technische universität
dortmund

C++ Grundlagen

Jens Quedenfeld

02. April 2014

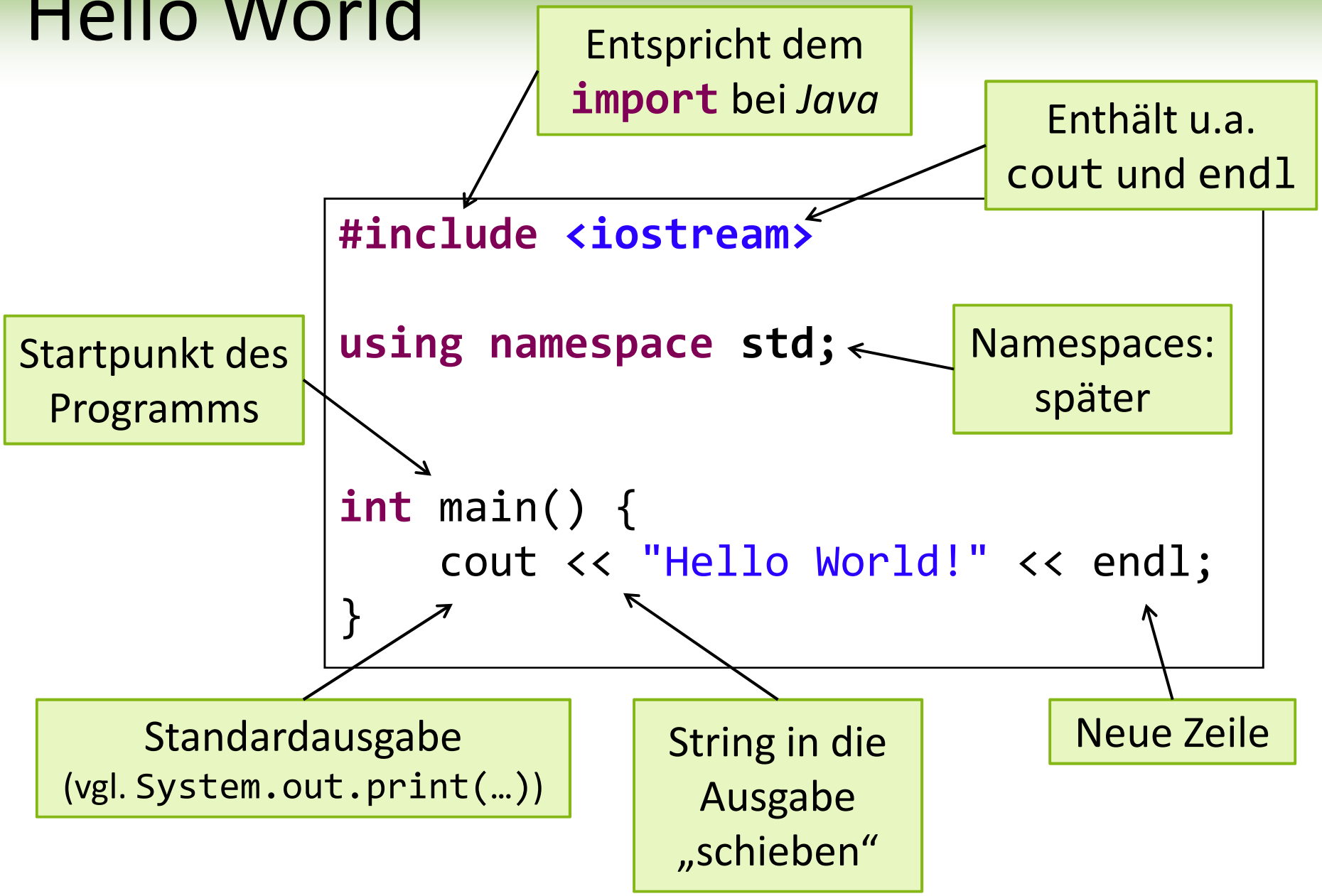
Inhalt

- Einführung
- Zeiger, Arrays und Referenzen
- Klassen und Objekte
- Templates
- Standardbibliothek
- Compiler, IDE, Profiling, Testing

C++

- Weitestgehend kompatibel zu C
- Objektorientiert wie *Java*
- Kontrollstrukturen wie in C/*Java* (`if`, `for`, `while`)
- Manuelle Speicherverwaltung (kein Garbage Collector)
- Designziel von C++: **Effizienz**

Hello World


Entspricht dem **import** bei *Java*
Enthält u.a. cout und endl
#include <iostream>
using namespace std;
Startpunkt des Programms
Namespaces: später
int main() {
cout << "Hello World!" << endl;
}
Standardausgabe (vgl. System.out.print(…))
String in die Ausgabe „schieben“
Neue Zeile

Zeiger

- **Adresse** einer Variable mit &

```
int i = 5;
cout << &i; // Ausgabe, z.B.: 0x28ff0c
```

- **Zeiger** (Pointer) = Variable, die Adresse speichert
  - Deklaration von Zeigern mit *

```
int* p = &i; // p = 0x28ff0c
```

- **Dereferenzierung** mit *

```
cout << *p; // Ausgabe: 5
*p = 6;     // i = 6;
cout << i;  // Ausgabe: 6
```

- Zeiger auf Zeiger

```
int** q = &p;
```

Zeiger

- Variablen (wie z.B. `int i;`) liegen auf dem **Stack**
  - Werden gelöscht, wenn Gültigkeitsbereich verlassen wird
  - Adressen können aber immer noch vorhanden sein:

Beispiel

```
int* foo() {
    int i = 5;
    return &i; // Adresse von i zurückgeben
} // i nach Verlassen nicht mehr vorhanden

void bar() {
    int* p = foo(); // p zeigt auf ungültige Adresse
    *p = 6; // Undefiniertes Verhalten
}
```

Dynamische Speicherverwaltung

- Variablen (wie z.B. `int i;`) liegen auf dem **Stack**
  - Adressen ungültig, wenn Funktion (etc.) verlassen wird
- **Lösung** Dynamische Speicherverwaltung
  - Mit `new` manuell Speicher auf **Heap** allokieren
  - Freigabe des Speichers mit `delete`

Beispiel

```
int* foo() {
    int* p = new int();
    *p = 5;
    return p;
}
```

```
void bar() {
    int* p = foo();
    *p = 6; // ok
    delete p; // Freigabe
}
```

Dynamische Speicherverwaltung

- **Lösung** Dynamische Speicherverwaltung
  - Mit **`new`** manuell Speicher auf **Heap** allokieren
  - Freigabe des Speichers mit **`delete`**
    - → Wichtig, sonst **Speicherlecks**
    - → **Kein** *Garbage Collector* wie bei *Java*
    - ⇒ Dadurch **performanter** als *Java*

Speicherlecks

```
void bar() {
    while (true) {
        int* p = foo(); // Allokation von 4 Byte
                        pro Aufruf von foo()
    } // Programm stürzt nach einigen Sekunden ab,
        da Speicher voll!
}
```

Arrays

- Konstante Länge

```
int a[10];    // Array der Länge 10
int b[3][3]; // 2-dimensionales Array der Größe 3x3
int c[n];     // nur erlaubt, wenn n konstant
```

- Dynamische Länge

```
int* d = new int[n];     // Array der Länge n
delete[] d;              // Speicher wieder freigeben
```

Arrays

- Zugriff auf Arrays wie bei Java

```
int a[10];      // Array der Länge 10
a[0] = 42;      // 0-tes Element bekommt Wert 42
a[10] = 42;     // Undefiniertes Verhalten
                // da Array nur 10 Elemente hat
```

- Arrays sind Zeiger auf das erste Element

```
*a = 5;         // a[0] = 5
*(a+1) = 6;     // a[1] = 6;
```

Arrays

Dynamische Speicherverwaltung mit **new**/**delete** ist **fehleranfällig**:

Beispiel

```
int memory_leak(int n) {
        int* array = new int[n];

        if (/*Sonderfall*/) {
                return 0; // Speicher-Leck
        }

        delete[] array;
        return n;
}
```

std::vector

- Gefahr bei **new**[] / **delete**[]: Speicher-Lecks
- Deshalb: ***std::vector*** aus der Standardbibliothek (STL)
- ***std::vector*** entspricht der ***ArrayList*** von *Java*
- Im Gegensatz zu *Java*:
  keine Geschwindigkeitsverluste gegenüber Arrays
- **std::vector** arbeitet intern mit **new**[] / **delete**[]
- Im *Destruktor* wird das intern verwaltete Array automatisch gelöscht (dazu später mehr)

  $\Rightarrow$ **keine** Speicher-Lecks

std::vector

Beispiel

```
#include <vector> // entspricht dem import bei Java

void vector_test() {
    std::vector<int> v(10); // vector mit 10 Elementen
    v[0] = 3;      // Zuweisung und
    int j = v[0]; // Zugriff wie bei Arrays!

    std::vector<int> w; // nicht std::vector<int> w();
    for (int i = 0; i < 5; i++)
        w.push_back(42); // automatische Vergrößerung
                         // entspricht add bei ArrayList
    w.size(); // Anzahl der Elemente: 5
} // Beim Verlassen wird der Speicher beider Vektoren
     automatisch freigegeben
```

std::vector

- Was ist, wenn eine Funktion ein **Array als Parameter** erwartet?

```
void function(int* array, int size);
```

- std::vector verwendet intern ein Array, also:

```
std::vector<int> vec;
function(&vec[0], vec.size());
```

Referenzen

Bereits kennengelernt Zeiger

Jetzt Referenzen

- Referenzen ≠ Zeiger
- Interne Zeiger auf Variablen
- „Alias“ für Variablen
- Deklaration mit **&** `int& ref`
- Referenzen müssen initialisiert werden
  - Referenzen können **nie** NULL sein
  - Referenzen nach Initialisierung **nicht** mehr änderbar
- Verwendung wie gewöhnliche Variablen
- Wofür braucht man dann Referenzen?

  → Häufig als Parametertyp für Funktionen verwendet

Beispiel

```
int i = 42;
int j = 43;
int& ref = i;
ref = 10; // i = 10
ref = j; // i = 43
// ref zeigt weiherhin auf i
```

```
int& ref = i;
int& ref;
int& ref = 5;
```

Referenzen

Call-by-Reference

Beispiel

```
void swap(int& a, int& b) {
    int t = a;
    a = b;
    b = t;
}

void main() {
    int x = 13;
    int y = 25;
    swap(x, y);
    cout << x << endl; // Ausgabe: 25
    cout << y << endl; // Ausgabe: 13
}
```

Funktionen

Übergabe von Parametern:

- **call-by-value**

```
→ void foo(std::vector<int> vec);
```

  → **Vorsicht** Explizite Kopie des Objekts

  → **Deshalb** nur bei kleinen Objekten (z.B. `int`)
  oder wenn sowieso eine Kopie erstellt wird

- **call-by-reference**

```
→ void foo(std::vector<int>& vec);
```

  → Zusätzlich **const**, falls Objekt nicht verändert wird:

```
→ void foo(const std::vector<int>& vec);
```

- **„call-by-pointer“**

```
→ void foo(std::vector<int>* vec);
```

  → Nur verwenden, wenn **NULL** als Wert erwünscht ist

Datentypen

- **Gleitkommazahlen** wie bei *Java*: `float`, `double`
- **Wahrheitswerte:** `bool` **(statt** `boolean`**)**
  - aber auch `int` (etc.) kann als Wahrheitswert genutzt werden:

```
if (myVector.size())
if (myVector.size() != 0)
```

} **äquivalent**

- **Ganze Zahlen:** `char`, `short`, `int`, `long`, `long long`
  - Länge ist plattformabhängig! In der Regel:

char	1 Byte
short	2 Byte
int	4 Byte
long long	8 Byte

  - Zusätzlich: `unsigned`
- **Konstanten**: `const` (bei *Java*: `final`)
  - Beispiel: `const unsigned long long N = 42;`

Strings

Zwei String-Repräsentationen in C++

- **`char*`**
  - Array fester Länge
  - String-Ende durch `\0` gekennzeichnet
  - Beim Verlängern: Gefahr von Pufferüberläufen
  - **Nicht** in C++ verwenden
- **`std::string`**
  - String erstellen:
  - Konkatenation:
  - Ausgabe:

```
std::string myString = "abc";
myString = myString + "def";
cout << myString;  // abcdef
```

  - Falls eine Funktion ein `char*` verlangt:

```
myString.c_str()
```

Main-Funktion

Einsprungspunkt in das Programm

- *Entweder*

```
int main()
```

- *Oder*

```
int main(int argc, const char** argv)
```

  - `argv` ist ein **Array** der Länge `argc` mit **Zeigern** vom Typ `const char*` (= null-terminierter String)
  - `argv[0]` ist gleich dem Programnamen (z.B. *abc.exe*)
  - Danach kommen die beim Programmstart übergebenen Parameter

- Rückgabetyp ist `int`, nicht `void`!
  - **Rückgabewert** `0` = erfolgreicher Programmlauf
    alles andere = Fehler, etc.
  - `return 0;` darf bei `main`-Funktion weggelassen werden

Eigene Klassen

Trennung von

Deklaration -> Person.h

```
#include <string>

#ifndef PERSON_H_
#define PERSON_H_

class Person {
private:
    std::string name_;
public:
    Person(const std::string& name);
    std::string getName() const;
};

#endif
```

Definition -> Person.cpp

```
#include "Person.h"

Person::Person(const std::string& name) :
    name_(name) {
}

std::string Person::getName() const {
    return name_;
}
```

Objekte erstellen

- Kein new verwenden!
  - `Person p("Emil");`
  - `Person q = Person("Emil"); // ok, evtl. langsamer`
  - `Person* r = new Person("Emil"); // Gefahr von Speicherlecks => vermeiden`
- Ohne Parameter
  - `Person p; // Konstruktor Person() wird aufgerufen`
  - **Nicht** `Person p(); // p wäre eine Funktion!`
  - **Aber** `Person q = Person(); // ok, evtl. langsamer`

Konstruktoren, Destruktoren

Folgende drei Elemente sind bei jeder Klasse implizit vorhanden

- **Kopier-Konstruktor**

```
Person(const Person& person)
```

  - Wird bei der Übergabe an eine Funktion ausgeführt (call-by-value)

```
void f(Person p);
//...
Person q;
f(q);
```

- **Destruktor**

```
~Person()
```

  - Wird ausgeführt, sobald Objekt zerstört wird

```
if (true) {
    Person p;
} // hier wird p zerstört
```

- **Zuweisungsoperator**

```
Person& operator=(Person& other)
```

  - Wird bei Zuweisungen ausgeführt

```
Person p;
Person q = p;
```

Dreierregel

- Wenn eine Klasse Kopierkonstruktor, Destruktor oder Zuweisungsoperator definiert, **dann auch die anderen beiden**
- Notwendig, wenn **new**/**delete** verwendet werden

Beispiel

```
class Class {
private:
    int length;
    int* array;
public:
    Class(int n) {
        length = n;
        array = new int[n];
    }
    ~Class() {
        delete[] array;
    }
    Class(const Class& other) {
        length = other.length;
        array = new int[length];
        for (int i = 0; i < length; i++)
            array[i] = other.array[i];
    }
    Class& operator=(const Class& other) {
        if (this != &other) {
            delete[] array; // altes Array löschen!
            length = other.length;
            array = new int[length];
            for (int i = 0; i < length; i++)
                array[i] = other.array[i];
        }
        return *this;
    }
};
```

Vererbung

Basisklasse

```
class Person {
protected:
    string name_;
public:
    Person(const string& name) :
        name_(name) {
    }
    void print() {
        cout << name_ << endl;
    }
};
```

Abgeleitete Klasse

```
class Student : public Person {
protected:
    int matnr_;
public:
    Student(const string& name, int matnr) :
        Person(name),
        matnr_(matnr) {
    }
    void print() {
        cout << name_ << ", ";
        cout << matnr_ << endl;
    }
};
```

Problem: Slicing

```
int main() {
    Student s("Fritz", 123);
    s.print();        // Ausgabe: Fritz, 123
    Person p = s;     // Slicing: matnr_ geht verloren!
    p.print();        // Ausgabe: Fritz
```

Vererbung

- **Slicing verhindern**

```
Student* s = new Student("Fritz", 123);
Person* p = s;
```

- **Ausgabe**

```
s->print(); // alternative Schreibeweise für
                    (*s).print();
p->print(); // ⟶ Ausgabe ist wieder: Fritz
```

- **Ursache**: `print()` ist nicht `virtual`
  - **Bisher** Sprungziel von `print()`-Aufruf wird beim Kompilieren festgelegt:
    - ⟶ `p` ist vom Typ `Person*`
    - ⟶ also wird bei `p->print()` die Methode `Person::print()` aufgerufen
  - **Wollen** Sprungziel zur Laufzeit bestimmen

Virtuelle Methoden

- **Wollen** Sprungziel zur Laufzeit bestimmen
- **Dazu** `print()` als **virtuelle** Methoden deklarieren

```
class Person { //...
    virtual void print() { //...
```

- **Jetzt** Gewünschte Ausgabe

```
p->print(); // Ausgabe: Fritz,123
```

- `Student::print()` ist dann ebenfalls **`virtual`**
- Bei *Java* sind alle Methoden implizit **`virtual`**
- Warum bei C++ nicht?
  - Objekte mit virtuellen Methoden benötigen zusätzlichen Zeiger

  ⇒ Höherer Speicherbedarf (4 oder 8 Byte)

  ⇒ Geringfügig höhere Laufzeit (für Sprungziel-Berechnung)

Vererbung

- Mehrfachvererbung möglich

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

- Abstrakte Methoden (pure virtual)

```
virtual void print() = 0;
```

Operator-Überladung

- In C++ können Operatoren überladen werden
  - Bereits gesehen, z.B. **`operator[]`** bei **`std::vector`**
    oder **`operator+`** bei **`std::string`**
  - Überladung sinnvoll, wenn Bedeutung intuitiv klar
- Überladung als Member-Funktion:

```
class Matrix {
    Matrix operator- (Matrix b);
};
```

- Überladung als freie Funktion

```
Matrix operator+ (Matrix a, Matrix b);
```

  - Falls auf **private** Attribute zugreifen soll:

```
class Matrix {
    friend Matrix operator+(Matrix a, Matrix b);
};
```

Templates

- Ähnlich wie **Generics** bei *Java*
- Bereits kennengelernt: **`std::vector`**
- Template-Argument können auch Basistypen sein
  - Bei *Java* **nicht** möglich: `ArrayList<int>`
  - Bei *C++* schon: `std::vector<int>`
- Realisierung bei *C++*:
  - Für jeden verwendeten Typ (**`int`**, **`double`**, …) wird die jeweilige Klasse (`std::vector`, …) separat kompiliert
  - Vorteile gegenüber *Java*: keine zusätzlichen Zeiger

    → bei Basistypen **speichereffizient** und schnell

Templates

- Klassen- und Funktions-Templates werden in der Header-Datei deklariert **und definiert**
- Beispiel:

Funktionstemplate

```
template <class T>
T add(T a, T b) {
    return a + b;
}
```

Benutzung des Templates

```
int main() {
    int x = 5;
    int y = 3;
    cout << add(x, y); // 8

    Person p, q;
    add(p, q); // Fehler beim
        // Kompilieren, da Person
}       // operator+ nicht überlädt
```

- Typüberprüfung findet beim Kompilieren statt

Iteratoren

- Alle Container-Klassen der STL besitzen die Methoden **`begin`**`()` und **`end`**`()`
- Vergleichbar mit **`Iterator`** und **`Iterable`** bei *Java*
- Zurückgegebene Objekte verhalten sich **wie Zeiger**

```
*(vec.begin() + 5); // Zugriff auf vec[5]
```

- **`begin`**`()` liefert Iterator, der auf den Anfang zeigt
- **`end`**`()` liefert Iterator, der hinter das Ende zeigt
- Beispiel-Anwendung: Sortieren

```
std::sort(vec.begin(), vec.end()); // Iteratoren
std::sort(&array[0], &array[N]);   // Zeiger
```

Iteratoren

- **Iteration über alle Elemente**

```
std::vector<int> vec;
std::vector<int>::iterator it = vec.begin();
        // oder kurz: auto it = vec.begin();
for (; it != vec.end(); ++it) {
    int value = *it;
    // ...
}
```

- **For-Each-Schleife**
  - Macht das Gleiche wie obiger Code:

```
for (int value: vec) {
    // ...
}
```

  - **Voraussetzung** Das Objekt `vec` besitzt die Methoden `begin()` und `end()`, die einen passenden Iterator zurückgeben

Typecasting bei C++

- C-Casts ähnlich wie bei Java:

```
double d = 4.53;
int i = (int) d; // i = 4
int j = int(d); // alternative Schreibweise
```

- C++-Casts
  - `static_cast<Class>(obj)`
    → verändert die Daten (Aufruf eines passenden Konstruktors)
  - `const_cast<Class>(const_obj)`
    → const „wegcasten“ (Objekt darf trotzdem nicht verändert werden)
  - `reinterpret_cast<int>(4.53f)`
    → lässt Daten unverändert (sehr hardware-nah)
  - `dynamic_cast<SubClass*>(base_obj)`
    → Überprüft zur Laufzeit, ob Cast zulässig (Vererbung)

Namespaces

- Große Programme bestehen aus vielen Klassen
  - → Gliederung notwendig
  - → bei *Java* **`package`**
  - → bei *C++* **`namespace`**
- Eigene namespaces

```
namespace My {
    class Person {/*...*/};
}
```

- Verwendung von namespaces

1. `My::Person p;`
2. `using My::Person;`
   `Person p;`
3. `using namespace My;`
   `Person p;`

Namespace

- Die Standardbibliothek befindet sich im Namespace **`std`**
- Verwende `using` **niemals** in Headerdateien!
  - Bei allen Dateien, die den Header inkludieren, wird der Namensraums ebenfalls „ausgeschüttet“

  → Dies ist unerwünscht
- Verschachtelte Namespaces
- Anonyme Namespaces
  - Sichtbarkeit nur innerhalb der Datei

```
namespace {
    int var; // globale Variable
}
```

Verschachtelung

```
namespace XY {
    namespace ABC {
        class Matrix;
    }
}
XY::ABC::Matrix m;
```

Datei-Zugriff

- Dateien lesen: **`ifstream`** `operator>>`
- Dateien schreiben: **`ofstream`** `operator<<`
- Lesen und schreiben: **`fstream`**
- Im Gegensatz zu *Java*:
  - Explizites `close()` **nicht notwendig**
  - Wird automatisch mit Destruktor ausgeführt

Beispiel: Datei schreiben

```
#include <fstream>

void write_file () {
    ofstream f("test.txt");
    f << "Dieser Text wird in die Datei geschreiben";
}
```

Standardbibliothek (STL)

Datenstrukturen

- `map<Key, Value>`
  - **Binärer Suchbaum**, Sortierung nach Key
  - Java-Äquivalent: `TreeMap<Key, Value>`
  - Siehe auch: `set<Value>`
- `unordered_map<Key, Value>`
  - **Hashtabelle**, Hash-Funktion muss selbst definiert werden
  - Java-Äquivalent: `HashTable<Key, Value>`
  - Siehe auch: `unordered_set<Value>` (= `HashSet<Value>`)
- `deque<Value>`
  - **Problem** bei `vector<Value>`
    - Zeiger auf Elemente werden bei Vergrößerung ungültig
  - Ebenfalls **konstante** Zugriffszeit
  - Zeiger auf Elemente bleiben **gültig**

Standardbibliothek (STL)

- <memory>
  - **Smart-Pointer**: **`unique_ptr`**, **`shared_ptr`**
- <random>
  - Verschiedene **Zufallsgeneratoren** und **Verteilungen**
  - Verwendet **nicht** die C-Funktion `rand()`
- <stringstream>
  - Umwandlungen von **`int`**, **`double`**, etc. in std::string und umgekehrt
- <cmath>
  - Mathematischen Funktionen (sin, cos, exp, log, …)

try-and-catch

- **Exceptions** auch bei C++
- Theoretisch kann alles geworfen werden (auch `int`)
  - Geworfenes Objekt sollte von `std::exception` erben

```
throw std::invalid_argument("Parameter xy ungültig!");
```

- Auffangen mit try und catch

```
try {
    // ...
} catch (std::exception& e) {
    std::cerr << e.what() << '\n'; // Fehlermeldung anzeigen
} catch (...) { // alles auffangen
    throw;     // und weiterwerfen
}
```

spezielle Syntax

Externe Bibliotheken

- Standardbibliothek von C++ ist verhältnismäßig **klein**
  - Z.B. keine Funktionen für Erstellen von Ordnern oder Navigation im Dateisystem
- **Boost**
  - Freie C++-Bibliothek, sehr umfangreich
  - Dateisystem, Speicherverwaltung, Datenstrukturen, Serialisierung, uvm.
- **Qt**
  - Grafische Benutzeroberflächen (plattformunabhängig)

IDEs, Compiler, Dokumentation

- Compiler
  - GCC (bzw. MinGW unter Windows)
  - MSVC++ (von Microsoft, nur für Windows)
  - uvm.
- IDEs
  - Microsoft Visual Studio
  - Eclipse CDT
  - Code::Blocks
- Dokumentation mit **Doxygen** (Syntax wie bei *JavaDoc*)

Hilfsprogramme

Valgrind

- Findet **Speicherlecks**
- Lässt Programm in virtueller Umgebung laufen
  - **Dadurch** 20-30 mal langsamer
  - **Vorteil**: Source-Code nicht nötig!
- Benutzung

```
valgrind --leak-check=yes myprog arg1 arg2
```

Ausgabe (Beispiel)

```
40 bytes in 1 blocks are definitely lost in loss record 1 of 1
     at 0x4C29703: operator new[](unsigned long) (vg_replace_malloc.c:384)
     by 0x40068F: f(int) (LeakTest.cpp:7)
     by 0x4006C7: main (LeakTest.cpp:14)
```

- Läuft nur unter Linux und Mac
- Alternative für Windows: *Dr. Memory*

Hilfsprogramme

Profiling mit ***gprof*** (gehört zum GCC-Compiler)

- Programm kompilieren (mit Parameter -pg)

```
g++ -o test.exe test.cpp -g -pg
```

- Programm ausführen

```
test.exe
```

→ Datei test.out wird erstellt (enthält Profiling-Informationen)

- Profiling-Daten anzeigen

```
gprof test.exe
```

Ausgabe (Beispiel)

```
  %    kumulativ   Selbst             Selbst   Gesamt
Zeit    seconds   seconds  Aufrufe us/Aufru us/Aufru  Name
80.59      1.23     1.23  1000000     1.23     1.23  g(int)
16.12      1.47     0.24  1000000     0.24     0.24  h(int)
 2.63      1.51     0.04  1000000     0.04     1.51  f(int)
 0.66      1.52     0.01                             main
```