlung möglich
- Wiederherstellbarkeit?
  - gültiger Zustand herstellbar
    - Fehler kann ignoriert werden
  - gültiger Zustand nicht herstellbar
    - $\rightarrow$ Programm terminieren
- Fehlerart
  - vorübergehender Fehler
  - periodische Fehler
  - permanente Fehler
- Fehlerbehandlungsstrategie
  - wie soll auf den Fehler reagiert werden?

Feherbehandlungsstrategie

- Fehler melden
  - wen? Benutzer, ”aufrufende” Dienste, Administratoren, Entwickler
  - was? Ort, Kontext (wie hierher), Beschreibung
- Fehler ignorieren
- Operation abbrechen
- nochmals probieren
  - u.U. mit tw. veränderten Daten
  - u.U. von Zeitpunkt abhängig
  - u.U. unter Zuhilfenahme eines anderen Dienstes
- Fehler beheben
  - z.B. mittels Fehlerkorrektur
- Programm terminieren

Aspekte der Fehlerbehandlung

- Erkennung einer Fehlersituation
- Übermittlung der Fehlerinformation zu Fehlerbehandlungsroutine
- Einhalten eines gültigen Zustandes
- Vermeiden von Ressource-Leaks

Erkennung der Fehlersituation

- Vorbedingung wird nicht eingehalten
  - bei *defined behaviour*, wenn fehlerhafte Argumente übergeben
- Nachbedingung wird nicht eingehalten
- Vorbedingung für andere aufgerufene Funktion wird nicht eingehalten
- Invariante wird nicht eingehalten

Übermittlung der Fehlerinformation

- Fehlercode zurückliefern
- Setzen eines Fehlerzustandes
- Werfen einer Exception

Fehlercode zurückliefern

- Aufrufer kann vergessen diesen abzufragen
  - kann durch C++ Attribut `[[nodiscard]]` verhindert werden
- Aufrufender Code besteht aus vielen Abfragen
- Rückgabetyp verbietet einen geeigneten Rückgabewert
  - z.B. `int`
  - alternativ: `std::optional`
  - alternativ: `std::pair` oder `std::tuple`

Setzen eines Fehlerzustandes

- Implementierungen
  - globale Variable
    - z.B. `errno`
    - nächster Aufruf überschreibt Wert
  - Instanzvariable
- Aufrufender Code besteht aus vielen Abfragen

Werfen einer Exception

- Auslösen/Werfen einer Exception
  - in C++: `throw`
- Abfangen einer Exception/Exception-Handler
  - in C++: `catch`
- Nichtbehandlung $\rightarrow$ Prozess terminiert!
- ”kleiner” Overhead beim Aufrufen einer Funktion
- Nachverfolgung durch stack trace

Nichtbehandlung

```
try {
    main(argc, argv);
} catch (...) {
    if (terminate_handler != nullptr) {
        terminate_handler();
    } eles {
        terminate();
    }
}
```

set_terminate()

```
void my_terminate() {
    cout << "uncaught exception!" << endl;
    exit(1);
}

int main() {
    set_terminate(my_terminate);
    ....
}
```

noexcept

```
vector<int> read_file(string_view name) noexcept {
    ...
}
```

- eine `noexcept` Funktion *soll* keine Exception werfen
  - wenn doch, dann $\rightarrow$ `terminate()`
- Compiler kann etwas besser optimierten Code generieren
  - siehe Overhead beim Aufrufen einer Funktion
- Dokumentation!!!

Exception-safety

- *nothrow guarantee* (auch *nofail*)
  - es wird keine Exceptions geworfen
- *strong exception guarantee*
  - wenn Operation fehlschlägt, dann wird der Zustand des Programmes in den Zustand übergeführt, der vor Aufruf der Operation gewesen ist.
  - d.h. fehlgeschlagene Operationen bewirken keine Nebeneffekte
    - d.h. hat rollback-Semantik

Exception-safety – 2

- *basic exception guarantee*
  - wenn Exception auftritt, dann ist Programm in gültigen Zustand
  - alle Invarianten sind gültig
  - d.h. fehlgeschlagene Operation kann Nebeneffekte bewirken
- *no exception guarantee*
  - wenn Exception auftritt, dann gibt es absolut keine Garantien
  - Invarianten sind nicht gültig
  - d.h. beliebige Nebeneffekte, memory leaks,…

Richtlinien

- Spezifiziere Vor-, Nachbedingungen und Invarianten
- Exceptions nur für Fehlersituation verwenden
- Behandle Fehler an der richtigen Stelle
  - d.h. nicht jede Exception muss bei jedem Funktionsaufruf abgefangen werden
- informative und prägnante Fehlermeldung angeben
  - interner vs. externer Benutzer?
    - $\rightarrow$ Informationsleak (z.B. SQL-Injection)

Richtlinien für C++

- Exception per `const exception&` abfangen
  - 1 Kopie (oder move-Operation) weniger
- `catch` - Exceptionhandler in richtige Reihenfolge bringen
- Destruktoren sollen/dürfen keine Exception werfen!!!
  - sind implizit `noexcept`
- Fehler in Konstruktoren mit Exceptions melden
  - beachten, dass Destruktor *nicht* aufgerufen wird!
- Verwende RAII, um Leaks zu verwenden

Exceptionklassen in C++

- `logic_error`
  - `invalid_argument`
  - `domain_error`
    - Eingaben außerhalb des gültigen Bereiches
    - wird von Standardbibliothek nicht verwendet
  - `length_error`
    - implementierungsabhängige Länge überschritten
    -
  - `out_of_range`
    - Zugriff auf ein Element außerhalb des definierten Bereiches
  - `future_error`
    - $\rightarrow$ `std::future`, `std::promise`,...

Exceptionklassen in C++– 2

- `runtime_error`
  - außerhalb des Wirkungsbereichs des Programmes
  - können nicht einfach vorhergesehen werden
  - z.B. `overflow_error`
- `bad_optional_access`
- `bad_cast`
- `bad_weak_ptr`
- ...