# Systemprogrammierung

# Teil 7: Einführung in C++
## Referenzen, Operator-Overloading, Namensräume, Klassen

Prof. Dr. H. Drachenfels
Hochschule Konstanz

Version 12.0
1.8.2017

## 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 Standardbibliothek
  *(u.a. Ein-/Ausgabe-Klassen, String-Klasse, Vector-Klasse, intelligente Zeiger)*

ISO-Standards:

- C++98 von1998 *(mit Ergänzungen 2003 und 2007)*
- C++11 von 2011 *(mit Ergänzungen 2014)*

*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:

```
#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 sind aber ebenfalls nutzbar:*

```
#include <cstdio>
...
std::printf(...);
...
```



## C++ Speicherverwaltung: new und delete

In C++98 wird Heap-Speicher mit dem Operator **new** allokiert
und muss mit dem Operator **delete** wieder freigegeben werden:

```
#include <iostream>
int main()
{
    int *p = new int(1);
    std::cout << *p << '\n';
    delete p;

    int *a = new int[2];
    a[0] = 10;
    a[1] = 20;
    for (int i = 0; i < 2; ++i)
    {
        std::cout << a[i] << '\n';
    }
    delete[] a;
}
```

*einzelne ganze Zahl, mit 1 initialisiert*

*Feld von zwei ganze Zahlen, nicht initialisiert*

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

# C++ Referenzen: Definition und Verwendung

Eine Referenz definiert einen Aliasnamen für einen Speicherbereich.

- **Variablen-Definition**:

```
Typ  Name  =  Wert;
Typ  &Aliasname  =  Name;
```

*Initialisierung ist bei Referenzvariablen Pflicht*

*das & kennzeichnet die Variable als Referenz*

- **Verwendung**:

  als Parameter- und Rückgabetyp von Funktionen *(insbesondere überladene Operatoren)*
  als Parameter von catch-Blöcken

  *der Compiler realisiert Referenzparameter mit Zeigern:*

```
void function(const int &n)
{
    int m = n;
    ...
}
...
int k = 1;
function(k);
```

```
void function(const int *n)
{
    int m = *n;
    ...
}
...
int k = 1;
function(&k);
```



# C++ Ausnahmen: try-catch-throw

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

```
try
{
    if (...) throw MeineAusnahme();  // Stil: Objekt werfen, nicht Adresse!
    ...
    int *p = new int[SEHR_GROSSE_ZAHL];  // wirft evtl. std::bad_alloc
    ...
}
catch (std::bad_alloc&)  // Stil: Ausnahme per Referenz fangen (wegen Polymorphie)!
{
    std::cerr << "Zu wenig Speicher!\n";
}
catch (...)  // ... faengt beliebige Ausnahme
{
   ...
}
```

*new vermeiden*

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

## C++ Operator-Overloading: Beispiel

C++ erlaubt das Überladen von Operatoren für benutzerdefinierte Typen
*(wird unter anderem in der Ein-/Ausgabebibliothek verwendet)*.

- Beispiel:

```
#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 ...
}

namespace
{
    Deklarationen ...
}
```

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

*unbenannter Namensraum macht Deklarationen für andere Übersetzungseinheit unsichtbar*

*Java-Entsprechung:* `package Paketname;`

- Qualifizierung von Namen mit **Scope Resolution Operator**:

```
Namensraumname::EinName
```

- mit **Using-Direktive** auch Kurzschreibweise ohne Namensraumname:

```
using namespace Namensraumname;
EinName
```

*Java-Entsprechung:* `import Paketname.*;`

## Beispiel-Programm 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 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 Funktionen Zugriff auf diese Daten haben (**Kapselung**).
  *Die Daten heißen auch Attribute, Member-Daten oder Instanzvariablen.*
  *Die Funktionen heißen auch Operationen, Methoden oder Member-Funktionen (spezielle Funktionen sind die Konstruktoren, der Destruktor und überladene Operatoren).*
- jede Klasse hat Konstruktoren, darunter auch immer einen Copy-Konstruktor:
  jedes neue Objekt wird garantiert mit einem Konstruktoraufruf intialisiert
- jede Klasse überlädt den Zuweisungsoperator
- jede Klasse hat genau einen Destruktor:
  wird bei jedem Objekt vor seiner Zerstörung *(= Speicherfreigabe)* als letztes aufgerufen
  *Der Destruktor muss allen Speicher freigeben, der innerhalb der Klasse zusätzlich für das betreffende Objekt allokiert worden ist.*

# C++ Klassen: Syntax (1)

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

```
class Klassenname
{
public:
    Klassenname();                                  // Default-Konstruktor
    ~Klassenname();                                 // Destruktor
    Klassenname(const Klassenname&);                // Copy-Konstruktor
    Klassenname& operator=(const Klassenname&);     // Zuweisungsoperator

    Rückgabetyp_1 Methode_1(...);
    ...
    Rückgabetyp_N Methode_N(...);

private:
    Datentyp_1 Instanzvariable_1;
    ...
    Datentyp_M Instanzvariable_M;
};
```

*Copy-Konstruktor, Destruktor, Zuweisungsoperator und eventuell den Default-Konstruktor ergänzt automatisch der Compiler, wenn sie fehlen (bei C++11 kommt noch mehr hinzu)*



# C++ Klassen: Syntax (2)

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

```
Rückgabetyp_1 Klassenname::Methode_1(...)
{
    ... // Rumpf
}
...
```

- die Funktionen einer Klasse haben implizit einen zusätzlichen 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 Instanzvariablen über **`this`**:

```
this->Instanzvariable_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*`;`

- **Objekt-Benutzung**:

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

  *Objektname*`.`*Methode_1*`(...)`

  *Objektzeiger*`->`*Methode_1*`(...)`

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

*Klassenname*`::`*Methode_1*`(&`*Objektname*`, ...)`
*Klassenname*`::`*Methode_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 Rückgabetyp

  *eine Klasse darf mehrere Konstruktoren haben, wenn sie unterschiedliche Parameter haben*

- ein parameterloser Konstruktor wird als **Default-Konstruktor** bezeichnet:

  *Klassenname*`()`

  *wird eine Klasse ganz ohne Konstruktoren oder nur mit Copy-Konstruktor deklariert, erzeugt der Compiler implizit einen Default-Konstruktor, der für alle Instanzvariablen mit Klassen-Typ deren Default-Konstruktor aufruft*

- ein Konstruktor mit genau einem Parameter
  vom Typ konstante Referenz der Klasse wird als **Copy-Konstruktor** bezeichnet:

  *Klassenname*`(`**`const`** *Klassenname* `&)`

  *initialisiert neues Objekt als Kopie eines bestehenden Objekts*

  *wird eine Klasse ohne Copy-Konstruktor deklariert, erzeugt der Compiler implizit einen, der die Daten komponentenweise kopiert*

## C++ Klassen: Konstruktoren (2)

Für Konstruktor-Implementierungen gibt es zwei Stile:

- **Initialisierungsliste** im Methodenkopf *(bevorzugter Stil)*

```
Klassenname::Klassenname()
:  Instanzvariable_1(Wert_1),  ...,  Instanzvariable_M(Wert_M)
{  ...  }
```

- Zuweisungen im Methodenrumpf *(funktioniert nicht bei **const**-Variablen)*

```
Klassenname::Klassenname()
{
      Instanzvariable_1  =  Wert_1;
      ...
      Instanzvariable_M  =  Wert_M;
}
```

- Konstruktoren sollten unbedingt eine Ausnahme werfen, wenn sie ein Objekt nicht konsistent initialisieren können



## C++ Klassen: Konstruktoren (3)

Ein Konstruktor-Aufruf findet automatisch statt

- beim Gültigwerden einer Variablen mit Klassen-Typ:

```
Klassenname  objektname;                    // Default-Konstruktor
Klassenname  objektname(einArgument);       // Konstruktor mit Parameter
Klassenname  objektname = anderesObjekt;    // Copy-Konstruktor
```

*globale Variablen sind gültig von Programmstart bis -ende*

*lokale Variablen sind gültig vom Durchlaufen ihrer Definition bis zum Verlassen des umschließenden Anweisungsblocks*

- bei **new** mit einem Klassen-Typ:

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

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

```
aFunction(objektname);
return  objektname;
```

## 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: **`delete`** *objektzeiger*`;`



## Beispiel-Programm Klasse (1)

- Quellcode Klassendeklaration *(datum.h)*:

```
#include <iostream>
class datum
{
private:
     int tag, monat, jahr;
public:
     static datum heute();
     datum();
     datum(int t, int m, int j);
     datum(const datum&);
     ~datum();
     datum& operator=(const datum&);
     bool operator==(const datum&) const;
     friend std::ostream& operator<<(std::ostream&, const datum&);
};
std::istream& operator>>(std::istream&, datum&);
```

# Beispiel-Programm Klasse (2)

- Quellcode Objektbenutzung *(datumtest.cpp)*:

```
#include "datum.h"
int main() {
    datum heute = datum::heute();  // Aufruf Fabrikmethode und Copy-Konstruktor
    datum d;  // Aufruf Default-Konstruktor
    std::cout << "Welches Datum ist heute [jjjj-mm-tt]? ";
    if (!(std::cin >> d)) {  // Aufruf operator>>(std::istream&, datum&)
        std::cerr << "Eingabefehler!\n";
        return 1;
    }
    if (d == heute) {  // Aufruf datum::operator==(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 (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 datum(p->tm_mday, p->tm_mon + 1, p->tm_year + 1900);
}

datum::datum()
{ /* nicht initialisiertes Datum zulassen */ }

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***

## Beispiel-Programm Klasse (4)

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

```
datum::datum(const datum &d)
: tag(d.tag), monat(d.monat), jahr(d.jahr)  // Initialisierungsliste
{ }

datum::~datum()
{ }

datum& datum::operator=(const datum &d)
{
    if (this != &d)  // keine Selbstzuweisung?
    {
        this->tag = d.tag;
        this->monat = d.monat;
        this->jahr = d.jahr;
    }
    return *this;
}
```

*diese Implementierungen von Copy-Konstruktor, Destruktor und Zuweisungsoperator würde der Compiler auch automatisch erzeugen*



## Beispiel-Programm Klasse (5)

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

```
bool datum::operator==(const datum& that) const
{
    return this == &that || (this->tag == that.tag
                          && this->monat == that.monat
                          && this->jahr == that.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)
{
    ...
}
```

# C++ Klassen: Standard-Bibliothek (1)

Ausschnitt aus der Klasse **std::string** *(nach ISO-Standard noch komplizierter)*:

```
class string
{
public:
     string();                                    // Konstruktoren
     string(const string& str) ;
     string(const char *s);
     ~string();                                   // Destruktor
     string& operator=(const string& str);        // Zuweisungen
     string& operator=(const char *s );
     string& operator+=(const string& str);
     string& operator+=(const char *s);
     const char *c_str() const;                   // Datenabfragen
     unsigned length() const;
     const char& operator[](unsigned pos) const ;
     char& operator[](unsigned pos);
     ...
};
```



# C++ Klassen: Standard-Bibliothek (2)

Operatoren außerhalb der Klasse **std::string** *(nach ISO-Standard noch komplizierter)*:

```
// Verknüpfungen
string operator+(const string& s1, const string& s2);
...
// Vergleiche
bool operator==( const string & s1 , const string& s2 );
...
// Ein-/Ausgabe
istream& operator>>(istream& is, string& s);
ostream& operator<<(ostream& os, const string &s);
...
```

Anwendungsbeispiel:

```
#include <string>   // damit std::string bekannt ist
char buffer[10];
std::cin >> buffer;   // Risiko eines Pufferüberlaufs
std::string s;
std::cin >> s;        // string-Objekt und operator>> sorgen für genug Speicher
```

# C++ Klassen: Standard-Bibliothek (3)

Ausschnitt aus der Template-Klasse **std::vector<>** *(nach ISO-Standard noch komplizierter)*:

```
template <typename T> class vector
{
public:
     vector();
     vector(unsigned n, const T& value = T());
     vector(const vector<T>& v);
     ~vector();
     vector<T>& operator=(const vector<T>& v);
     unsigned size() const;
     void resize(unsigned n, T c = T());
     T& operator[](unsigned i);
     T& at(unsigned i);
     ...
};
template <typename T>
bool operator==(const vector<T>& v, const vector<T>& w);
...
```

Template-Parameter



# C++ Klassen: Standard-Bibliothek (4)

- zu fast jedem Typ kann ein Vektortyp abgeleitet werden:

```
#include <vector>  // damit std::vector<> bekannt ist

// Vektor von vier ganzen Zahlen, alle mit 0 initialisiert:
std::vector<int> iv(4);

// Vektor von zwei Strings, mit Leerstrings initialisiert:
std::vector<std::string> sv(2);
```

- ein Vektor kennt im Gegensatz zum Feld seine Länge:

```
for (unsigned i = 0; i < iv.size(); i++) ...
```

- Vektorzugriff per `[]` ohne oder per `.at()` mit Indexprüfung:

```
iv[2] = 1;    // std::vector<int>::operator[](&iv, 2) = 1;
iv.at(2) = 1; // std::vector<int>::at(&iv, 2) = 1;
```

- ein Vektor kann im Gegensatz zum Feld
  per Zuweisungs-Operator kopiert und per Vergleichsoperatoren verglichen werden

## Beispiel-Programm: std::string

```
#include <iostream>
#include <string>

int main()
{
    std::string a = "halli"; // a("halli")
    std::string s = "hallo"; // s("hallo")
    std::string t;  // leerer String

    // compare, copy and concatenate strings
    if (a < s) // operator<(a, s)
    {
        t = a + s; // t.operator=(operator+(a, s))
    }

    // print string values and addresses
    std::cout << a << '\n' << s << '\n' << t << '\n'; // operator<<(..., ...)
    std::cout << sizeof a << '\n' << sizeof s << '\n' << sizeof t << '\n';
    std::cout << a.length() << '\n' << s.length() << '\n' << t.length() << '\n';
}
```



## Beispiel-Programm: std::vector<>

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> a(4);
    a.at(0) = 3421;
    a.at(1) = 3442;
    a.at(2) = 3635;
    a.at(3) = 3814;

    // print vector values
    for (unsigned i = 0; i < a.size(); ++i)
    {
        std::cout << i << ": " << a[i] << '\n'; // a.operator[](i)
    }

    // print vector size
    std::cout << "sizeof a = " << sizeof a << '\n';
    std::cout << "a.size() = " << a.size() << '\n';
}
```

## C++ Vererbung: Syntax

- **Unterklassen-Deklaration**:

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

*bei einer **public**-Ableitung sind alle öffentlichen Methoden 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 Klassenreferenz können in C++ polymorph sein:

```
Klassenname *Objektzeiger;
Klassenname &Objektreferenz;
```

} *erlauben auch Umgang mit Objekten einer Unterklasse*

- nur Methoden, die **virtual** markiert sind, können mit dynamischer Bindung aufgerufen werden:

```
class Klassenname
{
      ...
      virtual Rückgabetyp Methode(...);
      ...
};
```

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

# C++ Vererbung: Schnittstellen (1)

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

- **Schnittstellen-Deklaration**:

```
class  Schnittstellenname
{
public:
     virtual  ~Schnittstellenname() { }
     virtual  Rückgabetyp1  Methode1(...) = 0;
     ...
     virtual  RückgabetypN  MethodeN(...) = 0;
};
```

*entspricht Java* ***interface***

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

der Destruktor muss eine leere Implementierung haben: **{ }**

*entspricht Java* ***abstract***

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



# C++ Vererbung: Schnittstellen (2)

- Schnittstellen implementiert man per Vererbung mit **abgeleiteten Klassen**:

```
class  Klassenname  : public  Schnittstellenname
{
public:
     // Konstruktoren, Destruktor usw. nach Bedarf
     Rückgabetyp1  Methode1( ... );
     ...
     RückgabetypN  MethodeN( ... );
private:
     ...
};
```

*entspricht Java* ***implements***

die Klassen-Deklaration wiederholt alle Methodensignaturen der Schnittstelle ohne **= 0**,
wobei der Zusatz **virtual** fehlen darf

## Beispiel-Programm Schnittstelle (1)

- Quellcode Schnittstellendeklaration *(Uhr.h)*:

```
class Uhr
{
public:
    virtual ~Uhr() { }
    virtual void ablesen(int *s, int *m = 0) = 0;
};
```

- Implementierungsdatei *(Uhr.cpp) entfällt*



## Beispiel-Programm Schnittstelle (2)

- Quellcode Implementierungsklasse *(SystemUhr.h)*:

```
class SystemUhr : public Uhr
{
public:
    void ablesen(int *s, int *m = 0);
};
```

- Quellcode weitere Implementierungsklasse *(TestUhr.h)*:

```
class TestUhr : public Uhr
{
public:
    TestUhr();
    void stellen(int s, int m);
    virtual void ablesen(int *s, int *m = 0);
private:
    int stunde;
    int minute;
};
```

```
#ifndef ...
#define ...
#include ...
    :
#endif
```

*aus Platzgründen weggelassen*

## Beispiel-Programm Schnittstelle (3)

- Quellcode Implementierungsklasse *(SystemUhr.cpp)*:

```
#include "SystemUhr.h"
#include <ctime>

void SystemUhr::ablesen(int *s, int *m)
{
     std::time_t t = std::time(0);
     std::tm *lt = std::localtime(&t);

     *s = lt->tm_hour;
     if (m) *m = lt->tm_min;
}
```



## Beispiel-Programm Schnittstelle (4)

- Quellcode Implementierungsklasse *(TestUhr.cpp)*:

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

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

void TestUhr::stellen(int s, int m)
{
     if (s < 0 || m < 0)
          throw std::invalid_argument("negative Stunde oder Minute");
     this->stunde = (s + m / 60) % 24;
     this->minute = m % 60;
}

void TestUhr::ablesen(int *s, int *m)
{
     *s = this->stunde;
     if (m) *m = this->minute;
}
```

## Beispiel-Programm Schnittstelle (5)

- Objektbenutzung *(Gruss.h)*

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

class Gruss
{
public:
     explicit Gruss(Uhr *u);
     std::string gruessen();

private:
     Uhr *u;
};
```

*Polymorphie*

*bei Verwendung von SystemUhr schlecht testbar!*

- Objektbenutzung *(Gruss.cpp)*

```
#include "Gruss.h"

Gruss::Gruss(Uhr *u)
: u(u)
  { }
std::string Gruss::gruessen()
{
     int stunde;
     this->u->ablesen(&stunde);
     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*



## C++: Vergleich mit Java

Java ist ursprünglich als Vereinfachung von C++ entstanden.

Einige wichtige Unterschiede:

- in C++ können Klassen mehrere Oberklassen haben *(Mehrfachvererbung)*
- 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-Konstruktor und Zuweisungsoperator 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++ bevorzugt generische Programmierung mit Templates
  *Templates bieten wesentlich mehr Möglichkeiten als die Generics von Java*

# C++: Lernzettel