1.5. Strukturierte Anweisungen: Exception Handling

- Exceptions sind Objekte beliebigen Typs
- stack unwinding ruft Destruktoren aller erfolgreich konstruierten Objekte
- in einem `catch`-Block kann eine Exception mittels `throw;` 're-thrown' werden

```
....          catch (...) {
                             void handleAll(); // proto
....                         handleAll();
....          }

void handleAll() {
   try { throw; }
   catch (double x) { cout << x << endl; } // z.B.
}
```

1.5. Strukturierte Anweisungen: Exception Handling

- wird eine Exception nirgends 'gefangen', so endet dass Programm durch aufruf von `std::terminate()`(* (dies ruft wiederum `std::abort()`)
- mittels `std::set_terminate()` kann man dieses Verhalten ändern:

Prototyp
```
void (*set_terminate(void (*term_handler)())) ();
                                  ?????
```

oder leichter nachvollziehbar:

```
typedef void (*TH)();
TH set_terminate(TH);
```

(* es ist implementation-defined, ob dabei stack-unwinding stattfindet !!!

1.5. Strukturierte Anweisungen: Exception Handling

- `std::terminate()` wird auch gerufen, wenn während der Behandlung einer Ausnahme eine weitere Ausnahme auftritt
- Funktionen können mit sog. exception specifications ausgestattet sein, (entspricht den throws-Klauseln von Java aber)

Java: `void foo(); // lässt keine Exceptions 'raus'`

C++: `void foo(); // lässt beliebige Exceptions 'raus'`

```
void foo () throw (dies, das, nochwasanderes);
```

Java: vollständige Flussanalyse zur Compile-Zeit

C++: keinerlei Flussanalyse, aber Überwachung zur Laufzeit

- tritt eine Exception auf, die nicht spezifiziert wurde, wird `std::unexpected()` (dies ruft wiederum `std::terminate()`) gerufen

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

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`

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 wir in eine neue eingepackt ...

namespace std {
class nested_exception { ... // mixin class: 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);
};
```

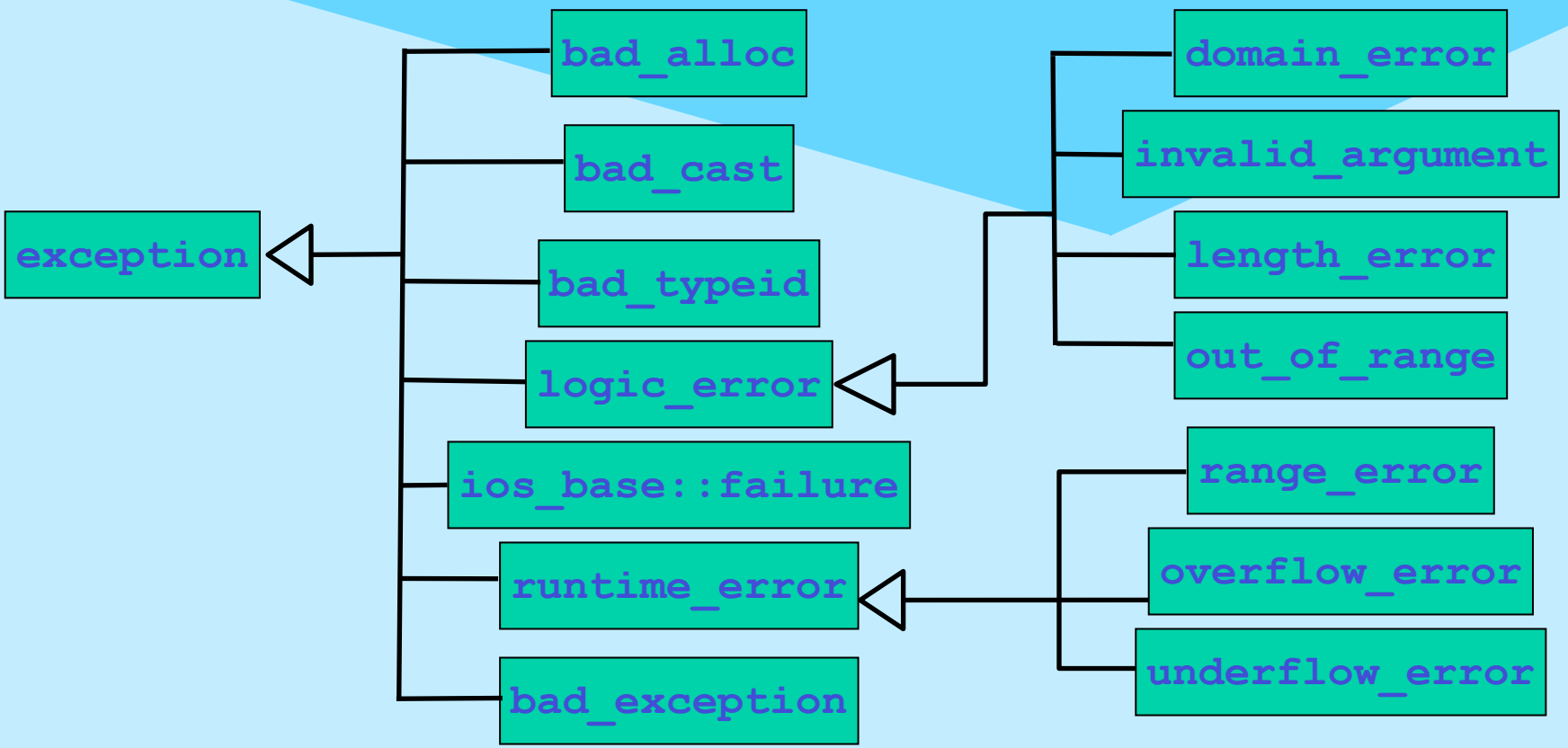
1.5. Strukturierte Anweisungen: Exception Handling

- 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

1.5. Strukturierte Anweisungen: Exception Handling



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)

```
// 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

```
// 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 einzelnem 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

```
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 !
}
```