

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

- mittels `std::set_unexpected()` kann man dieses Verhalten ändern
- Aber: Herb Sutter (Exceptional C++, Item 13) und auch Boost (Exception-specification rationale):

Never write an exception specification !

- Destruktoren sollten **NIEMALS** Ausnahmen erzeugen:

```
x::~~X() // throw ();
```

WARUM



1. Elementares C++

neu in C++11: noexcept

statisches no-throw:

```
void i_will_never_throw (int i) noexcept { ... } // noexcept(true)  
template <typename T> bool compare(T t1, T t2) noexcept( noexcept(t1==t2))  
    { return t1 == t2; }
```

sämtliche Exception-Vorkehrungen können im Code entfernt werden

falls doch eine Exception auftritt: terminate

1. Elementares C++

neu in C++11: exception nesting

Ausnahmen verpacken:

```
void lib_func (int i) {  
    try {  
        if (i<0) throw low_level_ex();  
    }  
    catch (...) {  
        std::throw_with_nested(high_level_exception());  
    }  
} // die aktuelle Ausnahme wird in eine neue eingepackt ...
```

```
namespace std {  
class nested_exception { ... // mixin class: std 18.8.6  
    [[noreturn]] void rethrow_nested() const;  
    [[noreturn]] template<class T> void throw_with_nested(T&& t);  
    template <class E> void rethrow_if_nested(const E& e);  
};
```

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling

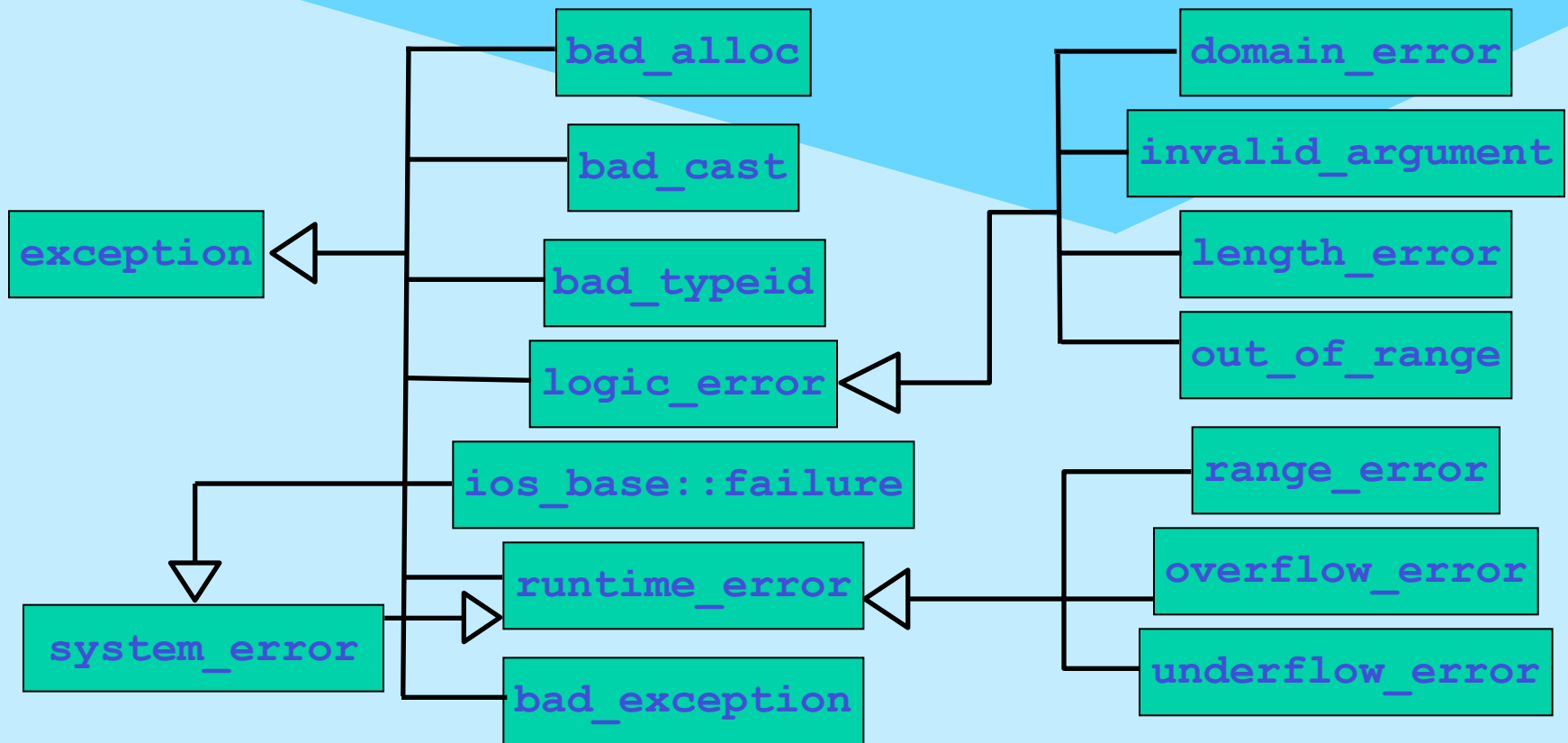
- es gibt die Möglichkeit eine ganze Funktion als **try**-Block zu implementieren:

```
int foo(int i)
try {
    may_throw(i); return 0;
}
catch (int ex) {
    return -1;
}
```

- Es gibt eine Reihe vordefinierter Ausnahmen

1. Elementares C++

1.5. Strukturierte Anweisungen: Exception Handling



Always be exception-aware ...

H. Sutter: Exceptional C++ Item 18

ITEM 18: CODE COMPLEXITY—PART 1 **DIFFICULTY: 9**

This problem presents an interesting challenge: How many execution paths can there be in a simple three-line function? The answer will almost certainly surprise you.

How many execution paths could there be in the following code?

```
String EvaluateSalaryAndReturnName( Employee e )
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

<i>If you found:</i>	<i>Rate yourself:</i>
3	Average
4–14	Exception-aware
15–23	Guru material

The 23 are made up of:

- 3 nonexceptional code paths
- 20 invisible exceptional code paths

watch the videos on:

<http://exceptionsafecode.com>

2. Klassen in C++

Um das Klassenkonzept ranken sich alle wichtigen (oo) Konzepte:

- abstrakte Datentypen (Daten & Operationen)
- Zugriffsschutz
- nutzerdefinierte Operatoren
- Vererbung, Polymorphie & Virtualität
- generische Typen (Templates)

2. Klassen in C++ [back -->](#)

```
// Stack.h
#ifndef STACK_H
#define STACK_H
class Stack {
protected:
    int *data;
    int top, max;
public:
    Stack(int = 100);
    Stack(const Stack&);
    ~Stack();
    void push (int);
    int pop();
    int full() const;
    int empty() const;
};
#endif
```

prevents multiple inclusion !

ein neuer Typ !

Memberdaten

Memberfunktionen

Konstruktoren (u.u. viele)

Destruktor (einer !)

const Memberfunktion: Zusage, das Objekt nicht zu verändern

Zugriffsmodi



2. Klassen in C++

```
// Stack.cc
#include "Stack.h"
#include <cstdlib>

Stack::Stack(int dim): max(dim), top(0), data(new int[dim]) { }
Stack::Stack(const Stack & other) // Copy-Konstruktor
: max(other.max), top(other.top), data(new int[other.max]) {
    for (int i=0; i<top; ++i)
        data[i]=other.data[i];
}
Stack::~~Stack() {
    delete [] data; // Feld statt einzelнем Objekt !
}
void Stack::push (int i) {
    if (!full()) data[top++]=i;
    else std::exit(-1);
} // if (!this->full()) this->data[this->top++]=i;
```

initializer list



NICHT: max

Scope resolution

2. Klassen in C++

```
int Stack::pop () {  
    if (!empty()) return data[--top];  
    else std::exit(-1);  
}  
int Stack::full() const { return top == max; }  
int Stack::empty() const { return top == 0; }
```

- alternativ Memberfunktionen im Klassenkörper: dann implizit inline
- oder außerhalb des Klassenkörpers mit expliziter inline-Spezifikation (im Headerfile !)

```
// Nutzung:  
#include "Stack.h"  
void foo() {  
    Stack s1 (1000); s1.push(123);  
    Stack *sp = new Stack; sp->push(321);  
    delete sp; // ansonsten memory leak !  
}
```

2. Klassen in C++

- Wann immer Objekte entstehen, läuft automatisch ein (passender) **Konstruktor** !
- Wann immer Objekte verschwinden, läuft automatisch der **Destruktor** !
- Klassen ohne nutzerdefinierten Konstruktor/Destruktor besitzen implizit
 - den sog. *default constructor* `X () {}` memberweise Kopie !
 - den sog. *default copy-constructor* `X (const X&) { ... }`
und
 - den sog. *default destruktur* `~X () {}`
- sobald nutzerdefinierte Konstruktor-Varianten vorliegen, gibt es nur noch den impliziten Copy-Konstruktor (wenn dieser nicht auch explizit definiert wird)

2. Klassen in C++

- Jedes Objekt enthält seine eigene Realisierung der Memberdaten (**NICHT** der Memberfunktionen!)
- Die Identität eines Objektes ist mit seiner Adresse verbunden !

```
bool Any::same (Any& other) {  
    return this == &other;  
}
```

?

Beispiel: Jedes **Stack**-Objekt hat das folgende Layout unabhängig davon, wie es entstanden ist !



kein overhead durch
Meta-Daten !

2. Klassen in C++

- es sind auch sog. unvollständige Klassendeklarationen erlaubt, von einer solchen Klasse können jedoch bis zu ihrer vollständigen Deklaration lediglich Zeiger & Referenzen benutzt werden:

```
class B;  
class A {B * my_B; ....}; // oder ... class B* my_B;  
class B {A * my_A; ....};
```

- strukturell identische Klassen mit verschiedenen Namen bilden verschiedene Typen (es gibt jedoch die Möglichkeit, nutzerdefiniert Kompatibilität herbeizuführen s.u.):

```
class X { public: int i; } x0;  
class Y { public: int i; } y0;  
X x1 = y0; Y y1 = x0; // beides falsch !!!  
x0 = y0; y0 = x;     // beides falsch !!!
```

2. Klassen in C++

- Konstruktorparameter sind beim Anlegen von Objekten (geeignet) anzugeben, d.h es muss einen entsprechenden Konstruktor geben

```
// direct initialization:  
X x0;           // needs X::X();  
X x1(1);        // needs X::X(int);  
X x2 = X(2,0);  // needs X::X(int,int);  
X *pb = new X (5,true); // needs X::X(int, bool);  
X x3(1, "zwei", '3'); // needs X::X(int,[const] char*,char);  
  
// copy initialization:  
X x3 = 1;       // X tmp(1); X x3(tmp); ggf. elision
```

2. Klassen in C++

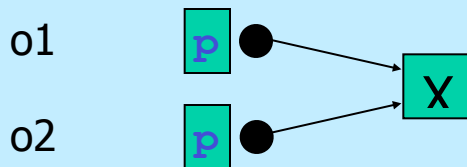
Copy-Konstruktoren

`X::X(const X&); // kanonische Form !`

shallow copy

(default copy ctor)

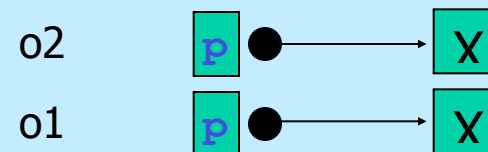
```
class SC {
    X* p;
public:
    SC(): p(new X) {}
};
SC o1;
SC o2=o1;
```



deep copy

(nutzerdefinierter copy ctor)

```
class DC {
    X* p;
public:
    DC(): p(new X) {}
    DC(const DC& src)
        : p(new X) {...copy X };
};
DC o1;
DC o2=o1;
```



2. Klassen in C++

- Konstruktoren können auch mit einem function try block implementiert werden, auch wenn ein passender handler vorliegt, wird die Ausnahme **immer** re-thrown !!!

```
struct Y {  
    X* p;  
    Y(int i) try : p(new X)  
    { if (i) throw "huhh"; }  
    catch(...)  
    { /* delete p; NOT ALLOWED !!! */  
      /* throw "huhh"; implicitly */}  
    ~Y() { delete p; }  
};
```

15.3 (10): Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.