Modernes C++

...für Programmierer

Unit 09: Klassen 2

by

Dr. Günter Kolousek

7. November 2019

Überblick

- Vererbung
- Destruktor
- abstrakte Klassen und Methoden
- Interfaces
- Explizites Überschreiben
- Ableitung und Überschreiben verhindern
- `noexcept`
- Virtuelle Konstruktoren
- Spezielle Methoden

Vererbung

- `public`
  - `public` und `protected` ebenfalls in abgeleiteter Klasse `public` und `protected`
  - für Spezifikationsvererbung (Subtypbeziehung)!!!
- `protected`
  - `public` und `protected` in abgeleiteter Klasse `protected`
- `private`: default!
  - `public` und `protected` in abgeleiteter Klasse `private`
  - für Implementierungsvererbung
    - Implementierung von Komposition: Instanz der Basisklasse ist als Teil der Instanz der Subklasse zu sehen

Vererbung – 2

```
#include <iostream>  // inheritance1.cpp
using namespace std;
struct Person {
    string doing() { return ”nothing”; } };
struct Teacher : public Person {
    string name;
    Teacher(string name) : name{name} {};
    string doing() { return ”teaching”; } };
int main() {
    Person p;
    cout << p.doing() << endl;
    Teacher t{”ko”};
    cout << t.doing() << endl;
    p = t;  // slicing!
    // now: t converted to p => no ”name” any more!
    cout << p.doing() << endl; }
```

Vererbung – 2

```
#include <iostream>  // inheritance1.cpp
using namespace std;
struct Person {
    string doing() { return "nothing"; } };
struct Teacher : public Person {
    string name;
    Teacher(string name) : name{name} {};
    string doing() { return "teaching"; } };
int main() {
    Person p;
    cout << p.doing() << endl;
    Teacher t{"ko"};
    cout << t.doing() << endl;
    p = t;  // slicing!
    // now: t converted to p => no "name" any more!
    cout << p.doing() << endl; }
```

```
nothing
teaching
nothing
```

Vererbung – 3

```
#include <iostream>  // inheritance2.cpp
using namespace std;
struct Person {
    string doing() { return ”nothing”; } };
struct Teacher : public Person {
    string name;
    Teacher(string name) : name{name} {};
    string doing() { return ”teaching”; } };
int main() {
    Person p;
    cout << p.doing() << endl;
    Teacher t{”ko”};
    cout << t.doing() << endl;
    Person* pp{&t};
    cout << pp->doing() << endl; }
```

Vererbung – 3

```
#include <iostream>  // inheritance2.cpp
using namespace std;
struct Person {
    string doing() { return ”nothing”; } };
struct Teacher : public Person {
    string name;
    Teacher(string name) : name{name} {};
    string doing() { return ”teaching”; } };
int main() {
    Person p;
    cout << p.doing() << endl;
    Teacher t{”ko”};
    cout << t.doing() << endl;
    Person* pp{&t};
    cout << pp->doing() << endl; }
```

```
nothing
teaching
nothing
```

Vererbung – 4

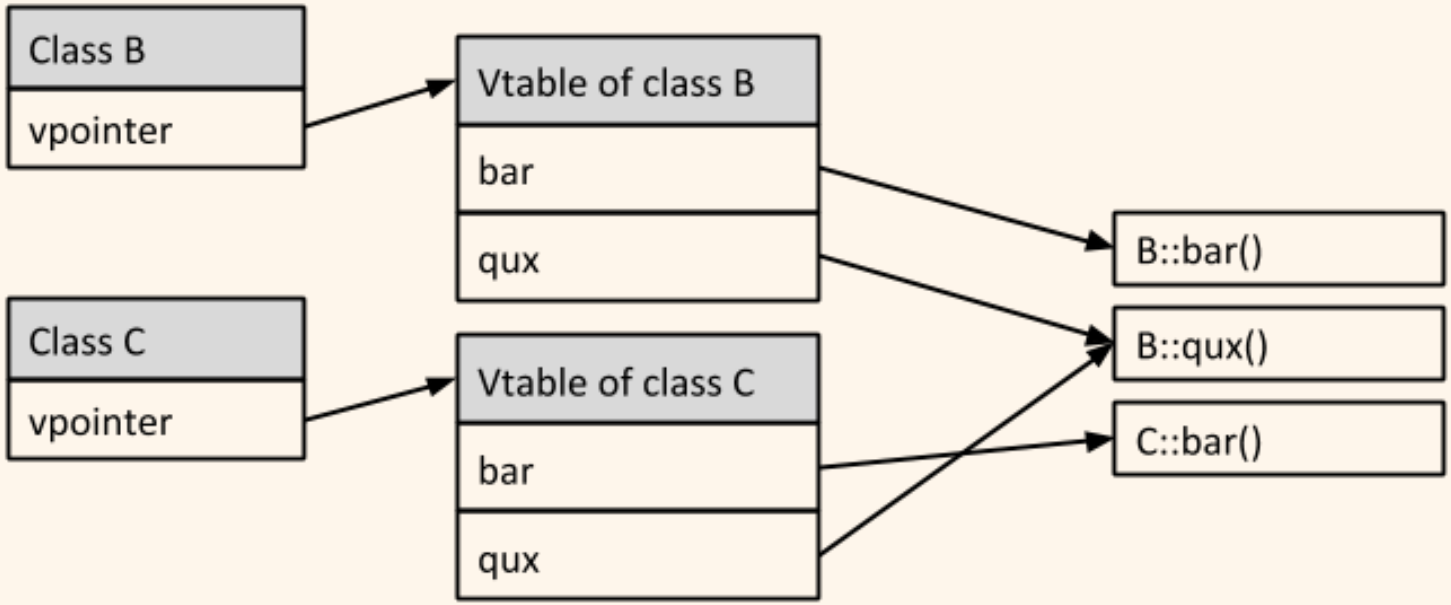
```
#include <iostream>  // inheritance3.cpp
using namespace std;
struct Person {
    virtual string doing() { return ”nothing”; } };
struct Teacher : public Person {
    string name;
    Teacher(string name) : name{name} {};
    string doing() { return ”teaching”; } };
int main() {
    Person p;
    cout << p.doing() << endl;
    Teacher t{”ko”};
    cout << t.doing() << endl;  // inline possible
    Person* pp{&t};
    cout << pp->doing() << endl;  }  // not inlined!
```

Vererbung – 4

```
#include <iostream>  // inheritance3.cpp
using namespace std;
struct Person {
    virtual string doing() { return "nothing"; } };
struct Teacher : public Person {
    string name;
    Teacher(string name) : name{name} {};
    string doing() { return "teaching"; } };
int main() {
    Person p;
    cout << p.doing() << endl;
    Teacher t{"ko"};
    cout << t.doing() << endl;  // inline possible
    Person* pp{&t};
    cout << pp->doing() << endl;  }  // not inlined!
```

```
nothing
teaching
teaching
```

Virtual function table (vtable)



Quelle: https://pabloariasal.github.io/2017/06/10/understanding-virtual-tables/

Vererbung – 5

```
#include <iostream>  // inheritance4.cpp
using namespace std;
struct Person {
    Person() { print_type(); }
    virtual void print_type() {
        cout << "Person" << endl; }
};
struct Teacher : public Person {
    void print_type() {
        cout << "Teacher" << endl; }
};
int main() {
    Teacher t{};
}
```

```
Person
```

Vererbung – 5

```
#include <iostream>  // inheritance4.cpp
using namespace std;
struct Person {
    Person() { print_type(); }
    virtual void print_type() {
        cout << "Person" << endl; }
};
struct Teacher : public Person {
    void print_type() {
        cout << "Teacher" << endl; }
};
int main() {
    Teacher t{};
}
```

```
Person
```

→ Objekt nicht fertig konstruiert!

Vererbung – 6

```cpp
#include <iostream>  // inheritance5.cpp
using namespace std;
struct Number {
    virtual void print_num(double d) {
        cout << "is double: " << d << endl; }
    virtual void print_num(int i) {
        cout << "is int: " << i << endl; }
};
struct SpecialNumber : public Number {
    void print_num(int i) {
        cout << "is special int: " << i << endl; }
};
int main() {
    SpecialNumber num;
    num.print_num(3.5);
}
```

```
is special int: 3
```

Vererbung – 6

```
#include <iostream>  // inheritance5.cpp
using namespace std;
struct Number {
    virtual void print_num(double d) {
        cout << "is double: " << d << endl; }
    virtual void print_num(int i) {
        cout << "is int: " << i << endl; }
};
struct SpecialNumber : public Number {
    void print_num(int i) {
        cout << "is special int: " << i << endl; }
};
int main() {
    SpecialNumber num;
    num.print_num(3.5);
}
```

```
is special int: 3
```

→ überschreiben… verschiedene Scopes!

Vererbung – 6

```cpp
#include <iostream>  // inheritance5.cpp
using namespace std;
struct Number {
    virtual void print_num(double d) {
        cout << "is double: " << d << endl; }
    virtual void print_num(int i) {
        cout << "is int: " << i << endl; }
};
struct SpecialNumber : public Number {
    void print_num(int i) {
        cout << "is special int: " << i << endl; }
};
int main() {
    SpecialNumber num;
    num.print_num(3.5);
}
```

```
is special int: 3
```

→ überschreiben… verschiedene Scopes! `virtual` dbzgl. nutzlos

Vererbung – 7

- Reihenfolge
  - der Konstruktoren: zuerst Basisklasse, dann aktuelle Klasse
  - der Destruktoren: ”umgekehrt”
- Konsequenzen von `virtual`
  - muss in Subklassen nicht angegeben werden
  - $\rightarrow$ `vtable` für jede Klasse und `vpointer` für jede Instanz!
    - `vtable` im read-only Teil des Prozesses
- Gründe `virtual` *nicht* zu verwenden:
  - Performance
  - Größe der Instanzen
  - Kontrolle, z.B.: `f()` ruft `g()` in Klasse X auf und `g()` ist nicht `virtual`, dann wird garantiert `X::g()` aufgerufen (und nicht ein `g()` einer Subklasse von X)
  - meist kein ”inlining” möglich
  - ”API-Probleme”

Vererbung – 8

- Container (der Standardbibliothek)
  - speichern Kopien
    - Vererbung $\rightarrow$ *slicing*

```
Person p;
Teacher t;
vector<Person> vp;
vp.push_back(p);
vp.push_back(t);  // !!!
```

  - Pointer
    - ok, aber nicht empfohlen
  - SmartPointer!

Vererbung – 9

```
#include <iostream>  // inheritance6.cpp
using namespace std;
struct A { int a;
    virtual void print(string msg) { cout << msg << ' '; }};
struct B : public A { int b; };
struct C : public A { int c; };
struct D : public B, public C {};  // -> dreaded diamond
int main() {
    D d;
    // d.print("Hi");  // error: mehrdeutig!
    d.B::print("Hi");
    // cout << d.a << endl;  // error: mehrdeutig!
    d.B::a = 1;    d.C::a = 2;
    cout << d.B::a << ' ' << d.C::a << endl;
    D* pd{new D};
    // cout << pd << ", " << static_cast<A*>(pd) << endl;
    //   error: mehrdeutig
}

Hi 1 2
```

Vererbung – 10

```
#include <iostream>  // inheritance7.cpp
using namespace std;
struct A { int a;
    virtual void print(string msg) { cout << msg << ' '; }};
struct B : public virtual A { int b; };
struct C : public virtual A { int c; };
struct D : public B, public C {};
int main() {
    D d;  d.print("Hi");  d.B::print("Hi");
    cout << d.a << endl;
    d.B::a = 1;    d.C::a = 2;
    cout << d.B::a << ' ' << d.C::a << endl;
    D* pd{new D};
    A* pa{static_cast<A*>(pd)};
    cout << pd << ", " << pa << ", " << (pd == pa) << endl;
}
```

```
Hi Hi 2047546882
2 2
0x55607a585ec0, 0x55607a585ee0, 1
```

Vererbung – 10

```
#include <iostream>  // inheritance7.cpp
using namespace std;
struct A { int a;
    virtual void print(string msg) { cout << msg << ' '; }};
struct B : public virtual A { int b; };
struct C : public virtual A { int c; };
struct D : public B, public C {};
int main() {
    D d;  d.print("Hi");  d.B::print("Hi");
    cout << d.a << endl;
    d.B::a = 1;    d.C::a = 2;
    cout << d.B::a << ' ' << d.C::a << endl;
    D* pd{new D};
    A* pa{static_cast<A*>(pd)};
    cout << pd << ", " << pa << ", " << (pd == pa) << endl;
}
```

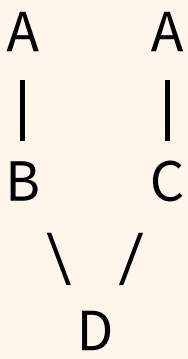
```
Hi Hi 2047546882
2 2
0x55607a585ec0, 0x55607a585ee0, 1
```

→ Compiler macht uns Glauben, dass gleich!!!

Vererbung – 11

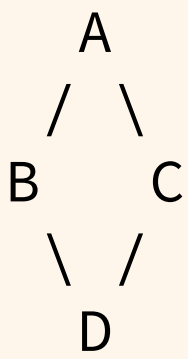
Speicherlayout

- nicht virtuell



  - B beinhaltet A
  - C beinhaltet A
  - D beinhaltet B und C

- virtuell



Vererbung – 12

```
#include <iostream>  // inheritance8.cpp
using namespace std;
struct A {
    virtual void foo()=0;
    virtual void bar()=0;
};
struct B : public virtual A {
    void foo() { bar(); }};
struct C : public virtual A {
    void bar() { cout << ”bar” << endl; }};
struct D : public B, public C {};
int main() {
    D d;  B& b{d};  C& c{d};
    d.foo();  b.foo();  c.foo();
}
```

```
bar
bar
bar
```

Vererbung – 12

```
#include <iostream>  // inheritance8.cpp
using namespace std;
struct A {
    virtual void foo()=0;
    virtual void bar()=0;
};
struct B : public virtual A {
    void foo() { bar(); }};
struct C : public virtual A {
    void bar() { cout << ”bar” << endl; }};
struct D : public B, public C {};
int main() {
    D d;  B& b{d};  C& c{d};
    d.foo();  b.foo();  c.foo();
}
```

```
bar
bar
bar
```

→ cross delegation…

Vererbung – 13

```cpp
#include <iostream>  // inheritance9.cpp
using namespace std;
struct A {
    int a;
    A() : a{42} {}
    A(int a) : a{a} {} };
struct B : public virtual A {
    int b;
    B() : A{0}, b{1} {} };
struct C : public B {
    int c;
    C() : c{2} {} };
int main() {
    C c;
    cout << c.a << ' ' << c.b << ' ' << c.c << endl;
}
```

```
42 1 2
```

Vererbung – 13

```cpp
#include <iostream>  // inheritance9.cpp
using namespace std;
struct A {
    int a;
    A() : a{42} {}
    A(int a) : a{a} {} };
struct B : public virtual A {
    int b;
    B() : A{0}, b{1} {} };
struct C : public B {
    int c;
    C() : c{2} {} };
int main() {
    C c;
    cout << c.a << ' ' << c.b << ' ' << c.c << endl;
}
```

```
42 1 2
```

(Default)Konstruktor von A wird aufgerufen beim Instanzieren von c

Vererbung – 13

```cpp
#include <iostream>  // inheritance9.cpp
using namespace std;
struct A {
    int a;
    A() : a{42} {}
    A(int a) : a{a} {} };
struct B : public virtual A {
    int b;
    B() : A{0}, b{1} {} };
struct C : public B {
    int c;
    C() : c{2} {} };
int main() {
    C c;
    cout << c.a << ' ' << c.b << ' ' << c.c << endl;
}
```

```
42 1 2
```

(Default)Konstruktor von A wird aufgerufen beim Instanzieren von c

→ daher selber aufrufen!

Vererbung – 14

```cpp
#include <iostream>  // inheritance9.cpp
using namespace std;
struct A {
    int a;
    A() : a{42} {}
    A(int a) : a{a} {} };
struct B : public virtual A {
    int b;
    B() : /* A{0}, */ b{1} {} };  // may stay...
struct C : public B {
    int c;
    C() : A{0}, c{2} {} };
int main() {
    C c;
    cout << c.a << ' ' << c.b << ' ' << c.c << endl;
}
```

```
0 1 2
```

Vererbung – 15

Quintessenz?

Vererbung – 15

Quintessenz?

Finger weg von so etwas!

`noexcept`

- `noexcept` $\equiv$ `noexcept(true)` $\rightarrow$ Zusicherung, dass Funktion keine Exception liefert
- `noexcept(false)` keine derartige Zusicherung
  - default, außer bei Destruktoren
- wenn Exception
  - wenn `noexcept`, dann: `terminate()`!
  - wenn `noexcept(false)`, dann: `terminate()`, wenn Exception nicht aufgefangen (im Zuge des *stack unwinding*)
- Destruktoren (implizit generiert oder nicht) sind default-mäßig `noexcept`!

noexcept - 2

```cpp
#include <iostream>  // noexcept.cpp
using namespace std;

class A {
  public:
    // ~A() {  terminate called after throwing an instance of
    ~A() noexcept(false) {
        throw 42;
    }
};

void f() { A a; }

int main() {
    try {
        f();
    } catch (...) {
        cout << "caught" << endl;
    }
}
```

Destruktor

```
#include <iostream>  // destructor.cpp
using namespace std;
struct Resource {
    ~Resource() {
        cout << ”dtor of Resource” << endl;
    }
};
struct Person {
};
struct Teacher : public Person {
    Resource r;
};
int main() {
    Person* tp{new Teacher{}};
    delete tp;
}
```

keine Ausgabe!

Destruktor

```cpp
#include <iostream>  // destructor.cpp
using namespace std;
struct Resource {
    ~Resource() {
        cout << "dtor of Resource" << endl;
    }
};
struct Person {
};
struct Teacher : public Person {
    Resource r;
};
int main() {
    Person* tp{new Teacher{}};
    delete tp;
}
```

keine Ausgabe!

→ Destruktor von Person wird aufgerufen (im Standard als UB)!

Destruktor – 2

```
#include <iostream>  // destructor2.cpp
using namespace std;
struct Resource {
    ~Resource() {
        cout << ”dtor of Resource” << endl;
    }
};
struct Person {
    virtual ~Person()=default;
};
struct Teacher : public Person {
    Resource r;
};
int main() {
    Person* tp{new Teacher{}};
    delete tp;
}
```

```
dtor of Resource
```

Destruktor – 3

```
#include <iostream>  // destructor3.cpp
using namespace std;
struct Person {
    ~Person() noexcept(false) { throw "error"; }
};
int main() {
    {
        Person p;
    }
    cout << "the end" << endl;
}
```

```
terminate called after throwing an instance of 'char const*'
```

erwartet, nicht wahr?

Destruktor – 4

```
#include <iostream>  // destructor4.cpp
using namespace std;
struct Person {
    ~Person() noexcept(false) { throw "error"; }
};
int main() {
    try {
        Person p;
    } catch (...) {
        cout << "caught" << endl;
    }
    cout << "the end" << endl;
}
```

```
caught
the end
```

Destruktor – 5

```cpp
#include <iostream>  // destructor5.cpp
using namespace std;
struct Person {
    ~Person() noexcept(false) { throw "error"; }
};
struct Team {
    Person p;
    ~Team() noexcept(false) { throw "error2"; }
};
int main() {
    try {
        Team t;
    } catch (...) {}
    cout << "the end" << endl;
}
```

```
terminate called after throwing an instance of 'char const*'
```

Destruktor – 5

```
#include <iostream>  // destructor5.cpp
using namespace std;
struct Person {
    ~Person() noexcept(false) { throw ”error”; }
};
struct Team {
    Person p;
    ~Team() noexcept(false) { throw ”error2”; }
};
int main() {
    try {
        Team t;
    } catch (...) {}
    cout << ”the end” << endl;
}
```

```
terminate called after throwing an instance of 'char const*'
```

`terminate` wird im Zuge des ”stack unwinding” aufgerufen, wenn währenddessen Exception geworfen wird!

Destruktor – 5

```cpp
#include <iostream>  // destructor5.cpp
using namespace std;
struct Person {
    ~Person() noexcept(false) { throw "error"; }
};
struct Team {
    Person p;
    ~Team() noexcept(false) { throw "error2"; }
};
int main() {
    try {
        Team t;
    } catch (...) {}
    cout << "the end" << endl;
}
```

```
terminate called after throwing an instance of 'char const*'
```

`terminate` wird im Zuge des "stack unwinding" aufgerufen, wenn währenddessen Exception geworfen wird! → *don't throw inside destructors!*

Destruktor – 7

- Destruktor *sollte* keine Exception werfen
- nicht `virtual` und `delete` auf Pointer zu Basisklasse $\rightarrow$ UB
- Faustregel: Destruktor einer *Basisklasse* sollte entweder
  - `virtual` und `public` (außer Klasse `final`) oder
  - nicht `virtual` und
    - `protected` sein: Anlegen einer Instanz einer abgeleiteten Klasse möglich (aber: kein `delete pBase` mehr, d.h. Anlegen nur am Stack)
    - `private` sein: generell kein Anlegen einer Instanz der *Basis*klasse möglich (aber: z.B. innerhalb als statische Variable in Klasse)

abstrakte Klassen und Methoden

```cpp
#include <iostream>  // abstract.cpp
using namespace std;

class Shape {  // abstract class!
  public:
    virtual string print()=0;  // pure virtual (abstract)!
    virtual ~Shape()=default;
};
class Circle : public Shape {
  public:
    string print() override {
        return "I am a circle";
    }
};
int main() {
    Shape* ptr{new Circle};
    cout << ptr->print() << endl;
}
```

```
I am a circle
```

Interfaces

Interfaces

- gibt es nicht

Interfaces

- gibt es nicht
- braucht man nicht

Interfaces

- gibt es nicht
- braucht man nicht
  - eine abstrakte Klasse mit nur abstrakten Methoden!

Interfaces

- gibt es nicht
- braucht man nicht
  - eine abstrakte Klasse mit nur abstrakten Methoden!
- eine Klasse implementiert viele Interfaces?

Interfaces

- gibt es nicht
- braucht man nicht
  - eine abstrakte Klasse mit nur abstrakten Methoden!
- eine Klasse implementiert viele Interfaces?
  - $\rightarrow$ Mehrfachvererbung…!

override

```
#include <iostream>  // override.cpp
using namespace std;

class IntCntr {
  public:
    virtual int incr(int i) {
        cout << "IntCntr" << endl; return i + 1; }
    virtual ~IntCntr()=default;
};
class LongCntr : public IntCntr {
  public:
    long incr(long l) {
        cout << "LongCntr" << endl; return l + 1; }
};
int main() {
    LongCntr lcntr{};
    IntCntr& icntr{lcntr};
    icntr.incr(1L);
}
```

```
IntCntr
```

override-2

```
#include <iostream>  // override2.cpp
using namespace std;
class IntCntr {
  public:
    virtual int incr(int i) {
        cout << "IntCntr" << endl; return i + 1; }
    virtual ~IntCntr()=default;
};
class LongCntr : public IntCntr {
  public:
    // override only applicable to virtual functions!
    long incr(long l) override {
        cout << "LongCntr" << endl; return l + 1; }
};
int main() {
    LongCntr lcntr{};
    IntCntr& icntr{lcntr};
    icntr.incr(1L);
}
```

→ long int LongCntr::incr(long int)« marked »override«, but does not override

override: nur wenn virtual!!!

final

```
#include <iostream>  // final.cpp
using namespace std;

class IntCntr {
  public:
    // final only applicable to virtual functions!
    virtual int incr(int i) final {
        cout << "IntCntr" << endl; return i + 1; }
    virtual ~IntCntr()=default;
};
class LongCntr : public IntCntr {
  public:
    virtual int incr(int i) override {
        cout << "LongCntr" << endl; return i + 1; }
};
int main() {
    LongCntr lcntr{};
    IntCntr& icntr{lcntr};
    icntr.incr(1L);
}
```

→ virtual function ‘virtual int LongCntr::incr(int)’ overriding final function

final – 2

```
#include <iostream>  // final.cpp
using namespace std;

class IntCntr final {
  public:
    virtual int incr(int i) final {
        cout << "IntCntr" << endl; return i + 1; }
    virtual ~IntCntr()=default;
};

class LongCntr : public IntCntr {
  public:
    virtual int incr(int i) override {
        cout << "LongCntr" << endl; return i + 1; }
};
int main() {
    LongCntr lcntr{};
    IntCntr& icntr{lcntr};
    icntr.incr(1L);
}
```

→ cannot derive from ‘final’ base ‘IntCntr’ in derived type ‘LongCntr’

Tipps

- Virtuelle Funktionen sollen genau einen der folgenden Spezifizierer haben:
  - `virtual`
  - `override`
  - `final`
- Verwende `virtual` nur, wenn es einen guten Grund dafür gibt
- Verwende `final` nur wenn notwendig
- Vermeide triviale Setter- und Getter-Methoden
  - aber beachte: API und ABI Abhängigkeiten!
    - `f(),g()` $\rightarrow$ `f(),g(),h()`

Tipps

- Virtuelle Funktionen sollen genau einen der folgenden Spezifizierer haben:
  - `virtual`
  - `override`
  - `final`
- Verwende `virtual` nur, wenn es einen guten Grund dafür gibt
- Verwende `final` nur wenn notwendig
- Vermeide triviale Setter- und Getter-Methoden
  - aber beachte: API und ABI Abhängigkeiten!
    - `f(),g()` $\rightarrow$ `f(),g(),h()` API & ABI ok!
    - `f(),g()` $\rightarrow$ `f(),g(int i=0)`

Tipps

- Virtuelle Funktionen sollen genau einen der folgenden Spezifizierer haben:
  - `virtual`
  - `override`
  - `final`
- Verwende `virtual` nur, wenn es einen guten Grund dafür gibt
- Verwende `final` nur wenn notwendig
- Vermeide triviale Setter- und Getter-Methoden
  - aber beachte: API und ABI Abhängigkeiten!
    - `f(),g()` → `f(),g(),h()` API & ABI ok!
    - `f(),g()` → `f(),g(int i=0)` API ✓, ABI ↯
- Vermeide `protected` Daten!
  - bricht Encapsulation und erhöht Abhängigkeit zwischen den Klassen

Virtuelle Konstruktoren

…gibt es in C++ nicht!

Virtuelle Konstruktoren

…gibt es in C++ nicht!

```
#include <iostream>  // virtualcons.cpp
using namespace std;
struct Person {
    virtual Person* clone() const=0;
    virtual Person* create() const=0;
    virtual ~Person()=default;
};
struct Teacher : public Person {
    Teacher* clone() const { return new Teacher{*this}; }
    Teacher* create() const { return new Teacher{}; }
};
int main() {
    Teacher t;
    Person& p{t};
    Person* pp{p.clone()};
    delete pp;
    pp = p.create();
    delete pp;
}
```

Spezielle Methoden

- Default-Konstruktoren
- Copy-Konstruktor
- Copy-Assignment-Operator
- Move-Konstruktor
- Move-Assignment-Operator
- Destruktor

Implizite Generierung

- Spezielle Methoden werden u.U. vom Compiler *implizit* generiert (dann als `noexcept`)
  - aber nur, wenn alle entsprechende Operationen der Instanzvariablen und der Superklassen `noexcept` sind.
- z.B. ein Copy-Konstruktor kann nur generiert werden, wenn die enthaltenen Instanzvariablen kopiert werden können.
- *user declared* spezielle Methode, wenn in Klassendeklaration in eine der folgenden Weisen angeführt:
  - mittels Definition
  - mittels `=default`
  - mittel `=delete`

Implizite Generierung – 2

- Wenn Compiler eine spezielle Methode nicht deklariert, dann wird diese auch nicht zum Überladen herangezogen.
  - Eine `deleted` spezielle Methode nimmt allerdings am Überladen teil.
  - Beispiel: `T o{std::move(other)};`
    - Copy Konstruktor vorhanden $\rightarrow$ kein Move Konstruktor vom Compiler $\rightarrow$ Copy Konstruktor wird zum Einsatz kommen.
    - Move Konstruktor `deleted` $\rightarrow$ Move Konstruktor wird vom Compiler gewählt $\rightarrow$ Compiler wird Fehler generieren.
- spezielle Regeln auf den folgenden Folien!

Implizite Konstruktoren

- Default-Konstruktor: wenn kein *user declared* Konstruktor
- Copy-Konstruktor: wenn kein *user declared* Copy-Konstruktor
  - aber: explizit `deleted`, wenn *user declared* Move-Konstruktor oder Move-Assignement-Operator
- Move-Konstruktor: wenn kein *user declared*
  - Copy-Konstruktor
  - Copy-Assignment-Operator
  - Move-Konstruktor
  - Move-Assignment-Operator
  - Destruktor

Implizite Zuweisungsoperatoren

- Copy-Assignment-Operator: wenn kein *user declared* Copy-Assignment-Operator
  - aber: explizit `deleted`, wenn *user declared* Move-Konstruktor oder Move-Assignement-Operator
- Move-Assignment-Operator: wenn kein *user declared*
  - Copy-Konstruktor
  - Copy-Assignment-Operator
  - Move-Konstruktor
  - Move-Assignment-Operator
  - Destruktor

Impliziter Destruktor

- wenn kein *user declared* Destruktor
  - `inline` und `noexcept`!

Rule of Five

Wenn eine der speziellen Elementfunktionen

- Destruktor
  - Wenn *user declared* Destruktor notwendig, dann werden compilergenerierte Copy-Konstruktor,… ”falsche” Ergebnisse liefern!
- Copy-Konstruktor
- Copy-Assignement-Operator
- Move-Konstruktor
- Move-Assignement-Operator

implementiert ist, dann sollen alle implementiert werden!

Rule of Three

Wenn eine der folgenden Elementfunktionen (member function)

- Destruktor
- Copy-Konstruktor
- Copy-Assignment-Operator

implementiert ist, dann sollen alle implementiert werden!

- vor C++ 11!
- d.h. heute veraltet

Rule of Zero

- Entwickle Klassen so, dass keiner der 5 speziellen Elementfunktionen implementiert werden müssen.
  D.h. Compiler wird die speziellen Elementfunktionen generieren!
- Faustregel: Immer *Rule of zero* verwenden!
  - Ausnahmen bestätigen die Regeln, siehe die folgenden Folien!

Beispiele

- Normale Klasse $\rightarrow$ Rule of zero
- Ressource Klasse

```cpp
#include <iostream>  // resource.cpp
using namespace std;
struct Resource {
    Resource() noexcept {}
    ~Resource() noexcept {}
    Resource(Resource&& o) noexcept {}
    Resource& operator=(Resource&& o) noexcept {
        return *this;
    }
};
int main() {
    Resource r;
    r = Resource{};
}
```

Beispiele – 2

- Container-Klasse: alle implementieren
- nicht verschiebbare Klasse

```
#include <iostream>  // immovable.cpp
using namespace std;
struct Immovable {
    Immovable() {}
    Immovable(const Immovable&)=delete;
    Immovable& operator=(const Immovable& o)=delete;
  // explicitly not copyable -> not moveable as well
  // in case you want to be explicit... feel free!
};
int main() {
    Immovable i;
    // i = Immovable{};  not possible any more
}
```