

Systemprogrammierung

Teil 5: Einführung in C++

Referenzen, Operator-Overloading,
Namensräume, Klassen, ...



C++: Überblick

Bjarne Stroustrup hat C++ als Erweiterung von C entwickelt:

- Ausnahmebehandlung, Namensräume, Referenzen, Überladen von Funktionen und Operatoren
- objektorientierte Programmierung: Klassen, Vererbung, Polymorphie, dynamische Bindung
- generische Programmierung: Templates
- objektorientierte und Template-basierte Erweiterungen der C-Standardbibliothek (*u.a. Ein-/Ausgabe-Klassen, String-Klasse, Vector-Klasse, intelligente Zeiger*)

ISO-Standards:

- C++98 von 1998 (*mit Ergänzungen 2003 und 2007*)
- C++11 von 2011 (*mit Ergänzungen 2014 und 2017*)

*Bjarne Stroustrup zu C++11:
"It feels like a new Language"*

weitere Bibliotheken außerhalb der ISO-Standards für viele Domänen, z.B.:

- Boost-Bibliotheken (*nützliche Erweiterungen der Standardbibliothek*)
- Qt (*grafische Benutzungsoberflächen*)

C++ Ein-/Ausgabe: Streams und Operatoren



In C++ dienen **Stream**-Objekte als Eingabe-Quellen und Ausgabe-Ziele. Ein-/Ausgabe-Anweisungen werden mit den Operatoren `<<` und `>>` formuliert. Die Variablen und Funktionen liegen im Namensraum `std`:


```
#include <iostream> // std::cout, std::cin, std::hex, std::endl, operator<<, operator>>

int main()
{
    std::cout << "Dezimalzahl eingeben: ";
    int zahl;
    std::cin >> zahl;
    std::cout << "Hexadezimalzahl: " << std::hex << zahl << std::endl;
}
```

Die C-Bibliotheksfunktionen gibt es weiterhin, jetzt im Namensraum `std`:

```
#include <cstdio> // aus <name.h> wird bei C++ <cname>
...
std::printf(...);
...
```

C++ Strings: verwendbar wie ein primitiver Werttyp

In C++ können **String**-Objekte per Zuweisung kopiert, per Operator **+** konkateniert und per Vergleichsoperatoren **==**, **<**, **<=** usw. verglichen werden: 

```
#include <string> // std::string, operator+, operator==, ...
...
std::string a = "halli";
std::string s = "hallo";
std::string t; // leerer String
if (a < s) {
    t = a + s; // t bekommt Wert hallihallo
}
...
```

Die C-Strings sind aber ebenfalls nutzbar:

```
#include <cstring>
...
std::size_t n = std::strlen("hallihallo");
...
```

C++ Arrays: echte Typen statt verkappter Zeiger

In C++ gibt es Arrays, die die Anzahl ihrer Elemente kennen.

- Arrays mit zur Übersetzungszeit fester Länge:

```
#include <array> // std::array, size, operator[ ], ...  
...  
std::array<double, 4> a = {3.625, 3.648, 3.853, 4.042};  
for (std::size_t i = 0; i < a.size(); ++i)  
{  
    std::cout << a[i] << std::endl;  
}
```

- Arrays mit variabler Länge:

```
#include <vector> // std::vector, size, operator[ ], ...  
...  
std::vector<double> v = {3.625, 3.648, 3.853, 4.042};  
for (std::size_t i = 0; i < v.size(); ++i)  
{  
    std::cout << v[i] << std::endl;  
}
```

C++ Referenzen: ersetzen an vielen Stellen die Zeiger

Eine Referenz definiert einen Aliasnamen für einen Speicherbereich.

Man kann sich Referenzen auch als Zeiger vorstellen, die garantiert zu jedem Zeitpunkt eine gültige Adresse enthalten, also insbesondere niemals den Wert NULL haben.

Achtung: die Referenzen von Java sind in C/C++-Sprechweise in Wahrheit Zeiger!

- **Variablen-Definition:**

`Typ Name = Wert;`

Initialisierung ist bei Referenzvariablen Pflicht

`Typ &Aliasname = Name;`

das & (ab C++11 auch&&) kennzeichnet eine Variable als Referenz

- **Verwendung:**

für Eingabeparameter (empfohlen bei `sizeof (Typ) > 4 * sizeof (int)`)

`void function(const Typ& inparam);`

für Ausgabeparameter

`void function(Typ& outparam);`

für Rückgabewerte (insbesondere bei überladenen Operatoren)

`Typ& operator=(const Typ& value);`

C++ Referenzen: Beispiel


Der C++-Compiler realisiert Referenzparameter mit Zeigern:

```
void increment(const int& m, int& n)
{
    n = m + 1;
}

...
int a = 1;
int b;
increment(a, b); // b wird 2
```

```
void increment(const int *m, int *n)
{
    *n = *m + 1;
}

...
int a = 1;
int b;
increment(&a, &b);
```



Hinweis:

der Eingabeparameter *m* sollte hier besser als *int* statt *const int&* definiert werden, weil `sizeof (int) <= 4 * sizeof (int)` und `sizeof (int) <= sizeof (int*)`

C++ Ausnahmen: try-catch-throw


In C++ können im Prinzip Werte aller Typen geworfen und gefangen werden. Guter Stil ist aber, nur Objekte zu werfen, die Instanz einer von `std::exception` abgeleiteten Ausnahmeklasse sind.

```
try
{
    if (...) throw MeineAusnahme(); // Stil: immer Objekt werfen, niemals Adresse!
    ...
    int *p = new int[SEHR_GROSSE_ZAHL]; // wirft evtl. std::bad_alloc
    ...
}
catch (std::bad_alloc& e) // Stil: Ausnahme per Referenz fangen (wegen Polymorphie)!
{
    std::cerr << e.what() << '\n'; // what liefert Fehlermeldung als C-String
}
catch (...) // fängt beliebige Ausnahme, insbesondere MeineAusnahme
{
    ...
}
```

ohne new !

der Name des Referenzparameters darf fehlen, wenn er im catch-Block nicht benutzt wird

C++ Heap: new und delete

In C++98 wird Heap-Speicher mit dem Operator **new** allokiert, mit Zeigern verwaltet und mit dem Operator **delete** bzw. **delete[]** wieder freigegeben: 

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    int *p = new int(1);
```

*einzelne ganze Zahl,
mit 1 initialisiert* 

```
    std::cout << *p << '\n';
```

```
    delete p;
```

```
    int *a = new int[2];
```

*Array von zwei ganze Zahlen,
nicht initialisiert*

```
    a[0] = 10;
```

```
    a[1] = 20;
```

```
    for (int i = 0; i < 2; ++i)
```

```
    {
```

```
        std::cout << a[i] << '\n';
```

```
    }
```

```
    delete[] a;
```


*wurde new mit [] aufgerufen,
muss auch delete mit [] aufgerufen werden*


```
}
```

C++ Heap: Intelligente Zeiger

Ab C++11 sollte Heap-Speicher mit intelligenten Zeigern (*smart pointers*) verwaltet werden. Intelligente Zeiger automatisieren die Freigabe von Heap-Speicher.

```
#include <memory> // smart pointer type std::unique_ptr<>, ...
#include <iostream>

int main()
{
     die Lebensdauer des Heap-Speicherstücks wird  
an die Lebensdauer der Variablen p geknüpft
    std::unique_ptr<int> p{ new int{1} };

     std::cout << *p << '\n'; // überladener operator*

    std::unique_ptr<int[]> a{ new int[2] };
    a[0] = 10; // überladener operator[]
    a[1] = 20;
    for (int i = 0; i < 2; ++i)
    {
        std::cout << a[i] << '\n';
    }
} automatischer Aufruf von delete bzw. delete[],  
weil die intelligenten Zeiger p und a ungültig werden
```

C++ Function-Overloading: Name-Mangling

C++ erlaubt das Überladen von Funktionen. 

Name-Mangling ist eine Technik, die aus den überladenen Funktionsnamen und den Parametertypen eindeutige Symbole für das Binden von Programmen bildet.

*Achtung: das Name-Mangling ist nicht standardisiert,
wird also von jedem Compiler potenziell anders umgesetzt*

- Beispiel für eine überladene Funktion:

```
int max(int, int);           // _Z3maxii
double max(double, double); // _Z3maxdd
```

*Symbole nach dem
g++ Name-Mangling*



- das Name-Mangling kann mit `extern "C"` unterbunden werden:

```
extern "C" int max(int, int, int);
```

*es kann in einem Programm nur eine Funktion **max** ohne Name-Mangling geben
mit C-Compiler übersetzte Funktionen werden so aus C++-Code aufrufbar*

C++ Operator-Overloading: Beispiel

C++ erlaubt das Überladen von Operatoren für benutzerdefinierte Typen, z.B. für Ausgabe-Streams:

```
#include <iostream>
```

```
enum jahreszeit {fruehling, sommer, herbst, winter};
```

```
std::ostream& operator<<(std::ostream& os, jahreszeit j)
{
    static const char *jahreszeiten[] = {
        "Fruehling", "Sommer", "Herbst", "Winter"};
    os << jahreszeiten[j];
    return os;
}
```

```
int main()
{
    jahreszeit j = sommer;
    std::cout << j << '\n'; // operator<<(std::cout, j) << '\n';
}
```

C++ Namensräume: Syntax

Namensräume (*Namespaces*) verringern das Risiko von Namenskonflikten:

- Namensraum-Deklaration:

```
namespace Namensraumname  
{  
    Deklarationen ...  
}
```



Java-Entsprechung:
package Paketname;

*definiert neuen Namensraum oder
erweitert bestehenden Namensraum
um weitere Deklarationen*

```
namespace  
{  
    Deklarationen ...  
}
```

*unbenannter Namensraum macht Deklarationen
für andere Übersetzungseinheit unsichtbar*

- Qualifizierung von Namen mit Scope Resolution Operator:

```
Namensraumname::EinName
```

- mit Using-Direktive auch Kurzschreibweise ohne Namensraumname:

```
using namespace Namensraumname;  
EinName
```

Java-Entsprechung:
import Paketname.*;

Beispielprogramm Namensraum

- Übersetzungseinheit Month
(besteht nur aus Header-Datei):

```
// month.h

#ifndef MONTH_H
#define MONTH_H

namespace htwg
{
    enum month
    {
        jan = 1, feb, mar,
        apr, may, jun,
        jul, aug, sep,
        oct, nov, dec
    };
}

#endif
```

- Hauptprogramm
(besteht nur aus Implementierungs-Datei):

```
// enumvar.cpp

#include "month.h"
using namespace htwg;
#include <iostream>
using namespace std;

int main()
{
    month m = htwg::oct;
    cout << m << '\n';
    ...
}
```

*ohne htwg::
mehrdeutig
wegen std::oct*

*eindeutig
htwg::month
gemeint*

eindeutig std::cout gemeint

C++ Klassen: Eigenschaften



C++ Klassen fassen die C-Konzepte Struktur (`struct`) und Funktion zusammen

- eine Klasse ist ein Bauplan für **Objekte**:

die Klasse legt fest, welche Daten ihre Objekte enthalten und welche Operationen auf diesen Daten möglich sind (**Kapselung**).

- eine Klasse hat **Membervariablen**:

enthalten die Daten eines Objekts

*Sprechweise Java:
Instanzvariablen*

- eine Klasse hat **Memberfunktionen**:

implementieren die Operationen

*Sprechweise Java:
Instanzmethoden*

*manche Operationen werden aber auch außerhalb der Klasse implementiert
(sie können dann in der Klasse als **friend** deklariert werden)*

- eine Klasse hat **Konstruktoren** und genau einen **Destruktor**:



jedes neue Objekt wird mit genau einem Konstruktoraufruf initialisiert und vor seiner Zerstörung (= *Speicherfreigabe*) läuft als letztes der Destruktor

Der Destruktor muss allen Ressourcen freigeben, der innerhalb der Klasse zusätzlich für das betreffende Objekt belegt worden sind (insbesondere Heap-Speicher).

C++ Klassen: Syntax (1)

- **Klassen-Deklaration** (meist in einer Header-Datei Klassenname.h): 

```
class Klassenname
```

```
{
```

```
public:
```

```
    Klassenname( );
```

```
// Default-Konstruktor
```

```
    Klassenname(const Klassenname& );
```

```
// Copy-Konstruktor
```

```
    Klassenname(Klassenname&& );
```

```
// Move-Konstruktor (ab C++11)
```

```
    ~Klassenname( );
```

```
// Destruktor
```

```
    Klassenname& operator=(const Klassenname& );
```

```
// Copy-Zuweisung
```

```
    Klassenname& operator=(Klassenname&& );
```

```
// Move-Zuweisung (ab C++11)
```

```
    Rückgabetyyp_1 Memberfunktion_1( ... );
```

```
    ...
```

```
    Rückgabetyyp_N Memberfunktion_N( ... );
```

```
private:
```

```
    Datentyp_1 Membervariable_1;
```

```
    ...
```


```
    Datentyp_M Membervariable_M;
```

```
};
```



Konstruktoren, Destruktor und
Zuweisungsoperatoren ergänzt
der Compiler in bestimmten Fällen
automatisch

C++ Klassen: Syntax (2)

- **Funktions-Definitionen** (meist in Implementierungsdatei-Datei Klassenname.cpp):
vor den Funktionsnamen muss `Klassenname::` stehen

```
Rückgabetyp_1 Klassenname::Memberfunktion_1(...)   
{  
    ... // Rumpf  
}  
...
```

- die Memberfunktionen einer Klasse haben implizit einen Parameter **this**:

`Klassenname * const this`  // konstanter Zeiger auf das Objekt des Aufrufs
*this müsste nach der heutigen Systematik von C++ eigentlich eine Referenz sein,
aus historischen Gründen ist es aber leider ein Zeiger* 

- Zugriff auf die Membervariablen über **this**:

this->Membervariable_1 // Kurzschreibeweise ohne **this->** möglich



C++ Klassen: Syntax (3)

- Objekt-Erzeugung

durch Variablen-Definition mit Klasse als Typ (*bei Wertobjekten die Regel*):

```
Klassenname Objektname; 
```

oder per Operator `new` auf dem Heap (*bei Entitäten die Regel*):

```
Klassenname *Objektzeiger = new Klassenname; // C++98
```



- Objekt-Benutzung:

Aufruf der öffentlichen Funktionen der zugehörigen Klasse
mit Komponentenauswahl- und Funktionsaufruf-Operator

```
Objektname.Memberfunktion_1( ... )
```

```
Objektzeiger->Memberfunktion_1( ... )
```



*der Compiler wandelt die obigen Schreibweisen in einfache Funktionsaufrufe
mit erstem Argument zum Initialisieren von **this**:*

```
Klassenname::Memberfunktion_1(&Objektname, ...)
```

```
Klassenname::Memberfunktion_1(Objektzeiger, ...)
```

C++ Klassen: Konstruktoren (1)

Konstruktoren sind diejenigen Funktionen einer Klasse, die Objekte initialisieren. Ein Konstruktor hat als Name den Klassennamen und hat keinen Rückgabetyt
Eine Klasse darf mehrere Konstruktoren haben, wenn sie unterschiedliche Parameter haben

Besondere Konstruktoren (*etwas vereinfacht dargestellt*):

- der parameterloser **Default-Konstruktor** initialisiert mit Standardwerten:

`Klassenname()`

ergänzt der Compiler automatisch, wenn eine Klasse ganz ohne Konstruktoren deklariert ist (ruft dann für alle Instanzvariablen mit Klassentyp deren Default-Konstruktor auf)

- der **Copy-Konstruktor** kopiert ein bestehendes Objekt:

`Klassenname(const Klassenname&)` 

ergänzt der Compiler unter Umständen automatisch, wenn er fehlt

- der **Move-Konstruktor** "bestiehlt" ein bestehendes Objekt (*Details dazu später*):


`Klassenname(Klassenname&&)` 

ergänzt der Compiler unter Umständen automatisch, wenn er fehlt


C++ Klassen: Konstruktoren (2)

Für Konstruktor-Implementierungen gibt es zwei Stile:

- Initialisierungsliste im Konstruktorkopf (*bevorzugter Stil*)

```
Klassenname::Klassenname( )  
: Membervariable_1( Wert_1) , ... , Membervariable_M( Wert_M)   
{ ... }
```

- Zuweisungen im Konstruktorrumpf (*funktioniert nicht bei `const`-Variablen*)

```
Klassenname::Klassenname( )  
{  
    Membervariable_1 = Wert_1;  
    ...  
    Membervariable_M = Wert_M;  
} 
```

- Konstruktoren (*mit Ausnahme des Move-Konstruktors*) sollten unbedingt eine Ausnahme werfen, wenn sie ein Objekt nicht konsistent initialisieren können

C++ Klassen: Konstruktoren (3)

Ein **Konstruktor-Aufruf** findet automatisch bei jeder Objekterzeugung statt:

- beim Gültigwerden einer Variablen mit Klassen-Typ:

```
Klassenname objektname; // Default-Konstruktor
Klassenname objektname(Argumentliste); // Konstruktor mit Parametern
Klassenname objektname{Argumentliste}; // Konstruktor mit Parametern (C++11)
Klassenname objektname = anderesObjekt; // Copy-Konstruktor
```

- bei **new** mit einem Klassen-Typ (C++98):

```
Klassenname *objektzeiger = new Klassenname;
Klassenname *objektzeiger = new Klassenname(Argumentliste);
```


- bei Wertparameter-Übergabe und Wert-Rückgabe von Funktions-Aufrufen:

```
aFunction( objektname ); // Copy-Konstruktor
return objektname; // Move-Konstruktor (bei C++98 Copy-Konstruktor) 
```

- bei impliziten Typanpassungen (*nicht bei **explicit** markierten Konstruktoren*):

```
objektname = wert; // Konstruktor mit Parameter vom Typ des Werts 
```

C++ Klassen: Destruktoren

Ein **Destruktor** ist diejenige Funktion einer Klasse, die Objekte vor ihrer Zerstörung (d.h. Speicherfreigabe) aufräumt. 

- ein Destruktor hat als **Name** den Klassen-Namen mit vorangestellter Tilde und hat weder Parameter noch einen Rückgabetyp:

`~Klassenname()`

jede Klasse hat genau einen Destruktor

wird eine Klasse ohne Destruktor deklariert, erzeugt der Compiler implizit einen Destruktor, der für alle Instanzvariablen mit Klassen-Typ deren Destruktor aufruft

Ein **Destruktor-Aufruf** findet automatisch statt

- beim Ungültigwerden einer Variablen mit Klassen-Typ:

```
{  
    Klassenname objektname;  
    ...  
} // objektname wird ungültig
```


- jedem **delete** für einen Zeiger mit Klassen-Typ (C++98): **delete** objektzeiger;

C++ Klassen: Zuweisungsoperatoren

Zuweisungsoperatoren sind diejenigen Funktionen einer Klasse, die Objekten einen neuen Wert geben, also quasi Objekte reinitialisieren.

Es gibt zwei Varianten von Zuweisungsoperatoren (ab C++11):

- die Copy-Zuweisung kopiert das Objekt der rechten Seite:

Klassenname& operator=(const Klassenname&) 

*ergänzt der Compiler unter Umständen automatisch
(ruft dann für alle Instanzvariablen deren Copy-Zuweisung auf)*

- die Move-Zuweisung "bestiehlt" das Objekt der rechten Seite (*Details später*):

Klassenname& operator=(Klassenname&&)

*ergänzt der Compiler unter Umständen automatisch
(ruft dann für alle Instanzvariablen deren Move-Zuweisung auf)*

wird anstelle der Copy-Zuweisung aufgerufen, wenn der rechte Operand der Zuweisung ein temporäres Objekt ist (z.B. der Rückgabewert einer Funktion)

C++ Klassen: Rule of five (1)

Die **Rule of five** (bei C++98 *Rule of three*) besagt, dass die folgenden speziellen Memberfunktionen in einer Klasse entweder alle explizit implementiert werden sollten oder keine davon:

- Destruktor
- Copy-Konstruktor
- Copy-Zuweisung
- Move-Konstruktor (*ab C++11*)
- Move-Zuweisung (*ab C++11*)

Begründung:

- Bei fehlender expliziter Implementierung ergänzt der Compiler die Funktionen unter bestimmten Umständen automatisch. Die Regeln dafür sind recht kompliziert, sodass eine Memberfunktion mitunter wider Erwarten existiert oder nicht existiert.
- Bei Klassen, die Ressourcen verwalten, können die automatisch erzeugten Implementierungen falsch sein, und wenn eine falsch ist, sind in der Regel alle fünf falsch.

C++ Klassen: Rule of five (2)

Ab C++11 kann der Automatismus für die Generierung von Memberfunktionen in einer Klasse überschrieben werden, z.B.:

- `Klassenname() = default;`

der Compiler soll unter allen Umständen einen Default-Konstruktor automatisch erzeugen, also auch dann, wenn es andere benutzerdefinierte Konstruktoren gibt

kann auch verwendet werden, um der erzeugten automatischen Implementierung ein anderes Zugriffsrecht als `public` zu geben.

- `Klassenname(const Klassenname&) = delete;`

der Compiler soll unter keinen Umständen einen Copy-Konstruktor automatisch erzeugen

üblich bei Entity-Klassen ohne Wertsemantik, die keine Kopien unterstützen dürfen (analog für Copy-Zuweisung, Move-Konstruktor und Move-Zuweisung)


C++ Klassen: static und friend Funktionen

- in einer **Klassen-Deklaration** können Funktionen als `static` oder `friend` markiert werden (*sie haben dann keinen Parameter `this`*):


```
class Klassenname
{
    ...
    static Rückgabetyp statische_Memberfunktion( ... );
    ...
    friend Rückgabetyp befreundete_Funktion( ... );
    ...
};
```

Sprechweise Java:
Klassenmethoden

- `static` Memberfunktionen werden mit Klassenname definiert und aufgerufen:

 Rückgabetyp **Klassenname::**statische_Memberfunktion(...) { ... } // Definition
wert = **Klassenname::**statische_Memberfunktion(...); // Aufruf

- `friend` Funktionen werden ohne Klassenname definiert und aufgerufen:

 Rückgabetyp befreundete_Funktion(...) { ... } // Definition
wert = befreundete_Funktion(...); // Aufruf

Beispielprogramm Klasse für Wertobjekte (1)

- Quellcode Klassendeklaration (datum.h):

```
#include <iostream>
```

```
class datum final
```

wegen final keine Unterklassen (ab C++11)

```
{
```

```
private:
```

```
    int tag, monat, jahr;
```

```
public:
```

```
    static datum heute();
```

```
    datum() = default; 
```

```
    datum(int t, int m, int j);
```

*Copy- und Move-Konstruktoren, Destruktor,
sowie Copy- und Move-Zuweisungen
ergänzt der Compiler automatisch*

```
    friend bool operator==(const datum&, const datum&);
```

```
    friend std::ostream& operator<<(std::ostream&, const datum&);
```

```
};
```

```
std::istream& operator>>(std::istream&, datum&);
```

Beispielprogramm Klasse für Wertobjekte (2)

- Quellcode Objektbenutzung (datumtest.cpp):

```
#include "datum.h"

int main() {
    std::cout << "Welches Datum ist heute [jjjj-mm-tt]? ";

    datum d; // Aufruf Default-Konstruktor
    if (!(std::cin >> d)) { // Aufruf operator>>(std::istream&, datum&)
        std::cerr << "Eingabefehler!\n";
        return 1;
    }

    datum heute = datum::heute(); // Aufruf Fabrikfunktion
    if (d == heute) { // Aufruf operator==(const datum&, const datum&) const
        std::cout << "Richtig, " << d << " ist das heutige Datum!\n";
    }
    else {
        std::cout << "Falsch, " << heute << " ist das heutige Datum, nicht " << d << "!\n";
    }
}
```

Aufruf operator<<(std::ostream&, const datum&)

Beispiel-Programm Klasse für Wertobjekte (3)

- Quellcode Klassenimplementierung (datum.cpp):

```
#include "datum.h"
```

```
...
```

```
datum datum::heute( )
```

```
{
```

```
    std::time_t t = std::time(0);
```

```
    std::tm *p = std::localtime(&t);
```

```
    return {p->tm_mday, p->tm_mon + 1, p->tm_year + 1900};
```

```
}
```

*Initialisierung des Rückgabewerts
(Konstruktoraufruf)*

```
datum::datum(int t, int m, int j)
```

```
: tag(t), monat(m), jahr(j) // Initialisierungsliste
```

```
{
```

```
    // Konsistenzpruefung (stark vereinfacht)
```

```
    if (t < 1 || t > 31 || m < 1 || m > 12)
```

```
    {
```

```
        throw std::invalid_argument( );
```

```
    }
```

```
}
```

*Objekt werfen, nicht Objektadresse
(deshalb ohne new)*

Beispielprogramm Klasse für Wertobjekte (4)

- Fortsetzung Quellcode Klassenimplementierung (*datum.cpp*):

```
bool operator==(const datum& lhs, const datum& rhs)
{
    return &lhs == &rhs || (lhs.tag == rhs.tag
                             && lhs.monat == rhs.monat
                             && lhs.jahr == rhs.jahr);
}

std::ostream& operator<<(std::ostream& os, const datum& d)
{
    return os << d.jahr
           << ' - ' << std::setw(2) << std::setfill('0') << d.monat
           << ' - ' << std::setw(2) << std::setfill('0') << d.tag;
}

std::istream& operator>>(std::istream& is, datum& d)
{
    ...
}
```

Beispielprogramm Klasse für Entitäten (1)

- Quellcode Klassendeklaration (termin.h):

```
#include "datum.h"
#include <string>
```

```
class termin final
{
```

```
private:
```

```
    datum wann;
```

```
    std::string was;
```

```
public:
```

```
    termin(const datum&, const std::string&);
```

```
    termin(const termin&) = delete;
```

```
    termin& operator=(const termin&) = delete;
```

```
    termin(termin&&) = delete;
```

```
    termin& operator=(termin&&) = delete;
```

```
    void verschieben(const datum&);
```

```
    datum get_datum() const;
```

```
    std::string get_beschreibung() const;
```

```
};
```

*Entitäten sollen
nicht kopierbar und
nicht verschiebbar sein*

Beispielprogramm Klasse für Entitäten (2)

- Quellcode Klassenimplementierung (termin.cpp):


```
#include "termin.h"

termin::termin(const datum &d, const std::string &s)
: wann(d), was(s)
{ }

void termin::verschieben(const datum &d){ this->wann = d; }
datum termin::get_datum() const { return this->wann; }
std::string termin::get_beschreibung() const { return this->was; }
```

- Quellcode Objektbenutzung (termintest.cpp):

```
...
termin pruefung = {datum::heute(), "Pruefung Systemprogrammierung"};
prueferKalender.eintragen(&pruefung);
kandidatenKalender.eintragen(&pruefung);
pruefung.verschieben({1, 4, 2040});
...
```

 *Initialisierung
(Konstruktoraufruf)*

datum(1, 4, 2040)

C++ eingebettete Klassen

Hilfsklassen, die nur gemeinsam mit einer anderen Klasse gebraucht werden, können in diese Klasse eingebettet werden:

```
class EnclosingClass {  
    ...  
    class NestedClass {  
        ...  
    }  
    ...  
}
```

NestedClass hat vollen Zugriff auf EnclosingClass, auch auf die privaten Teile, umgekehrt gilt das nicht
Java-Entsprechung: statisch eingebettete Klasse, aber dort wechselseitig voller Zugriff

Benutzung der Hilfsklassen:

außerhalb von EnclosingClass muss die eingebettete Klasse mit dem qualifizierten Namen **EnclosingClass :: NestedClass** angesprochen werden

ansonsten in der Benutzung keine Unterschiede zu gewöhnlichen Klassen

Beispiel-Programm eingebettete Klassen (1)

- Quellcode Klassendeklaration (*intlist.h*):

*intlist verwaltet ganze Zahlen
in einer einfach verketteten Liste*

```
class intlist final  
{  
private:
```

```
    class element;
```

```
    element *head;
```

```
public:
```

```
    intlist ( );
```

```
    ~intlist ( );
```

```
    intlist& insert(int);
```

```
    intlist(const intlist&) = delete;
```

```
    intlist& operator=(const intlist&) = delete;
```

```
    intlist(intlist&&) = delete;
```

```
    intlist& operator=(intlist&&) = delete;
```

```
    ...
```

*eingebettete Hilfsklasse für die Listenelemente
wird nur intern gebraucht, deshalb vollständige
Deklaration in intlist.cpp*

*Entitäten sollen
nicht kopierbar und
nicht verschiebbar sein*

Beispiel-Programm eingebettete Klassen (2)

- Fortsetzung intlist.h:

...

class iterator **final**

{

private:

element *current;

explicit iterator(element*);

public:

bool operator!=(const iterator&) const;

int& operator*() const;

iterator& operator++();

friend class intlist;

};

iterator begin();

iterator end();

};

*eingebettete Klasse für die Iteration
über die Listenelemente*



*Iteratoren werden in C++
wie Zeiger benutzt*


Fabriktfunktionen für Iteratoren (müssen so heißen)



Beispiel-Programm eingebettete Klassen (3)

- Quellcode Klassenimplementierung (*intlist.cpp*):

```
#include "intlist.h"
```

```
class intlist::element final   
{
```

```
    element *next;
```

```
    int n;
```



```
    element(element *e, int m) : next(e), n(m)  
    { }
```

```
    friend class intlist;
```

```
    friend class intlist::iterator; 
```

```
};
```


```
...
```


Beispiel-Programm eingebettete Klassen (4)

- Fortsetzung *intlist.cpp*:

```
intlist::iterator::iterator(element *e) : current(e)
{ }

bool intlist::iterator::operator!=(const iterator& i) const
{
    return this->current != i.current;
}

int& intlist::iterator::operator*() const 
{
    return this->current->n;
}

intlist::iterator& intlist::iterator::operator++( )
{
    this->current = this->current->next;
    return *this; 
}
```

Beispiel-Programm eingebettete Klassen (5)

- Fortsetzung *intlist.cpp*:

```
intlist::intlist() : head(nullptr)
{ }

intlist::~intlist()
{
    element *e = this->head;
    while (e != nullptr)
    {
        element *x = e;
        e = e->next;
        delete x;
    }
}

...
```

```
...

intlist& intlist::insert(int n)
{
    this->head
        = new element(this->head, n);
    return *this;
}

intlist::iterator intlist::begin()
{
    return intlist::iterator(this->head);
}

intlist::iterator intlist::end()
{
    return intlist::iterator(nullptr);
}
```

Beispiel-Programm eingebettete Klassen (6)

- Quellcode Klassenbenutzung (*listvar.cpp*):

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

int main()
{
    // Liste anlegen
    intlist list;
    list.insert(3814).insert(3635).insert(3442).insert(3421);

    // Liste ausgeben
    for (int n : list) {
        std::cout << n << std::endl;
    }

    for (auto i = list.begin(); i != list.end(); ++i) {
        std::cout << *i << std::endl; // i.operator*()
    }
}
```

- Unterklassen-Deklaration:

```
class Unterklassenname : public Oberklassenname
{
public:
    // zusätzliche und überschriebene Funktionen ...
private:
    // zusätzliche Daten ...
};
```

mehrere Oberklassen erlaubt
(Mehrfachvererbung)

bei einer **public**-Ableitung
sind alle öffentlichen Funktionen
der Oberklasse auch in der
Unterklasse öffentlich
(entspricht Java **extends**)

- Definition von Unterklassen-Konstruktoren:

```
Unterklassenname::Unterklassenname( )
: Oberklassenname( )
{
    ...
}
```

in der Initialisierungsliste muss ein
Oberklassen-Konstruktor aufgerufen werden
(fehlt der Aufruf, ergänzt der Compiler
einen Aufruf des Oberklassen-Defaultkonstruktors)
(entspricht Java **super()**)

C++ Vererbung: Polymorphie und dynamische Bindung

- nur Variablen vom Typ Zeiger auf Klasse oder vom Typ Klassenreferenz können in C++ **polymorph** sein: 

```
Klassenname *Objektzeiger; } erlauben auch Umgang mit  
Klassenname &Objektreferenz; } Objekten einer Unterklasse
```

- nur **virtual** markierte Memberfunktionen, können mit **dynamischer Bindung** aufgerufen werden:

```
class Klassenname  
{  
    ...  
    virtual Rückgabetyf Funktion(...);  
    ...  
};
```

*zu Memberfunktionen ohne **virtual**
gibt es in Java keine Entsprechung*

- der Destruktor einer Oberklasse muss immer **virtual** markiert werden, es sei denn, er hat das Zugriffsrecht **protected**
*es drohen sonst Speicherlecks beim **delete** über einen Zeiger auf Oberklasse*

Beispiel-Programm Unterklasse (1)

- Quellcode Unterklassendeklaration (ortstermin.h):

```
#include "termin.h"  
#include <string>
```

```
class ortstermin final : public termin  
{  
private:  
    std::string wo;
```

```
public:  
    ortstermin(const std::string&, const datum&, const std::string&);  
    std::string get_ort() const;  
};
```

*die Oberklasse darf in **termin.h**
nicht **final** markiert sein und
muss einen **virtual** markierten
Destruktor haben*

*weil die Oberklasse keine Copy- / Move-Konstruktoren und -Zuweisungen hat
(dort mit **= delete** unterbunden), erzeugt der Compiler die Funktionen auch
für die Unterklasse nicht*

Beispiel-Programm Unterklasse (2)

- Implementierungsdatei (ortstermin.cpp)

```
#include "ortstermin.h"
```

```
ortstermin::ortstermin(const std::string& t, const datum& d, const std::string& s)
: termin(d, s), wo(t)
{ }
```

```
std::string ortstermin::get_ort() const { return this->wo; }
```

- Quellcode Objektbenutzung (ortstermintest.cpp):

```
...
```

```
ortstermin ot{"G151", {11, 10, 2016}, "Uebung"};
```

```
ot.verschieben(datum::heute());
```

```
std::cout << ot.get_ort() << ',' << ot.get_datum() << ',' << ot.get_beschreibung() << '\n';
```

```
const termin& t = ot; // Referenzen sind polymorph
```

```
std::cout << t.get_beschreibung() << '\n';
```

```
...
```

} verschieben(...) wegen **const**
nicht aufrufbar und
get_ort() wegen Typ Oberklasse
nicht aufrufbar

C++ Vererbung: Schnittstellen (1)

C++ macht keinen prinzipiellen Unterschied zwischen Klassen und Schnittstellen (*beides `class`*).

- Schnittstellen-Deklaration:

```
class Schnittstellename
{
public:
    virtual ~Schnittstellename() = default;
    virtual Rückgabety1 Funktion1(...) = 0;
    ...
    virtual RückgabetyN FunktionN(...) = 0;
};
```

entspricht Java *interface*

der Destruktor und die Memberfunktionen müssen **public** und **virtual** deklariert sein
(*nur virtual-Funktionen werden mit dynamischer Bindung aufgerufen*)

der Destruktor muss eine leere Implementierung haben: **= default**

die Memberfunktionen haben keine Implementierung (*pure virtual function*): **= 0**

entspricht Java
abstract

C++ Vererbung: Schnittstellen (2)

- Schnittstellen implementiert man per Vererbung mit abgeleiteten Klassen:

```
class Klassenname : public Schnittstellenname
{
public:
    ... // Konstruktoren, Destruktor usw. nach Bedarf
    Rückgabety1 Funktion1( ... ) override;
    ...
    RückgabetyN FunktionN( ... ) override;
private:
    ...
};
```

*entspricht Java
implements*



die Klassen-Deklaration wiederholt alle Funktionssignaturen der Schnittstelle mit **override** statt **= 0**, der Zusatz **virtual** darf fehlen

Beispiel-Programm Schnittstelle (1)

- Quellcode Schnittstellendeklaration (uhr.h):

```
#ifndef UHR_H
#define UHR_H

class uhr
{
public:
    virtual ~uhr() = default;
    virtual void ablesen(unsigned& s, unsigned& m) const = 0;
    uhr(const uhr&) = delete;
    uhr& operator=(const uhr&) = delete;
    uhr(uhr&&) = delete;
    uhr& operator=(uhr&&) = delete;
protected:
    uhr() = default;
};

#endif
```

*Implementierungsdatei uhr.cpp
entfällt bei einer Schnittstelle*

*mit gelöschten Copy- / Move-
Konstruktoren und -Zuweisungen
werden alle Implementierungsklassen
automatisch zu nicht kopier- und
verschiebbaren Entitäten*

Beispiel-Programm Schnittstelle (2)

- Schnittstellenbenutzung (gruss.h)

```
#include "uhr.h"
#include <string>

class gruss final
{
public:
    explicit gruss(uhr *u);
    std::string gruessen();
    ...

private:
    uhr *u;
};
```

Polymorphie



*bei Verwendung
der Systemuhr
schlecht testbar*



- Schnittstellenbenutzung (gruss.cpp)

```
#include "gruss.h"

gruss::gruss(uhr *u)
: u(u) { }

std::string gruss::gruessen()
{
    int stunde, minute;
    this->u->ablesen(stunde, minute);

    if (6 <= stunde && stunde < 11)
        return "Guten Morgen";
    if (11 <= stunde && stunde < 18)
        return "Guten Tag";
    if (18 <= stunde && stunde <= 23)
        return "Guten Abend";
    throw std::string("Nachtruhe!");
}
```

*dynamische
Bindung*

Beispiel-Programm Schnittstelle (3)



- Quellcode abgeleitete Mock-Klasse (testuhr.h):

*eine Mock-Klasse implementiert eine Schnittstelle speziell für Testzwecke
(mock = engl. Attrappe)*

```
#include "uhr.h"

class testuhr final : public uhr
{
public:
    testuhr();

    void stellen(unsigned s, unsigned m); // erlaubt gezielte zeitabhängige Tests
    void ablesen(unsigned& s, unsigned& m) const override;

private:
    unsigned stunde;
    unsigned minute;
};
```


Beispiel-Programm Schnittstelle (4)

- Quellcode abgeleitete Mock-Klasse (testuhr.cpp):

```
#include "testuhr.h"
#include <stdexcept>

testuhr::testuhr( )
: stunde(0), minute(0) { }

void testuhr::stellen(unsigned s, unsigned m)
{
    this->stunde = (s + m / 60) % 24;
    this->minute = m % 60;
}

void testuhr::ablesen(unsigned& s, unsigned& m) const
{
    s = this->stunde;
    m = this->minute;
}
```

C++ Vererbung: Typanpassung

mit dem Operator `dynamic_cast<>` können Typanpassungen innerhalb einer Vererbungshierarchie formuliert werden, die zur Laufzeit geprüft werden (*funktioniert aber nur für Klassen, die `virtual` markierte Member enthalten*):

Beispiel:

```
class C : public A, public B { ... }
```

```
...
```

```
// Upcast von Unterklasse C nach Oberklasse A:
```

```
A *a = new C();
```

```
// Crosscast von Oberklasse A nach Oberklasse B:
```

```
B *b = dynamic_cast<B*>(a);
```

```
if (!b) ... // Fehlerbehandlung
```

```
// Downcast von Oberklasse B nach Unterklasse C:
```

```
C *c = dynamic_cast<C*>(b);
```

```
if (!c) ... // Fehlerbehandlung
```

C++: Vergleich mit Java

Java ist ursprünglich als Vereinfachung von C++ entstanden.

Einige wichtige Unterschiede:

- in C++ sind Klassen als Werttyp verwendbar, sind sogar vorrangig so gedacht
*deshalb Objekte nicht nur auf dem Heap, sondern auch auf dem Stack
und auch ineinander verschachtelte möglich
deshalb Copy- und Move-Konstruktoren sowie -Zuweisungsoperatoren in jeder Klasse*
- in C++ Operator-Overloading möglich
Operatoren können dadurch auf benutzerdefinierte Typen angewendet werden
- in C++ kein Garbage-Collector
*deshalb Operator **delete** zur Speicherfreigabe und in jeder Klasse ein Destruktor
und in neueren Versionen Bibliotheksklassen zur Kapselung von Zeigern (intelligente Zeiger)*
- in C++ können Klassen mehrere Oberklassen haben (Mehrfachvererbung)
*Achtung: nur **virtual** markierte Memberfunktionen haben dynamische Bindung,
das gilt insbesondere auch für den Destruktor*

C++: Index

Ausnahmen 5-7
Bjarne Stroustrup 5-1
C++98, C++11 5-1
class 5-15
Copy-Konstruktor 5-15,5-18,5-20,5-23
Copy-Zuweisung 5-15,5-22,5-23
Default-Konstruktor 5-15,5-18,5-20,5-24
delete, delete[] 5-8
Destruktor 5-14,5-15,5-21,5-23
dynamic_cast<> 5-8
friend 5-25
Function-Overloading 5-10
Initialisierungsliste 5-19
Intelligente Zeiger 5-9
Klasse 5-14,5-15,5-26,5-30,5-33,5-34,5-35
Konstruktor 5-14,5-15,5-18,5-19,5-20,5-23,5-24
Memberfunktion 5-14,5-15,5-16,5-17
Move-Konstruktor 5-15,5-18,5-20,5-23
Move-Zuweisung 5-15,5-22,5-23
Name-Mangling 5-10

Namensraum, **namespace** 5-2,5-12,5-13
new 5-8
Operator-Overloading 5-11
operator<< 5-2,5-11, 5-26,5-27,5-29
operator= 5-15,5-22
operator== 5-3,5-26,5-27,5-29
operator>> 5-2,5-26,5-27,5-29
Referenz 5-5,5-6
Rule of five 5-23,5-24
smart pointer 5-9
static 5-25
std::array<>, std::vector<> 5-4
std::cin, std::cout 5-2
std::string 5-3
std::unique_ptr 5-9
Stream 5-2
try-catch-throw 5-7
using 5-12
Zuweisungsoperator 5-15,5-22