# Grundkurs C++

## Typwandlung
## Ausnahmebehandlung

Martin Knopp, Martin Gottwald, Stefan Röhrl

02.05.2018

# Schlüsselwort „const“

```
int a = 5;
const int b = 3;,

a = 2; // in Ordnung
b = 4; // Fehler: const-Variable kann nicht verändert werden
```

# const-Zeiger

```
const int* pX; // Zeiger auf const int
*pX = 3;       // Nicht möglich
pX = &y;       // Zeiger auf andere Variable zeigen
               // zu lassen ist möglich

int* const pY; // const-Zeiger auf int
*pY = 4;       // in Ordnung
pY = &z;       // Zeiger kann nicht auf andere Variablen zeigen

const int* const pZ; // const-Zeiger auf const int
*pZ = 3; pZ = &w;    // beides nicht möglich
```

# const bei Methoden

```
class Test {
    public:
        Test(int i) {
            m_i = i;
        };
        ~Test();
        const int* GetI() const { // Methode ändert keine
            return &m_i;          // Attribute des Objekts
        };
    private:
        int m_i;
};
```

# Referenzen

```
void swap(int& i, int& j) {
    int tmp = i;
    i = j;
    j = tmp;
}

int main() {
    int x, y;
    swap(x,y);

    return 0;
}
```

Referenzen sind keine Kopien oder Pointer auf ein Objekt, sondern das Objekt selbst.

# Referenzen

- Eine einmal erzeugte Referenz kann nicht mehr auf ein anderes Objekt „verschoben“ werden.
- Referenzen können nicht „null“ sein.
- Referenzen können nicht uninitialisiert sein:

```
int& rA; // falsch

int A = 3;
int& rA = A; // richtig
```

# Standard-C-Casting

```
int main() {
    int a = 3;
    int b = 4;
    float c = a/b;
    float d = (float)a/b;
    unsigned long e = (unsigned long)a;
}
```

- C++ ist typsicher
- Typumwandlung ist fehleranfällig
  - mit Bedacht verwenden
  - auffindbar machen
  - den Compiler mithelfen lassen

# Typumwandlung à la C++

- die Wandlung zum Beruhigen des Compilers
  - ersetzt Wandlung zwischen Standardtypen und implizite Wandlung
  - sagt dem Compiler, dass man Informationsverlust in Kauf nimmt (z. B. double $\rightarrow$ int)
  - Schlüsselwort `static_cast<type>(...)`
- entlang von Klassenhierarchien
  - Schlüsselwort `dynamic_cast<type>(...)`
- Daten anders interpretieren als üblich
  - Schlüsselwort `reinterpret_cast<type>(...)`
- Aufheben der const-Eigenschaft
  - Schlüsselwort `const_cast<type>(...)`

# static_cast

```
int main() {
    int i = 1234;
    long l;
    float f;

    // Konvertierung ohne cast:
    l = i;
    f = i;
    // oder mit cast:
    l = static_cast<long>(i);
    f = static_cast<float>(i);
}
```

# static_cast

```
// Verlustbehaftete Konvertierung

i = l; // Hier koennen Stellen verloren gehen
i = f; // Hier kann Genauigkeit verloren gehen

// sagt „ich weiß“, eliminiert Warnungen
i = static_cast<int>(l);
i = static_cast<int>(f);
char c = static_cast<char>(i);
```

# const_cast

```
class Foo {
    public:
        void func() {} // a non-const member function
};

void someFunction(const Foo& f) {
    f.func(); // Compiler-Fehler!
    Foo &fRef = const_cast<Foo&>(f);
    fRef.func(); // okay
}
```

# dynamic_cast

```
void meineFunktion(Auto* meinAuto) {
    Cabrio* meinCabrio = dynamic_cast<Cabrio*>(meinAuto);

    if (meinCabrio != NULL)
        meinCabrio->verdeckOeffnen();
    else
        std::cerr << "Kein Cabrio!" << std::endl;

    meinAuto->bremsen(); //Klappt immer!
}
```

# dynamic_cast

- `B *b_ptr = dynamic_cast<B*> object_ptr;`
- gibt 0 zurück, wenn es nicht klappt
- Upcast geht immer
- Downcast gelingt nur, wenn `object_ptr` tatsächlich auf ein Objekt von richtigem Typ zeigt
- vtable muss angelegt sein (d. h. die Basisklasse muss eine virtuelle Methode besitzen)
- Umwandlung von einer Kindklasse in eine andere wird scheitern

# reinterpret_cast

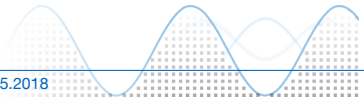
```
int i = 25;

float *fp = reinterpret_cast<float*>(i);
// Zeigt auf einen wahrscheinlich undefinierten Float
// an Speicheradresse 25
```

- mächtigster Cast-Operator
- Variablen werden „uminterpretiert“, das Ergebnis ist nur in den wenigsten Fällen direkt nutzbar → Rückcasting i. d. R. erforderlich
- geeignet für bitweise Modifikationen
- sehr fehleranfällig, nur in wohlüberlegten Situationen verwenden!

Teil II

# Ausnahmebehandlung



# Motivation

- Im Ablauf eines Programms passieren unerwartete Dinge
  - Dateien fehlen oder haben nicht erwarteten Inhalt, Timeouts, Speichermangel
  - Fehlende Daten führen zu lokal nicht lösbaren Problemen (Fehler kann häufig nur auf höherer Programmebene gelöst werden)
- Rückgabewerte sind nicht immer geeignet und können ignoriert werden
- Erreichen der Zielsetzung und Fehlerbehebung sind völlig unabhängige Aspekte
  - Code zur Wahrung der Funktionalität $\leftrightarrow$ Code zur Fehlerbehebung
  - Vorteil bei unabhängigen Codefragmenten, d. h. bei der Teamarbeit

# Ansatz

- Fehler erkennen und melden
  - Man wirft eine Ausnahme/Exception
  - Schlüsselwort `throw`
- Codestellen auf geworfene Ausnahmen überprüfen
  - Schlüsselwort `try`
- Behandlung der aufgetretenen Ausnahme
  - (ab)fangen eines Fehlers
  - Schlüsselwort `catch`

# throw

```
throw int(5);
throw Fehlerklasse();
```

- Man kann Ausnahmen beliebigen Typs werfen:
  Standarddatentypen, beliebige Klassen
- Die Auswahl hängt mit davon ab, welche Information transportiert werden soll bzw. wie verschiedenartig die zu erwartenden Fehler sind.
- Das geworfene Objekt wird erzeugt und bis zur Behandlung an aufrufende Funktionen zurückgegeben.
- Unbehandelte Ausnahmen führen zu Programmabbruch.
- Bei der Behandlung einer Ausnahme kann man diese durch `throw;` erneut werfen.

# try-Block

```
try {
    // Anweisungsblock
}
catch(int) {
    // Fehlerbehandlung für Integer
}
catch(AK) {
    // Fehlerbehandlung für Klasse AK
}
catch(BK){
    // Fehlerbehandlung für Klasse BK
}
catch(...){
    // Fehlerbehandlung für alle anderen Fehler
}
```

Nur ein **`catch`** wird ausgeführt!

Prüfung in Reihenfolge des Codes!

# Was tun mit einer gefangenen Ausnahme?

- Problem beheben und Funktion erneut aufrufen
- Weiter ohne erneuten Aufruf der Funktion
  - einen Zustand herstellen, der ohne die Funktion zurechtkommt
  - alternative Berechnung/Standardwerte verwenden
- Im Rahmen der aktuellen Möglichkeiten handeln und das Problem nach oben weiterreichen
  - durch Werfen der gleichen Ausnahme
  - durch Werfen einer anderen Ausnahme
- Das Programm beenden

Wenn man die Ausnahmenbehandlung möglichst einfach und übersichtlich hält, vereinfacht man einerseits das Debugging und bekommt andererseits ein stabileres Programm.

# Unbehandelte und unerwartete Ausnahmen

- Bleibt eine Ausnahme unbehandelt: Aufruf von `std::terminate()`
- Kann bei Bedarf umgestellt werden:
  `terminate_handler set_terminate(terminate_handler)`
- Ausnahmenspezifikation
  ```
  void wirftFehler() throw(Kl, std::bad_alloc);
  ```
- Ohne Ausnahmenspezifikation: Alle Ausnahmen sind möglich
- Werfen einer nicht genannten Ausnahme: Aufruf von `std::unexpected()`
- Kann bei Bedarf umgestellt werden:
  `terminate_handler set_unexpected(terminate_handler)`
- `unexpected()`: Hilfreich vor allem beim Testen

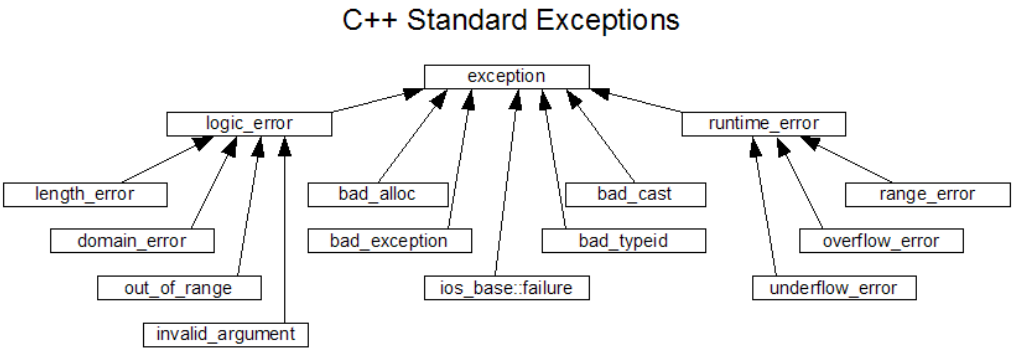
# Standardausnahmen

- Sprachelemente und die Standardbibliothek werfen Ausnahmen
- Standardausnahmen sind in Klassenhierarchie gegliedert
- *exception* empfiehlt sich auch als Basisklasse für eigene Fehlerklassen

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

    private:
        ...
}
```

# C++ Standard Exceptions



# dynamic_cast

```
void meineFunktion(Auto* meinAuto) {
    Cabrio* meinCabrio = dynamic_cast<Cabrio*>(meinAuto);

    if (meinCabrio != NULL)
        meinCabrio->verdeckOeffnen();
    else
        std::cerr << "Kein Cabrio!" << std::endl;

    meinAuto->bremsen(); //Klappt immer!
}
```

# dynamic_cast mit Referenz

```
void meineFunktion(Auto& meinAuto) {
    try {
        Cabrio& meinCabrio = dynamic_cast<Cabrio&>(meinAuto);
    }

    catch (const std::bad_cast& e) {
        std::cerr << "Kein Cabrio!" << std::endl;
    }

    meinAuto->bremsen(); //Klappt immer!
}
```