

Kurseinheit 3: Klassen 2

1. Exceptions
2. Casts und RTTI
3. Beispiel SimpleString: Step 1

1. Exceptions

Fehlerbehandlung

In C wurde bereits Fehlerbehandlung eingehend diskutiert. Bei neueren Funktionen wie `strncpy_s` wird der Fehlercode in den Rückgabewert codiert. Dabei steht meist eine 0 für "okay". Andere Rückgabeparameter werden dabei mittels Call by Reference übergeben.

Objektorientierte Sprachen bieten eine weitere Funktionalität an: **Exceptions!**

Wird eine Exception (Objekt) geworfen, so wird die Funktion/der Block verlassen, bis irgendwo ein Exception Handling stattfindet. Dies kann innerhalb dieser Funktion oder oberhalb im Call Stack sein. Falls kein Exception Handling stattfindet, wird das Programm unterbrochen.

Wer kann Exceptions werfen?

- Der eigene Programmcode (throw)
- Aber auch Bibliotheksfunktionen, die verwendet werden.



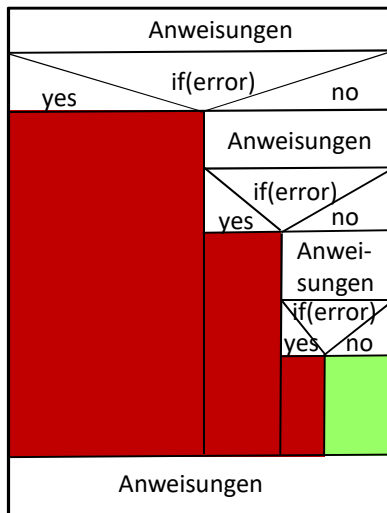
Bug

1. Exceptions

3

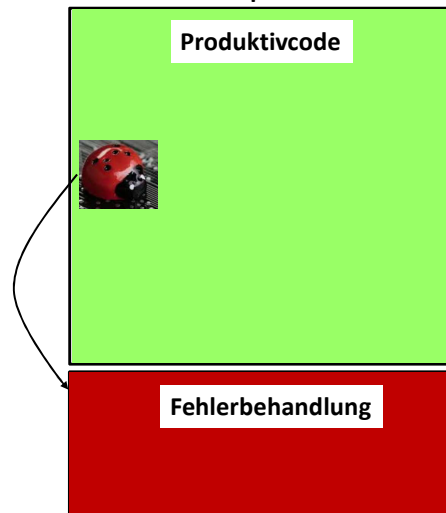
Prinzipien: Fehlerbehandlung

Klassische Fehlerbehandlung



Code unverständlicher - schneller

Exceptions



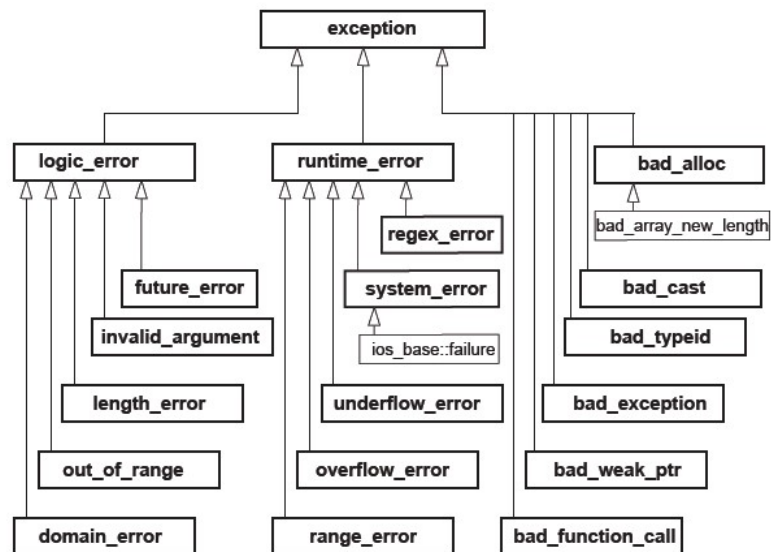
Code besser verständlich - langsamer

1. Exceptions

4

Vorhandene Exceptions

C++ bietet schon Exceptions (Klassen) an. Rechts ist eine solche Vererbungshierarchie als Klassendiagramm aufgezeigt. Vererbung wird in einem späteren Kapitel noch behandelt.



1. Exceptions

5

Vorhandene Exceptions

Um die vorhandenen Exceptionklassen nutzen zu können, müssen unterschiedliche Header eingebunden werden.

Die Exceptionklassen befinden sich alle im Namensraum std (Name Space).

Klasse	Bedeutung	Header
<code>exception</code>	Basisklasse	<code><exception></code>
<code>logic_error</code>	theoretisch vermeidbare Fehler, zum Beispiel Verletzung von logischen Vorbedingungen	<code><stdexcept></code>
<code>invalid_argument</code>	ungültiges Argument bei Funktionen	<code><stdexcept></code>
<code>length_error</code>	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet	<code><stdexcept></code>
<code>out_of_range</code>	Bereichsüberschreitungsfehler	<code><stdexcept></code>
<code>domain_error</code>	anderer Fehler des Anwendungsbereichs	<code><stdexcept></code>
<code>future_error</code>	für asynchrone System-Aufrufe	<code><future></code>
<code>runtime_error</code>	nicht vorhersehbare Fehler, z.B. datenabhängige Fehler	<code><stdexcept></code>
<code>regex_error</code>	Fehler bei regulären Ausdrücken	<code><regex></code>
<code>system_error</code>	Fehlermeldung des Betriebssystems	<code><system_error></code>
<code>range_error</code>	Bereichsüberschreitung	<code><stdexcept></code>
<code>overflow_error</code>	arithmetischer Überlauf	<code><stdexcept></code>
<code>underflow_error</code>	arithmetischer Unterlauf	<code><stdexcept></code>
<code>bad_alloc</code>	Speicherzuweisungsfehler (Details siehe Abschnitt 7.2)	<code><new></code>
<code>bad_typeid</code>	falscher Objekttyp (vgl. Abschnitt 6.9)	<code><typeinfo></code>
<code>bad_cast</code>	Typumwandlungsfehler (vgl. Abschnitt 6.8)	<code><typeinfo></code>
<code>bad_weak_ptr</code>	kann vom <code>shared_ptr</code> -Konstruktor geworfen werden	<code><memory></code>
<code>bad_function_call</code>	kann von <code>function::operator()()</code> geworfen werden	<code><functional></code>

1. Exceptions

6

Basisklasse: exception

```
class exception
{
public:
    exception();
    exception(const char* const &message);
    exception(const char* const &message, int);
    exception(const exception &right);
    exception& operator=(const exception &right);
    virtual ~exception();

    virtual const char *what() const;
};
```

} MS Erweiterung von exception

Gibt den Text zurück

<https://stackoverflow.com/questions/62908900/why-doesnt-stdexception-have-a-move-constructor>.

Move Semantic (Move Constructor und Move Assignment) fehlt in Exceptions. Details unter Stackoverflow.

1. Exceptions

7

Beispiel try-catch intraprozedural

```
try
{
    std::cin >> s32Val;
    if (s32Val == 0)
    {
        throw std::runtime_error("Zero not allowed");
    }
    std::cout << "More Code" << std::endl;
}
catch (std::runtime_error re)
{
    std::cout << re.what() << std::endl;
}
catch (...)
{
    std::cout << "Other exception" << std::endl;
}
```

Tritt eine Exception auf, wird der Block verlassen.

Findet sich kein passender catch-Block, wird die Funktion verlassen. catch(...) fängt alle nicht behandelten Exceptions.

Bei einfachen Lernbeispielen wird meist nur ein Text ausgegeben. In der Praxis müssen hier deutlich komplexere Vorgänge realisiert werden (Rollbacks, Speicherfreigabe, ...).

1. Exceptions

8

Beispiel try-catch interprozedural

```
void vCallerDiv(void)
{
    try
    {
        int32_t s32Res = s32Div(4, 0);
    }
    catch (std::exception ex)
    {
        std::cout << ex.what() << std::endl;
    }
}

int32_t s32Div(int32_t s32Val1, int32_t s32Val2)
{
    int32_t s32Res;
    if (s32Val2 == 0)
    {
        throw std::exception("div by zero");
    }

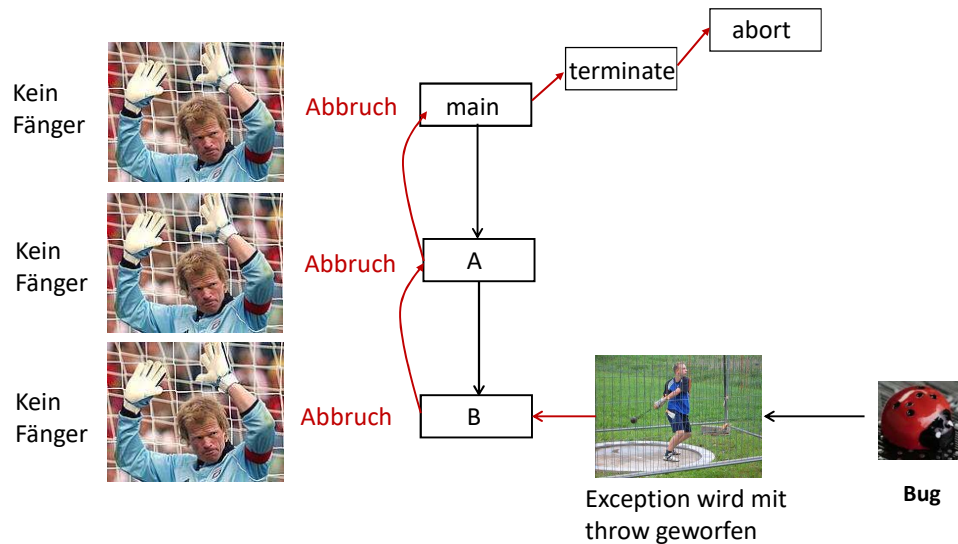
    s32Res = s32Val1 / s32Val2;
    return s32Res;
}
```

Falls hier kein Exception – Handling, dann würde die Exception weiter nach oben (Caller von vCallerDiv) propagiert werden (1). Ansonsten wird in catch-Block gesprungen (2).

1. Exceptions

9

Exception wird nicht „gecatched“?



1. Exceptions

10

noexcept (C++11)

Funktionen, die keine Exception werfen, sollten mit `noexcept` (äquivalent zu `noexcept(true)`) deklariert werden. Dies bringt Vorteile bei der Codegenerierung.

Default-Konstruktor (leerer Konstruktor, Kopierkonstruktor, Movekonstruktor, der Default-Destruktor sowie die Copy- und Move-Zuweisungsoperator sind alle `noexcept`.

```
class Car
{
public:
    Car(Car&& rCar) noexcept;
};
```

Bei Verwendung einer Initialisierungsliste ist `noexcept` vor die Initialisierungsliste zu schreiben.

```
Car::Car(Car&& rCar) noexcept : pac_( rCar.pac_ )
{
    std::cout << "Move Constructor of Car entered" << std::endl;
    u32MaxSpeed_ = rCar.u32MaxSpeed_;
    pac_ = rCar.pac_;
    rCar.pac_ = nullptr;
}
```

2. Casts und RTTI

11

Implizite und explizite Casts

Implizite und explizite Casts wurden bereits in C behandelt. Die bekannten expliziten C-Casts können weiterhin verwendet werden. Allerdings sollte man in C++ eher diese C++-Casts verwenden. Die C++-Casts sind sicherer und können schon beim Compilieren auf Probleme hinweisen.

- `static_cast <Type>(Expression)`
- `const_cast<Type>(Expression)`
- `reinterpret_cast <Type>(Expression)`
- `dynamic_cast<Type>(Expression)`



https://m.media-amazon.com/images/I/41ljmcm8s8L_AC.jpg

Werden statt impliziter immer explizite Casts verwendet, so ist ersichtlich, dass der Programmierer diese Typumwandlung wohlüberlegt hat.

2. Casts und RTTI

12

`static_cast<Type>(Expression)`

```
f64_t f64Val = 47.11;

int32_t s32Val1 = f64Val; //Warning in C++
int32_t s32Val2 = (int32_t)f64Val;
int32_t s32Val3 = static_cast<int32_t>(f64Val);
```

Nur eine Warnung in C++:
conversion from
'f64_t' to 'int32_t',
possible loss of data

```
eColor eMyColor = eColor::Red;

//s32Val1 = eMyColor; //Compilerfehler in C++
s32Val2 = (int32_t)eMyColor;
s32Val3 = static_cast<int32_t>(eMyColor);
```

Hier Fehler in C++:
A value of type
„eColor“ cannot be
assigned to an entity
of type „int32_t“

```
fCar* pCar1 = new Car();

//Empty* pEmptyThing1 = pCar1; // Compilerfehler C++
Empty* pEmptyThing1 = (Empty*)pCar1; // später Runtime error
//Empty* pEmptyThing1 = static_cast<Empty*>(pCar1); // Compilerfehler C++
```

Unterschied: `static_cast` findet den Fehler
während der Compilezeit!

2. Casts und RTTI

13

`const_cast<Type>(Expression)`

Entfernt die const-Eigenschaft – Ist anhand des Namens nicht zu erwarten.

```
const uint32_t cu32Val = 73U;
uint32_t u32Val = cu32Val; //No problem
u32Val++;

vDoSomething(u32Val); //No problem
//vDoSomething(cu32Val); //Compiler Error
vDoSomething(const_cast<uint32_t>&(cu32Val));
//cu32Val++; //is still const, Compiler Error
```

Eine konstante Variable darf nicht als Referenz an eine Funktion übergeben werden, die dann den Wert des Originals ändert. Mit `const_cast` wird die const-Eigenschaft für den Methodenaufwurf entfernt.

```
void vDoSomething(uint32_t& u32X)
{
    u32X++;
}
```

2. Casts und RTTI

14

`reinterpret_cast<Type>(Expression)`

Die **Brute-Force-Methode**, die zu Laufzeitfehlern führen **kann**, falls hier was nicht stimmig ist.

```
class Car
{
public:
    uint32_t u32GetMaxSpeed() const
    {
        return u32MaxSpeed_;
    }

private:
    uint32_t u32MaxSpeed_ = 130U;
};
```

```
class Mouse
{
public:
    void vEatCheese(void)
    {
        u32Cheese_--;
    }

private:
    uint32_t u32Cheese_ = 100U;
};
```

```
Car* pCar1 = new Car();
Mouse* pMouse = reinterpret_cast<Mouse*>(pCar1); //Kein Fehler C++
pMouse->vEatCheese();
std::cout << pCar1->u32GetMaxSpeed() << std::endl;
```

u32MaxSpeed_ in pCar1 wird verändert. Kein Laufzeitfehler!

2. Casts und RTTI

15

dynamic_cast<Type>(Expression)

Wird bei polymorphen Klassen (Vererbung und virtuelle Methoden) verwendet.

```
class Father
{
public:
    Father() {}
    ~Father() {}
    virtual void vDoIt() {}
};

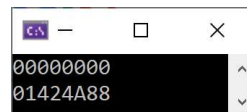
class Child : public Father
{
public:
    Child(){}
    ~Child() {}
    virtual void vDoIt() {}
};
```

```
Father* pFather1 = new Father();
Father* pFather2 = new Child();

Child* pChild1 =
    dynamic_cast<Child*>(pFather1);
std::cout << pChild1 << std::endl;

Child* pChild2 =
    dynamic_cast<Child*>(pFather2);
std::cout << pChild2 << std::endl;
```

pFather1 ist Father-Objekt und kann nicht gecastet werden. pFather2 ist Child-Objekt und kann gecastet werden.



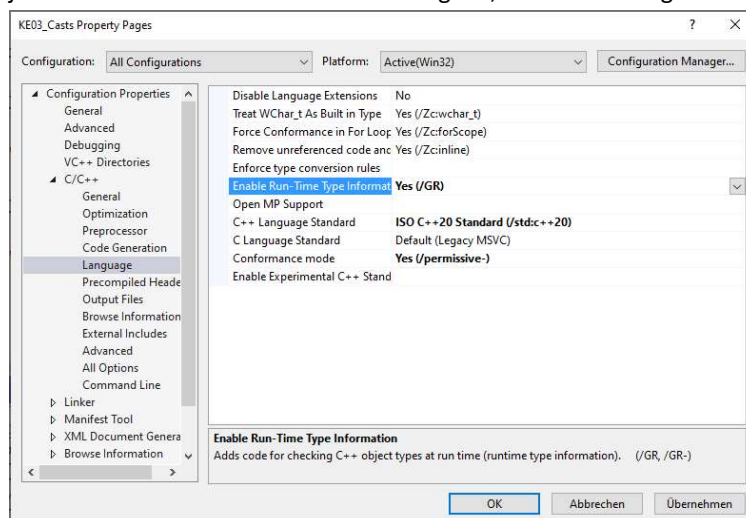
Mit dynamic_cast kann in polymorphen Klassen überprüft werden, um welchen Objekttyp es sich handelt! Dabei ist der Rückgabewert auf nullptr zu überprüfen.

2. Casts und RTTI

16

Runtime Type Information - Einschalten

Ist anfänglich ausgeschaltet. Soll die RTTI benutzt werden, ist diese einzuschalten! **Nachteil** ist, dass für jede Klasse Metainformationen hinterlegt ist, was den Code größer macht.

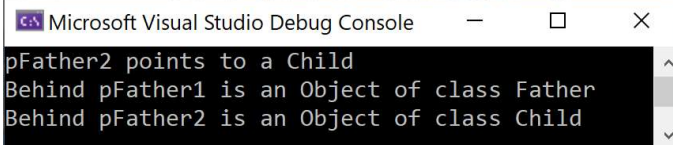


2. Casts und RTTI

17

Runtime Type Information - Anwendung

```
Father* pFather1 = new Father();
Father* pFather2 = new Child();
if (typeid(*pFather1) == typeid(Child))
{
    std::cout << "pFather1 points to a Child" << std::endl;
}
if (typeid(*pFather2) == typeid(Child))
{
    std::cout << "pFather2 points to a Child" << std::endl;
}
std::cout << "Behind pFather1 is an Object of "
          << typeid(*pFather1).name() << std::endl;
std::cout << "Behind pFather2 is an Object of "
          << typeid(*pFather2).name() << std::endl;
```



```
Microsoft Visual Studio Debug Console
pFather2 points to a Child
Behind pFather1 is an Object of class Father
Behind pFather2 is an Object of class Child
```

2. Casts und RTTI

18

Runtime Type Information – typeid-Operator

Der typeid-Operator liefert eine Referenz auf ein Objekt der Klasse type_info zurück! (siehe typeid.h oder typeidinfo)

```
class type_info
{
protected:
    const char *__name;
public:
    const char* name() const { return __name; }
    //...
};
```

Um den konkreten Objekttyp zu bestimmen, sollte eher ein dynamic_cast eingesetzt werden, da hier die zusätzlichen Meta-Informationen eines Objektes nicht erstellt werden (Speicher und Performance).

RTTI ist ähnlich (viele Diskussionen dazu unter stackoverflow) zu Reflections in Java und C#.

3. Beispiel SimpleString Step1

19

SimpleString: char-Array verpackt in eine Klasse

```
class SimpleString
{
public:
    explicit SimpleString(uint32_t u32NewMaxSize) noexcept(false);
    ~SimpleString() noexcept(true);
    SimpleString(const SimpleString&) = delete;
    SimpleString(SimpleString&&) = delete;
    SimpleString& operator=(const SimpleString&) = delete;
    SimpleString & operator=(SimpleString&&) = delete;

    void vPrintString(const char* cpcTag) const noexcept(true);
    bool bAppendLine(const char* cpcLine) noexcept (true);

private: // Three Attributes
    uint32_t u32MaxSize_;
    uint32_t u32Length_;
    char* pcBuffer_;
};
```

↑
Zeiger auf C-Zeichenkette

Default-Implementierungen werden gelöscht.

3. Beispiel SimpleString Step1

20

SimpleString: RAII – Konstruktor allokiert

```
SimpleString::SimpleString(uint32_t u32NewMaxSize) noexcept(false)
: u32MaxSize_{ u32NewMaxSize }, u32Length_{0}
{
    std::cout << "SimpleString Constructor called" << std::endl;
    if (u32MaxSize_ == 0U) // should be at least 1
    {
        throw std::runtime_error("SimpleString: u32MaxSize == 0U");
    }

    pcBuffer_ = new char[u32MaxSize_]; // like malloc
    if (pcBuffer_ == nullptr)
    {
        throw std::runtime_error("SimpleString new failed");
    }
    pcBuffer_[0] = 0x00; // EOS
}
```

Zwei Attribute (u32MaxSize_ und u32Length_) werden in der Initialisierungsliste initialisiert. Für das dritte Attribut (pcBuffer_) wird dynamisch Speicher allokiert (u32MaxSize_).

3. Beispiel SimpleString Step1

21

~SimpleString: RAI – Destruktor gibt Speicher frei

```
SimpleString::~SimpleString() noexcept
{
    std::cout << "SimpleString Destructor called" << std::endl;

    if (pcBuffer_ != nullptr)
    {
        delete[] pcBuffer_;
    }
}
```

If expression evaluates to a null pointer value, no destructors are called, and the deallocation function may or may not be called (it's unspecified), but the default deallocation functions are guaranteed to do nothing when passed a null pointer.

<https://en.cppreference.com/w/cpp/language/delete>

Eine Abfrage wäre folglich oben nicht notwendig. Ähnliches gilt für free aus C.

The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs.

<http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf>

3. Beispiel SimpleString Step1

22

Zwei Methoden

```
void SimpleString::vPrintString(const char* cpcTag) const noexcept
{
    std::cout << cpcTag << pcBuffer_ << std::endl;
}
bool SimpleString::bAppendLine(const char* cpcline) noexcept
{
    bool bRet = false;
    const auto aLength_pcline = strlen(cpcline);

    if ((aLength_pcline + u32Length_ + 2U) <= u32MaxSize_)
    {
        strncpy_s((pcBuffer_ + u32Length_), (u32MaxSize_ - u32Length_),
                  cpcline, aLength_pcline);
        u32Length_ += aLength_pcline;
        pcBuffer_[u32Length_++] = '\n'; // tricky
        pcBuffer_[u32Length_] = 0x00;   // EOS
        bRet = true;
    }
    return bRet;
}
```

Trick mit Rückgabewert ohne else.

Attribut u32Length_ enthält die Anzahl der Zeichen in pcBuffer_

3. Beispiel SimpleString Step1

23

Anwendung von SimpleString

```
SimpleString* pSimpleString1 = new SimpleString{ 13U };\n\npSimpleString1->vPrintString("pcBuffer is\\n");\n\nbRet = pSimpleString1->bAppendLine("Hello");\npSimpleString1->vPrintString("pcBuffer is\\n");\nstd::cout << "bRet is " << bRet << std::endl;\n\nbRet = pSimpleString1->bAppendLine("World");\npSimpleString1->vPrintString("pcBuffer is\\n");\nstd::cout << "bRet is " << bRet << std::endl;\n\n// Adding further character "" + '\\n' failed\nbRet = pSimpleString1->bAppendLine("");\npSimpleString1->vPrintString("pcBuffer is\\n");\nstd::cout << "bRet is " << bRet << std::endl;\n\ndelete pSimpleString1; // don't forget -> Mem Leak
```



```
SimpleString Constructor called\npcBuffer is\n\npcBuffer is\nHello\n\nbRet is 1\npcBuffer is\nHello\nWorld\n\nbRet is 1\npcBuffer is\nHello\nWorld\n\nbRet is 0\nSimpleString Destructor called
```