Programmierkurs
C++ für Fortgeschrittene

SS 2011

Teil 2

Sandro Andreotti - andreott@inf.fu-berlin.de

THEMA 1
NAMESPACES

namespace

Namespaces kapseln Code:

```
namespace A
{
    void f() { std::cout << "A"; }
    char const * x;
    struct S { ... };
}

...

    A::f();
    A::x = "hallo";
    A::S x;
```

Metaphorisch gesprochen verhält sich Namespace zum Code wie Verzeichnis zur Datei: Genau wie in einer Verzeichnisstruktur sind Namespaces in einer baumartigen Struktur organisiert. Wenn man sich in einem Namespace aufhält, sind alle in diesem Namespace enthaltenen Symbole direkt erreichbar. Sub-Namespaces können wie Unterverzeichnisse qualifiziert werden (mit ::). Außerdem ist es möglich, vollständig zu Qualifizieren.

Ein Unterschied zum Dateisystem: Mit „using namespace“ kann man sich in mehreren Namespaces gleichzeitig aufhalten, das geht bei Verzeichnissen nicht.

Namespaces dürfen nicht innerhalb von Funktionen oder innerhalb von Klassendefinitionen definiert werden.

`namespace`, geschachtelt

```
namespace A
{
    void f() { std::cout << "A"; }

    namespace B
    {
        void f() { std::cout << "B"; }
    }
}

int main()
{
    A::f();     //Ausgabe: A
    A::B::f();  //Ausgabe: B
}
```

`namespace`, geschachtelt (II)

```
namespace A
{
    void f() { std::cout << "A"; }
    void g() { std::cout << "g"; }

    namespace B
    {
        void f() { std::cout << "B"; }
        void h()
        {
            g();        //Ausgabe: g
            f();        //Ausgabe: B
            ::A::f();   //Ausgabe: A
        }
    }
}
```

using namespace

Mit using namespace öffnet man einen Namespace für die direkte Benutzung:

```
namespace A
{
    void f() { std::cout << "A::f"; }
    void g() { std::cout << "A::g"; }
}
void f() { std::cout << "::f"; }

using namespace A;
...
    g();        // Ausgabe A::g
    f();        // Fehler: mehrdeutig
```

THEMA 2
TYPKONVERSION

Konversion von Typen

type conversion: »Interpretiere einen Ausdruck des Typs T1 so, als wäre er vom Typ T2.«

Verschiedene Möglichkeiten der Konversion:

1. Explizit: casts
2. Implizit:
   - standard conversion
   - user defined conversions:
     - constructors
     - assignment operators
     - conversion operators

Quiz 1: *casts*

Frage: Was wird ausgegeben?

```
struct A { int x; };
struct B { short int y; };
struct C: A, B { char z; };

C * ptrC = (C *) 10;
A * ptrA = (A *) ptrC;
B * ptrB = (B *) ptrC;

printf("%i, %i, %i",(int)ptrC,(int)ptrA,(int)ptrB);
```

Antwort: 10, 10, 14

(unter der Annahme, dass sizeof(int) = 4)

ptrC wird mit dem C-style-cast auf das in C enthaltene B-Objekt gecastet. Da vor dem B aber das A (mit sizeof(A) = 4) in C liegt, verschiebt sich der this-Pointer um 4 Bytes.

Hätten wir ptrC auf 0 statt auf 10 gesetzt, so wäre 0, 0, 0 ausgegeben worden. (Spezialbehandlung des Nullpointers)

Verschiedene *casts*

- `reinterpret_cast`: Pointer unverändert lassen und einfach nur uminterpretieren:

```
B * ptrB = reinterpret_cast<B *>(ptrC); //ptrB == 10
```

- `static_cast`: Verschiebt pointer bei Objektvererbung:

```
B * ptrB = static_cast<B *>(ptrC); //ptrB == 14
```

- `const_cast`: beseitigt `const`

```
int const x = 20;
int y = const_cast<int>(x);
```

Außerdem gibt es in C++ auch noch dynamic_cast, der eventuelle Laufzeitchecks vornimmt, um z.B. auch zu virtuell vererbten Basisklassen zu casten. Mit dynamic_cast werden wir uns zu einem anderen Zeitpunkt näher befassen.

(explicit) casts

1. conversion operator:

```
reinterpret_cast<T>(p);
static_cast<T>(p);
const_cast<T>(p);
dynamic_cast<T>(p);
```

2. cast expression:

```
(T) p;
```

3. functional notation:

```
T(p);
```

dynamic_cast werden wir noch im Zusammenhang mit virtuellen Klassen behandeln.

Die functional notation wird motiviert durch die Schreibweise bei der Verwendung von Konstruktoren bei Klassen: Wenn C eine Klasse ist, so kann ein C'tor der Form **C(p)** als *conversion* von **p** nach **C** aufgefasst werden kann. Aber auch in einfache Typen wie int oder float kann auf diese Weise gecastet werden:

int x = int(3.7); //cast eines double in einen int

Generell sollten die conversion operators (1) den C-style casts (2,3) vorgezogen werden, siehe http://www.acm.org/crossroads/xrds3-1/ovp3-1.html.

.

standard conversion

1. array ⇒ pointer

```
int a[20];
int * b = a;
char const * c = "ich bin ein string literal";
```

2. function ⇒ pointer

```
void f()
{
   printf("hallo\n");
}

void (* p)() = f;    //p is pointer to function
p();
```

[Literatur: Siehe ISO/IEC 14882, Kapitel 4 „Standard conversion“]

Anmerkung: String literale können vermittels *standard conversion* auch in nicht-const Zeiger verwandelt werden. Davon ist jedoch eher abzuraten, da dieses Feature im Standard (1998) als „deprecated“ gekennzeichnet wurde. [Siehe ISO/IEC 14882, Annex D]

standard conversion (II)

3. qualification conversion

Das Hinzufügen von zusätzlichen **`const`**.

```
int       *       *       x;

int       *       * const y1 = x;
int       * const *       y2 = x;
int       * const * const y3 = x;
int const * const *       y4 = x;
int const * const * const y5 = x;
```

Die vereinfachte Faustregel lautet: "Man darf **const** immer hinzufügen, aber nicht weglassen"

Genauer gilt: „Rechts neben dem ersten neuen **const** muss überall **const** stehen, mit Ausnahme der am weitesten rechts stehenden Position."

Wird an beliebiger Stelle ein **const** weggelassen, kann keine *standard conversion* durchgeführt werden.

standard conversion (III)

4. integral conversion

Ganzzahlentyp ⇒ Ganzzahlentyp

```
short int x = 10;
int y = x;
```

signed ⇔ **unsigned**

```
int x = 10;           //signed
unsigned int y = x;
signed int z = y;
```

enum ⇒ Ganzzahlentyp

```
enum { A, B, C, D } x = C;
char y = x;           //y = 2;
```

Die Verwandlung in **int** oder **unsigned int** wird auch „*integral promotion*“ genannt.

standard conversion (IV)

5. floating point conversion

Kommazahltyp ⇒ Kommazahltyp

```
float x = 10.0;
double y = x;
```

6. floating-integral conversion

Kommazahltyp ⇒ Ganzzahltyp

```
float x = -10.9;
int i = x;      //i = -10;
```

Ganzzahl ⇒ Kommazahltyp

```
int i = 16793195;
float x = i;    //x = 16793196;
```

Die Verwandlung in „double“ heißt auch „floating point promotion“.

Bei der Umwandlung von Fließkommazahl zur Ganzzahl werden die Nachkommastellen abgeschnitten. Bei der Konversion von **bool** gilt: **false** wird zu 0.0, **true** zu 1.0

Bei der Umwandlung von Ganzzahl zu Fließkommazahl kann es (wie gezeigt) zu Rundungsfehlern kommen.

Anmerkung: Die Umwandlung von Fließkommazahlen in Ganzzahlen und umgekehrt ist nicht trivial, da die binäre Darstellungen nicht die gleichen sind. Zum Glück werden solche Dinge gewöhnlich von der CPU unterstützt (bei der Pentium-Familie: Instruktionen FILD („integer load“) und FIST („integer store“))

standard conversion (V)

7. pointer conversion

Null ⇒ Null-Pointer

```
int * i = 0;
```

Pointer ⇒ void-Pointer

```
int * i = 0;          void * v = i;
int const * j = 0;    void const * u = j;
```

Pointer auf Klasse ⇒ Pointer auf Basisklasse

```
struct Base {};
struct Derived: Base {};
Derived * d = 0;
Base * b = d;
```

Die Konversion Pointer auf Klasse zu Pointer auf Basisklasse entspricht einem static_cast.

Ähnliche Regeln gelten auch für *pointer to member types*.

Eine ähnliche Regel wie bei Pointer auf Klasse => Pointer auf Basisklasse gilt auch bei Referenzen: Referenz auf eine Klasse => Referenz auf Basisklasse.

standard conversion (VI)

8. boolean conversion

Zahlen/**enum**-Werte/Pointer ⇒ **bool**

```
int * i = -1;
bool b = i;    // b == true
```

Wiese Konversion stellt sicher, dass Zahlen, enum-Variablen und beliebige Pointer auch als Testwerte in if-Anweisungen auftauchen können.

Beachte, dass folgender Code fehlerhaft ist:

```
struct C { };
C obj;
if (obj)   //Fehler: obj ist weder bool noch Zahl noch enum noch pointer
{ //...
}
```

Benutzerdefinierte Konversion

Benutzerdefinierte Konversion bei Klassen:

1. C'tor

   Typ ⇒ Klasse

2. assignment operator

   Typ ⇒ Klasse

3. conversion function

   Klasse ⇒ Typ

Alle drei Arten von Funktionen müssen (nicht statische) Member-Funktionen sein.

Konversion durch C‘tor

- Irgendwas ⇒ Klasse
- C‘toren werden *nicht* vererbt.

```
struct Complex
{
   Complex(double const x): m_r(x), m_i(0) { }

protected:
   double m_r, m_i;
};

Complex c(17.5);                  //explizit

void f(Complex const & c) { ... }
f(17.5);                          //implizit
```

Anmerkung zum Beispiel: Hier wird gezeigt, wie Basisklassen und Members initialisiert werden können, bevor der C‘tor der Klasse aufgerufen wird. Da es sich bei m_r und m_i jedoch im Beispiel um primitive Datentypen handelt, könnte hier die Initialisierung auch direkt im body des C‘tors erfolgen:

```
Complex(double const x)
{
    m_r = x;
    m_i = 0;
}
```

Anmerkung: Initialisierungen von Objekten werden stets über C‘tors und nicht über assignment-Operatoren ausgeführt (und das obwohl in der Initialisierung das Zeichen = auftaucht).

Faustregel: Argumentübergabe bei Funktionsaufruf ist eine Initialisierung (darum wird C‘tor verwendet anstatt eines assignment operators).

C‘toren können für implizite Konversionen verwendet werden, es sei denn, sie werden mit dem Keyword **explizit** deklariert.

assignment operators

- Umwandlung: Irgendwas ⇒ Klasse
- **`operator =`** werden *nicht* vererbt.

```
struct Complex
{
   Complex & operator = (double const x)
   {
      m_r = x; m_i = 0;
      return *this;
   }
   double m_r, m_i;
};

Complex c;
c = 17.5;      //Aufruf des operator =
```

Wenn für eine Klasse explizit kein **operator =** definiert wird, wird vom Compiler implizit ein „*copy assignment operator*“ angelegt.

Der **operator =** einer Basisklasse wird stets durch die **operator =** der abgeleiteten Klasse (u.U. den implizit vom Compiler erzeugten) verdeckt. So erklärt sich, dass der **operator =** faktisch nicht vererbt wird.

Assignment-Operatoren können nicht für implizite Konversionen verwendet werden. (Siehe unten)

conversion functions

- Umwandlung: Klasse ⇒ Irgendwas
- *conversion functions* werden vererbt.

```
struct Complex
{
  double r_m, r_i;
  operator double() { return r_m; }
};

Complex c;
double x = c;
double y = (double) c;
double z (c);
x = c;
```

Konversionsfunktionen können für implizite Konversionen verwendet werden. (Siehe unten)

Implizite Konversion

Beispiele für implizite Konversionen mit *conversion function*:

```
struct Complex
{
   operator double() { ... }
};

void f(float i) { ... }

Complex c;

f(c);
double x = c + 10;
if (c) { ... }
```

„Implizit“ bedeutet hier: Die Konversion wird vom Compiler nach Bedarf selbst vorgenommen, ohne dass dies ausdrücklich im Code angegeben ist (wie z.B. bei einem Cast). Im Gegensatz dazu wären casts „explizite“ Konversionen. Auch der direkte Aufruf eines C‘tors wäre eine explizite Konversion. C‘tors können jedoch auch zur impliziten Konversion verwendet werden (siehe oben).

Konversionsketten

Bei impliziter Konversion sind „Ketten“ (*sequences*) von Konversionen zulässig:

1. standard conversion sequence:

   0, 1, oder mehrere verschiedene Standard-Konversionen hintereinander.

2. user defined conversion sequence:

   Besteht aus:

   1. einer *standard conversion sequence*
   2. genau einer *user defined conversion*
   3. einer zweiten *standard conversion sequence*

Als user defined conversion kommen C‘toren, die nicht mit „**explizit**“ deklariert sind, sowie Konversionsfunktionen in Frage.

Nur eine *user defined conversion* !

Höchstens eine benutzerdefinierte Konversion wird bei der impliziten Konversion benutzt:

```
struct A
{
   operator int() { ... }
};

struct B
{
   operator A() { ... }
};

B b;
int i = b;  //Error!
```

Konversionen - Mehrdeutigkeiten

Beispiel:

```
struct B;
struct A
{
   A (B const &) { ... }
};

struct B
{
   operator A() { ... }
};

void f(A const & a) { ... }

B b;
f(b);     //Error!
```

[Literatur: Meyers, „Effective C++“, Item 26]

Konversionen - Mehrdeutigkeiten (II)

Beispiel:

```
void f(int);
void f(char);

double d = 6.02;
f(d);                    //Error
```

Merke: »Bei mehreren gleich guten Alternativen für implizite Konversion: Compilerfehler!«

Güte impliziter Konversionen

Reihenfolge von „gut“ zu „schlecht“:

1. keine Konversion
2. array ⇒ pointer, function ⇒ pointer
3. Ganzzahl zu **`int`**, Kommazahl zu **`double`**
4. restliche *standard conversion sequences*
5. *user defined conversion sequences*

Die Konversionen unter 3 werden auch *integral promotion* bzw. *floating point* promotion genannt.

Güte impliziter Konversionen (II)

Faustregel 1: »*qualification conversion*: Je weniger zusätzliche `const`, desto besser.«

Faustregel 2: »*pointer conversion*: Je kürzer in der Ableitungshierarchie, desto besser. Am schlechtesten ist `void*`.«

Die Faustregel 1 stimmt allerdings nicht zu 100%: Die const-Position ganz rechts (gleich vor dem Variablennamen) geht nicht in die Zählung ein.

Beispiel:

```
void f(int const * i) { cout << "A" };
void f(int * const i) { cout << "B" };

int * i;
f(i);              //Ausgabe "B"
```

Verständis-Quiz 1

Frage: Was wird ausgegeben?

```
void f(int const * * )              { cout << "A"; }
void f(int * const * const )        { cout << "B"; }
void f(int const * const * const ) { cout << "C"; }

int * const * i;
f(i);
```

Antwort: B

A scheidet aus, da dies keine Standardkonversion ist. C ist schlechter als B, da „mehr“ const hinzugekommen sind.

Verständnis-Quiz 2

Frage: Was wird ausgegeben?

```
struct A {};
struct B: A {};
struct C: B {};

void f(A * x) { cout << "A"; }
void f(B * x) { cout << "B"; }
void f(void *) { cout << "void *"; }

C c;
f(&c);
```

Antwort: „B“

Alle 3 Funktionen könnten durch Konversion aufgerufen werden. Konversion zu **B*** ist jedoch besser als zu **A*** (kürzere Ableitungskette), und Konversion zu **A*** ist besser als zu **void** *.

THEMA 3
FUNKTIONSÜBERLADUNG

Funktionsüberladung

Frage: »Mehrere überladene Funktionen mit gleichem Namen: Welche wird aufgerufen?«

```
void f(int x) { cout << "int"; }
void f(double x) { cout << "double"; }

f(10);    // Ausgabe "int"
f(10.0);  // Ausgabe "double"
```

Antwort: ...unzählige Regeln...

Wir beschränken uns auf die wichtigsten.

Ablauf *overload resolution* (1)

Schritt 1: »Erweitere Suche, bis eine Funktion gefunden wird!«

```
void f(int) { }      //wird aufgerufen
namespace N
{
    void g()
    {
       f(10);        //findet in N kein f
    }
}
```

„Koenig lookup“

Bei der argumentabhängigen Namenssuche („*Koenig lookup*“) wird auch in Namespaces gesucht, aus denen die Typen der Argumente der Funktion kommen:

```
namespace N
{
    struct A { };
    void f(int) { }
    void f(A &) { }
}

f(17);          //Error!
f(N::A());      //findet N::f(A &)
```

Koenig lookup ist z.B. sehr nützlich, wenn man Operatoren für Klassen schreiben möchte, welche in bestimmten Namespaces definiert sind. Operatoren werden nämlich häufig als globale Funktionen definiert. Gäbe es kein Koenig Lookup, so würde folgendes Programm scheitern:

```
namespace N
{
        class A{};
        A & operator ++ (A & obj) {}
}

int main()
{
        A a;
        ++ a; // finde operator ++ ueber Koenig lookup

        return 0;
}
```

Ablauf *overload resolution* (2)

Schritt 2: »Betrachte nur Kandidaten mit der richtigen Argumentzahl!«

```
void f(int i, double d) { }
void f(int i, ...) { }
void f(int i, char c, double x = 0.0) { }
template <typename T> void f(int i, T t) { }

f(10, 'a');  // 4 mögliche Kandidaten
```

Anmerkung: Es würde die 3. Funktion ausgeführt werden.

Ablauf *overload resolution* (2)

Wenn keine gefunden wird: Fehler!

```
void f(int) { }      //wird nicht aufgerufen
namespace N
{
    void f() { }     //passt nicht
    void g()
    {
       f(10);        //Error! Kenne nur N::f()
    }
}
```

Ein ähnliches Problem ergibt sich bei Funktionen in Basisklassen: Sobald eine abgeleitete Klasse eine Funktion mit gleichem Namen wie eine Funktion der Basisklasse definiert, wird die Basisklassen-Funktion nicht mehr gefunden.

Anmerkung: Gäbe es in **N** keine Funktion mit dem Namen **f()**, so würde der Compiler die Suche auf den globalen Namespace ausweiten, und erfolgreich die Funktion **::f(int)** finden.

`using`-Deklaration

`using` macht außerhalb liegende Definitionen gleichberechtigt.

```
void f(int) { }       //wird aufgerufen
namespace N
{
    void f() { }      //passt nicht
    using ::f;        //jetzt ist ::f(int) bekannt
    void g()
    {
       f(10);         //Ok
    }
}
```

Wiederum funktioniert dies auch im Falle von Basisklassen.
Bsp:

```
struct Base
{
    void f(int) {};
};

struct Derived: Base
{
    void f() {};
    using Base::f;
    void g()
    {
        f(10);   //ok
    }
};
```

Ablauf *overload resolution* (3,4)

Schritt 3: »Nimm von allen Kandidaten den besten!«
(Regeln: Siehe auf den nächsten Seiten...)

Schritt 4: »Prüfe, ob der gewählte Kandidat auch zugänglich ist!«

```
struct C
{
    public: static void f(int) { }
    private: static void f(double) { }
};

C::f(1.0); //Error
```

Der beste Kandidat (1)

Regel 1: »Je *kürzer* die notwendigen impliziten Konversionen, desto *besser* der Kandidat.«

```
void f(int *) {} //wird aufgerufen
void f(int const *) {}
void f(void *) {}

int i = 17;
f(&i);
```

Die erste der drei Funktionen wird aufgerufen.

Der beste Kandidat (2)

Regel 2: »Ellipsis (variadic function) ist *schlechter* als alle impliziten Konversionen.«

```
void f(char, void *) {} //wird aufgerufen
void f(char, ...) {}

int i = 17;
f(0, &i);
```

Wiederum wird die erste Funktion aufgerufen.

Der beste Kandidat (3)

Regel 3: »Je spezieller der Template-Ausdruck, desto *besser*.«

```
template <typename T>
void f(T &) {}                                  // (1)

template <typename T>
void f(vector<T> &) {}                          // (2)

template <typename T1, typename T2>
void f(vector<pair<T1, T2> > &) {}              // (3)


vector<pair<int, bool> > v;
f(v);                                           // call (3)
```

Der beste Kandidat (4)

Regel 4: »Templates sind *besser* als die meisten impliziten Konversionen.«

```
template <typename T>
void f(T *) {}                          // (1)

void f(char const *) {}                 // (2)

char const arr [] = "hallo";
f(arr);                                 // call (2)

char arr [] = "hallo";
f(arr);                                 // call (1)
```

Templates sind schlechter als: Keine Konversion, array=>pointer und function=>pointer Konversionen. Alle anderen impliziten Konversionen sind schlechter als Templates.

Quiz

Frage: Was wird ausgegeben?

```
struct Base
{
   void f(int *) { cout << "A"; }
};
struct Derived: Base
{
   using Base::f;
   void f(int * const) { cout << "B"; }
   void f(int const *) { cout << "C"; }
};

Derived obj;
int i;
obj.f(&i);
```

Antwort: B

Begründung: **Base::f** bleibt trotz using-Deklaration durch **f(int * const)** überlagert. Bemerke: Bei **f(int * const)** und **f(int *)** handelt es sich um den gleichen Funktionsprototypen („ein **const** in der rechtesten Position zählt nicht“).

1. Aufgabe: (PRÜFUNGSAUFGABE)

Schreiben Sie eine Klasse zum Rechnen mit komplexen Zahlen. Sorgen Sie dafür, dass zumindest folgende Operationen unterstützt werden:

-Konversion komplexe Zahl zu Zahl (Zahl = int, double, …; es wird der Realteil der kompexen Zahl übergeben).

-Konversion Zahl zu komplexe Zahl (der Imaginärteil der komplexen Zahl wird auf 0 gesetzt).

-Addition zwischen zwei komplexen Zahlen und zwischen einer komplexen Zahl und einer Zahl.

2. Aufgabe:

Definieren Sie verschiedene Symbole in einer Hierarchie von Namespaces, und untersuchen Sie folgende Fragen:

-Kann man einen einmal definierten Namespace nochmals definieren?

-Wie werden die Inhalte eines Namespaces "vererbt"? Formulieren Sie die dahinter stehende allgemeine Regel.

-Namespace **A** enthalte "**using namespace B**". Was passiert, wenn sowohl in **A** als auch in **B** Funktionen namens **f** definiert worden sind?

-Kann ein Namespace **A** Inhalte an einen Namespace **C** vererben, die **A** erst vermittels eines **using namespace B** zugänglich gemacht worden sind?

Aufgabe 3

Schreiben Sie Funktionen **stringType(str)**, die für verschiedene String-Typen **str** unterschiedliche Beschreibungen zurückliefert:

-ist **str** ein basic_string, so liefert **stringType(str)** "basic_string" zurück.

-ist **str** ein String Literal, so liefert **stringType(str)** "string literal" zurück.

-ist **str** ein vector, so liefert **stringType(str)** "vector" zurück.

-ist **str** ein vector von vectoren, so liefert **stringType(str)** "vector of vectors" zurück.

-in allen anderen Fällen liefert **stringType(str)** "others" zurück.