Bereits behandelt:

- Namensräume, Klassen, Vererbung
- Templates
- Standard-Bibliothek

Heute:

- C++: Ausnahmen, Elementzeiger, Alias-Analyse, inline
- Projekte: Header, Übersetzen und Linken, Makefiles, Tools

- Mittels `throw` kann eine Ausnahme geworfen werden.
  Dies kann z. B. geschehen, um einen Fehler anzuzeigen.

- Durch `catch` kann eine solche Ausnahme gefangen werden.
  An dieser Stelle kann dann auf den Fehler reagiert werden.

- `try` grenzt den Bereich ein, aus dem die Ausnahme gefangen werden kann.

```
try
{
    p = new double[n];
}
catch (std::bad_alloc)
{
    std::cerr << "Kein Speicher frei!" << std::endl;
}
```

```
#include <iostream>

int main()
{
    char c;
    std::cout << "Geben Sie einen Buchstaben ein: ";
    std::cin >> c;

    try
    {
        if (c == 'X')
            throw(1);
        else
            throw(c);
    }
    catch (int)
    {
        std::cout << "Wer hat hier einen 'int' geworfen?" << std::endl;
    }
    catch (char msg)
    {
        std::cout << "Sie haben ein " << msg << " geworfen!" << std::endl;
    }

    return 0;
}
```

```
#include <iostream>

int main()
{
    try
    {
        try
        {
            throw(1);
        }
        catch (char)
        {
            std::cout << "Ich habe ein 'char' gefangen" << std::endl;
        }
    }
    catch (int)
    {
        std::cout << "Ich habe ein 'int' gefangen" << std::endl;
    }

    return 0;
}
```

- Mit `throw` wirft man ein Objekt eines beliebigen Typs.
- Danach sucht das Programm am Ende aller umschließenden `try`-Blöcke von innen nach außen nach einem `catch`, das diesen Typ fangen kann. Der zugehörige `catch`-Block wird dann ausgeführt. Wird kein passendes `catch` gefunden, bricht das Programm ab.
- Ein umschließender `try`-Block ist ein `try`-Block, den das Programm betreten, aber noch nicht wieder verlassen hat. Dieser Block muss nicht in der aktuellen Funktion liegen, sondern kann auch in einer aufrufenden Funktion liegen. (Um dorthin zu gelangen, wird Stack-Unwinding durchgeführt.)
- Mit `catch (...)` kann man alle Ausnahmen fangen.
- Eine Funktion kann in einer `throw`-Liste angeben, welche Ausnahmen aus ihr und den von ihr aufgerufenen Funktionen geworfen werden können:
  ```
  void f() throw(int, char);
  void g() throw();
  ```
  Wird eine andere Ausnahme geworfen, bricht das Programm ab.

```
#include <iostream>

void f()
{
    throw("Grippe");
}

void g()
{
    f();
}

int main()
{
    try
    {
        g();
    }
    catch (const char msg[])
    {
        std::cout << "Ich habe mir die " << msg << " gefangen" << std::endl;
    }

    return 0;
}
```

```
#include<iostream>

struct A
{
    int i;
    void f() { std::cout << "Hallo" << std::endl; }
};

int main()
{
    int A::* ptr       = &A::i;    // Zeiger auf ein Datenelement von A
    void (A::* fptr)() = &A::f;    // Zeiger auf eine Elementfunktion von A

    A a;
    a.*ptr = 3;
    (a.*fptr)();
    std::cout << a.i << std::endl;

    A* p = &a;
    p->*ptr = 4;
    (p->*fptr)();
    std::cout << a.i << std::endl;

    return 0;
}
```

- Ein Elementzeiger zeigt auf ein Element in einer Klasse. Dazu speichert er die Adresse relativ zum Klassenanfang. Um mit dem Elementzeiger auf ein Datenelement zugreifen zu können, braucht man zusätzlich noch eine Instanz der Klasse.

- Der Elementzeiger wird deklariert als
  `Datentyp Klasse::* ElemZeiger`
  Es gibt keine Elementzeiger auf den Typ `void`.

- Die relative Adresse eines Datenelements, die man in einem solchen Zeiger speichern kann, erhält man mit
  `&Klasse::Datenelement`
  Man kann einen Elementzeiger aber auch mit `0` belegen.

- Auf ein Datenelement der `Instanz` der `Klasse` greift man so zu:
  `Instanz.*ElemZeiger`
  `ZeigerAufInstanz->*ElemZeiger`

- Bei Zeigern auf Elementfunktionen ist die Syntax ähnlich.

```
void func (double c[], const double a[], const double b[], int n)
{
    for (int i=0; i<n; ++i)
        c[i+1]=a[i]+b[i];
}
```

Bei dieser Funktion spielt es für die Optimierung eine große Rolle, ob `c` auf dasselbe Feld wie `a` oder `b` zeigt. Denn dann muss eine Addition komplett abgeschlossen sein, bevor die nächste beginnt.

Durch `__restrict` kann man dem Compiler mitteilen, dass auf das Feld, auf das ein Zeiger zeigt, auch nur über diesen Zeiger zugegriffen wird und nicht über einen anderen Zeiger oder eine sonstige Variable:

```
void func (double* __restrict c, const double a[], const double b[], int n)
{
    for (int i=0; i<n; ++i)
        c[i+1]=a[i]+b[i];
}
```

Der Compiler darf annehmen, dass Zeiger, die auf unterschiedliche Datentypen zeigen, nicht auf denselben Speicherbereich zeigen:

```
#include <iostream>

int main()
{
    short a[2];

    a[0] = 0x1111;
    a[1] = 0x1111;

    *(int *)a = 0x22222222;   // Verletzung der Alias-Regeln

    std::cout << std::hex << a[0] << ' ' << a[1] << std::endl;
    return 0;
}
```

Bei Optimierung liefert dieses Programm nicht unbedingt das erwartete Resultat 2222 2222!

Mittels einer Union kann man das Problem aber umgehen. Hier weiß der Compiler, dass beide Daten an derselben Speicherstelle liegen:

```
#include <iostream>

int main()
{
    union
    {
        short a[2];
        int i;
    } u;

    u.a[0] = 0x1111;
    u.a[1] = 0x1111;

    u.i = 0x22222222;

    std::cout << std::hex << u.a[0] << ' ' << u.a[1] << std::endl;
    return 0;
}
```

Ferner können Variablen vom Typ `char` sich den Speicherplatz mit anderen Variablen teilen.

- Bei einem normalen Funktionsaufruf werden zunächst die Rücksprungsadresse und die Funktionsargumente auf den Stack gelegt. Danach wird ein Sprung zu der Funktion durchgeführt. Am Ende der Funktion wird der `return`-Wert auf den Stack gelegt und der Rücksprung zu der gespeicherten Adresse durchgeführt.
- Bei einer kleinen Funktion, die häufig aufgerufen wird, ist das verschenkte Zeit. Daher kann man die Funktion auch als `inline` deklarieren. Die Funktion wird dann nicht als eigenständige Einheit übersetzt, sondern an der Stelle des Aufrufs in das Programm eingesetzt.
- Vorteil: Die Sprünge entfallen. Nachteil: Der Code kann aufgebläht werden, da die Funktion mehrfach auftaucht.
- Die Funktion muss definiert (und nicht nur deklariert) sein, bevor man sie aufruft.
- Bei hoher Optimierung versuchen die Compiler, selbst Funktionen zu inlinen, wenn dies profitabel erscheint.

In Klassen definierte Funktionen sind automatisch `inline`.

```
inline void f1() {}    // inline
void f2() {}           // nicht inline

struct A
{
    A() {}             // inline
    void g1() {}       // inline
    inline void g2();
    void g3();
};

void A::g2() {}        // inline
void A::g3() {}        // nicht inline
```

- Quellcode-Dateien sollten nie beliebig groß werden, denn sonst werden sie unübersichtlich. Es ist besser, sie in einzelne Dateien zu zerlegen, die kleinere logische Einheiten widerspiegeln.

- Header-Dateien dienen als Bindeglieder zwischen diesen Dateien, indem sie gemeinsame Datentypen und Funktionsprototypen (die man zum Aufruf der Funktionen braucht) zur Verfügung stellen.

- Variablen und Funktionen sollten nicht in Headern definiert werden, da doppelte Definitionen Linker-Fehler verursachen würden.

- Eine Ausnahme bilden `inline`- und Template-Funktionen, die aufgrund ihrer Natur nicht gelinkt werden und daher in Headern auftauchen können.

header.h:

```
#ifndef _MY_HEADER_  // header guard
#define _MY_HEADER_  // verhindert doppeltes einbinden

struct A
{
    int i;
    A() : i(1) {}
};

void print_A (const A&);

#endif
```

---

main.cc:

```
#include "header.h"

int main()
{
    A a;
    print_A(a);
    return 0;
}
```

print.cc:

```
#include <iostream>
#include "header.h"

void print_A (const A& a)
{
    std::cout << a.i << std::endl;
}
```

- Die einzelnen `.cc`-Dateien kann man jetzt einzeln übersetzen und die erhaltenen `.o`-Dateien dann hinterher zusammenlinken:

```
g++ -O2 -c main.cc
g++ -O2 -c print.cc
g++ -o MeinProgramm main.o print.o
```

- Ändert man nur `print.cc`, so muss man auch nur diese Datei neu übersetzen und hinterher alles zusammenlinken.

- Ändert man eine Header-Datei, dann muss man alle Teile neu übersetzen, die diesen Header direkt oder indirekt einbinden.

- Das Übersetzen kann man mit dem Befehl `make` automatisieren. Man legt dazu ein `Makefile` an, das die Namen der Dateien enthält, die erzeugt werden sollen. Zu jeder Datei gehört eine Liste, wovon sie abhängig ist. Dann folgen die Befehle, die zum Erzeugen ausgeführt werden müssen.

```
CXX    = g++                          # Name des C++-Compilers
CFLAGS = -O2 -Wall -W -pedantic       # Compiler-Optionen
LIBS   =                              # zu linkende Bibliotheken

MeinProgramm: main.o print.o
        $(CXX) -o MeinProgramm main.o print.o $(LIBS)

main.o: main.cc header.h
        $(CXX) $(CFLAGS) -c main.cc

print.o: print.cc header.h
        $(CXX) $(CFLAGS) -c print.cc

.PHONY: clean

clean:
        rm -f *.o MeinProgramm
```

- Ein Debugger hilft bei der Fehlersuche.
  Dazu sollte man sein Programm mit der Option `-g` übersetzen.

- Open Source Debugger unter Linux: `dbx`, `gdb`, `ddd`, `gvd`

- Die häufigsten Befehle in `gdb`:

`list`	Programmcode an der aktuellen Stelle auflisten
`run`	Programmausführung starten
`break <file::line>`	Haltepunkt setzen
`next`	nächsten Befehl ausführen (aber nicht in das Unterprogramm hineinspringen)
`step`	nächste Anweisung ausführen (und evtl. in das Unterprogramm hineinspringen)
`continue`	Programmausführung fortsetzen
`print <Variable>`	Wert einer Variable ausdrucken
`bt`	aktuellen Stack auflisten

- Das Programm `valgrind` hilft dabei, falsche Speicherzugriffe aufzudecken.

- Wenn ein Programm zu langsam läuft, kann man ein Laufzeit-Profil des Programms erzeugen. Darin kann man sehen, wo mehr Zeit beansprucht wird als vermutet.

- Open Source Profiler unter Linux: `gprof`

- Man übersetzt sein Programm mit der Option `-pg` und startet es. Beim Laufen erzeugt das Programm eine Datei `gmon.out`, aus der `gprof` das Laufzeitprofil erstellt.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 19.88      2.04     2.04   967680     0.00     0.00  ECL<1, CoordT>::eval()
 11.60      3.23     1.19       21    56.67    83.85  MBuilder<double>::Build()
  8.77      4.13     0.90   414720     0.00     0.00  ECL<2, double>::GetGrad()
  8.48      5.00     0.87  6224964     0.00     0.00  ECL<2, double>::eval()
  7.70      5.79     0.79   161280     0.00     0.00  ECL<2, CoordT>::eval()
```

- Größere Projekte haben eine längere Entwicklungszeit, und Fehler werden oft erst nach einiger Zeit bemerkt. Daher ist es sinnvoll, nicht nur die aktuellste Version zu haben, sondern auch ältere, damit man nachschauen kann, bei welchen Änderungen der Fehler eingeschleppt wurde. Das grenzt den Fehler oft schon stark ein.

- Meist arbeiten mehrere Entwickler in einem Team zusammen. Damit nicht jeder mit einer anderen Programmversion arbeitet, sollte der Quellcode vernünftig verwaltet werden.

- Hier helfen Programme zur Versions-Kontrolle. Einige bekannte sind: `cvs`, `subversion`, `arch`

- Wenn man eine Änderung am Quellcode vorgenommen und getestet hat, dann „checkt“ man sie ein. Die Kollegen können diese Änderung dann „auschecken“. Über alle Änderungen führt die Versionskontrolle Buch, so dass man auch alte Versionen auschecken kann.