

# Systemprogrammierung

## Teil 7: 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 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)

*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 Benutzeroberflächen)

# 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 <stdio>
...
std::printf( ... );
...
```

# C++ Speicherverwaltung: new und delete

In C++ 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); // einzelne ganze Zahl, mit 1 initialisiert
    std::cout << *p << '\n';
    delete p;

    int *a = new int[2]; // Feld 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++ Referenzen: Definition und Verwendung

Eine Referenz definiert einen Aliasnamen für einen Speicherbereich.

- **Variablen-Definition:**

*Typ Name = Wert;*

*Typ &Aliasname = Name; // & kennzeichnet eine Referenz-Variable*

- **Verwendung:**

als Parameter- und Rückgabety von Funktionen (*insbesondere überladene Operatoren*)

*der Compiler realisiert Referenz-Parameter 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++ 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);

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

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

# 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.*;`

## 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 ein Copy-Konstruktor:

jedes neue Objekt wird garantiert mit einem Konstruktor-Aufruf initialisiert

- 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 frei geben, 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

    Rückgabety_1 Methode_1(...);
    ...
    Rückgabety_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

```
Rückgabetyyp_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 private Instanzvariablen über **this**:

```
this->Instanzvariable_1 // Kurzschreibweise ohne this-> möglich
```

## C++ Klassen: Syntax (3)

---

- **Objekt-Erzeugung**  
durch Variablen-Definition mit Klasse als Typ:

```
Klassenname Objektname;
```

oder per Operator **new** auf dem Heap:

```
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 Rückgabotyp  
*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 (Speicherfreigabe) aufräumt.

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

```
~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 (Date.h):

```
class Date
{
public:
    Date();                // Default-Konstruktor
    Date(int d, int m, int y); // Konstruktor mit Parametern
    Date(const Date &d);    // Copy-Konstruktor

    ~Date();               // Destruktor

    Date &operator=(const Date &d);

    void print() const;

private:
    int day;
    int month;
    int year;
};
```

## Beispiel-Programm Klasse (2)

---

- Quellcode Objektbenutzung:

```
#include "Date.h"
#include <iostream>    // std::cerr ...
#include <stdexcept>   // std::invalid_argument

int main()
{
    try {
        Date d1;                // Aufruf Default-Konstruktor
        Date d2(1, 9, 2000);    // Aufruf Konstruktor mit Parametern
        Date d3 = d1;           // Aufruf Copy-Konstruktor: Date d3(d1);
        d3 = d2;                // Aufruf Zuweisungsoperator: d3.operator=(d2);
        d3.print();             // Aufruf Date::print
    } // Destruktor-Aufrufe: d3.~Date(); d2.~Date(); d1.~Date();
    catch (std::invalid_argument &) {
        std::cerr << "Falsches Datum\n" ;
    }
}
```

Ausnahme unbedingt  
per Referenz fangen!

## Beispiel-Programm Klasse (3)

- Quellcode Konstruktoren (*Date.cpp*):

```
Date::Date() // heimlicher Parameter: Date * const this
{
```

```
    std::time_t t = std::time(0);
    std::tm *p = std::localtime(&t);

    this->day = p->tm_mday
    this->month = p->tm_mon + 1;
    this->year = p->tm_year + 1900;
}
```

```
Date::Date(int d, int m, int y)
```

```
: day(d), month(m), year(y)
{
    if (d < 1 || d > 31 || m < 1 || m > 12) throw std::invalid_argument();
}
```

```
Date::Date(const Date &d)
```

```
: day(d.day), month(d.month), year(d.year)
{ }
```

Objekt werfen,  
nicht Objektadresse,  
deshalb ohne new

diese Implementierungen  
des Copy-Konstruktors  
würde der Compiler auch  
automatisch erzeugen

## Beispiel-Programm Klasse (4)

- Quellcode Destruktor, Zuweisung und Zugriffsfunktion (*Date.cpp*):

```
Date::~Date()
{ /* nichts zu tun */ }
```

```
Date& Date::operator=(const Date &d)
{
    if (this != &d) { // keine Selbstzuweisung?
        this->day = d.day;
        this->month = d.month;
        this->year = d.year;
    }
    return *this;
}
```

```
void Date::print() const
```

```
{
    std::cout << this->year
               << '-' << std::setw(2) << std::setfill('0') << this->month
               << '-' << std::setw(2) << std::setfill('0') << this->day;
}
```

diese Implementierungen von  
Destruktor und Zuweisung  
würde der Compiler auch  
automatisch erzeugen

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

Ausschnitt aus der Klasse **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 **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 **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);
...
```



## 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 (int 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; } erlauben auch Umgang mit
Klassenname &Objektreferenz; } 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 Schnittstellename
{
public:
    virtual ~Schnittstellename() { }
    virtual Rückgabety1 Methode1(...) = 0;
    ...
    virtual RückgabetyN 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: **{ }**

die Methoden 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 virtual Schnittstellename
{
public:
    // Konstruktoren, Destruktor usw. nach Bedarf
    Rückgabety1 Methode1( ... );
    ...
    RückgabetyN 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 (*Date.h*):

```
class Date
{
public:
    virtual ~Date() { }
    virtual void get(int *d, int *m, int *y) const = 0;
};
```

- Implementierungsdatei (*Date.cpp*) entfällt

## Beispiel-Programm Schnittstelle (2)

---

- Quellcode Implementierungsklasse (*CurrentDate.h*):

```
class CurrentDate : public virtual Date
{
public:
    void get(int *d, int *m, int *y) const;
};
```

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

```
class FixedDate : public virtual Date
{
public:
    FixedDate(int d, int m, int y);
    void get(int *d, int *m, int *y) const;
private:
    const int day;
    const int month;
    const int year;
};
```



## Beispiel-Programm Schnittstelle (3)

---

- Quellcode Implementierungsklasse (CurrentDate.cpp):

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

void CurrentDate::get(int *d, int *m, int *y) const
{
    std::time_t t = std::time(0);
    std::tm *p = std::localtime(&t);

    *d = p->tm_mday;
    *m = p->tm_mon + 1;
    *y = p->tm_year + 1900;
}
```

## Beispiel-Programm Schnittstelle (4)

---

- Quellcode Implementierungsklasse (FixedDate.cpp):

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

FixedDate::FixedDate(int d, int m, int y)
: day(d), month(m), year(y)
{
    if (d < 1 || d > 31 || m < 1 || m > 12)
    {
        throw std::invalid_argument("Falsches Datum");
    }
}

void FixedDate::get(int *d, int *m, int *y) const
{
    *d = day;
    *m = month;
    *y = year;
}
```

## Beispiel-Programm Schnittstelle (5)

- Quellcode Objektbenutzung:

```
#include "CurrentDate.h"
#include "FixedDate.h"
#include <iostream>
#include <stdexcept>

void printDate(Date *d)
{
    int day;
    int month;
    int year;
    d->get(&day, &month, &year);
    std::cout << day << '.'
              << month << '.'
              << year << '\n';
}

...
```

*Polymorphie*

*dynamische Bindung*

```
...
int main()
{
    try
    {
        Date *d;
        d = new CurrentDate();
        printDate(d);
        delete d;
        d = new FixedDate(1, 9, 2000);
        printDate(d);
        delete d;
    }
    catch (std::invalid_argument& x)
    {
        std::cout << x.what() << '\n';
    }
}
```

## 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 verschachtelt 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

---

Bjarne Stroustrup    Boost-Bibliotheken    C++    `class`    Copy-Konstruktor  
Default-Konstruktor    `delete`    `delete[]`    Destruktor    Initialisierungsliste  
intelligente Zeiger    Klassendeklaration    Methodendefinition    Namensraum  
`namespace`    `new`    Operator-Overloading    `operator<<`    `operator=`  
`operator>>`    `private:`    `public:`    pure virtual function    Qt    Referenz  
Schnittstellendeklaration    `std::cin`    `std::cout`    `std::string`    `std::vector<>`  
Stream    Template    Unterklassendeklaration    `using`    `virtual`