Abgeleitete Klassen

⟨Basis⟩ ⟶ ":" ⟨BasisSpec⟩ {"," ⟨BasisSpec⟩}
⟨BasisSpec⟩ ⟶ [⟨AccessSpec⟩] ["virtual"] ⟨ClassName⟩
⟨AccessSpec⟩ ⟶ "private" | "protected" | "public"

- Einschub von ⟨Basis⟩ nach Name in Vereinbarung einer Klasse
- Vereinbarte Klasse ist *abgeleitet* von ihren Basis-Klassen
- Abgeleitete Klasse *erbt* Komponenten von ihren Basis-Klassen
  Ausnahmen: Konstruktor, Destruktor, `operator=`
- Geerbte Komponenten *höchstens* so sichtbar wie ⟨AccessSpec⟩

Einfache Ableitung

```
class A {
  public:
    int x;
};
class B : public A {};

int main() {
  B b;
  b.x = 7;
  cout << b.x << endl;
}
```

Zugriffsattribut `protected`

- Zusätzliches Zugriffsattribut wie `private` und `public`
- Verwendbar in ⟨BasisSpec⟩ und Vereinbarung von Klassenkomponenten
- Verhalten i.W. wie `private`
- Erlaubt Zugriff in Komponenten abgeleiteter Klassen

Konstruktoren für abgeleitete Klassen

- Vererbte Komponenten werden mit Konstruktor der jeweiligen Basisklasse initialisiert
- Ausführung Konstruktoren aller Basisklassen vor Initialiserungen anderer Komponenten (in Reihenfolge der ⟨BasisSpec⟩)
- Konstruktoren der Basisklassen können ausgewählt werden/Parameter erhalten durch Erwähnung in Initialisierungsliste

Initialisierung bei abgeleiteten Klassen

```
class A { public:
  int x;
  A() : x(7) {}
  A(int x_) : x(x_) {} };
class B : public A { public:
  B() {}
  B(int x_) : A(x_ / 2) {} };
int main() {
  B b1{}; cout << b1.x << " ";
  B b2{7}; cout << b2.x << endl;
}
```

```
7 3
```

Überschattung von Klassenkomponenten

- Namen von Komponenten in abgeleiteten Klassen überschatten jene der Basisklassen
- Objekte abgeleiteter Klassen enthalten *Unterobjekte* der Basisklassen mit *allen* Komponenten
- Zugriff auf überschattete Namen mit scope resolution operator (`::`) angewandt auf *Komponentennamen*

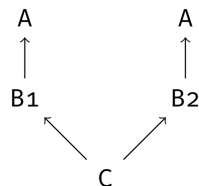
Überschattung in abgeleiteten Klassen

```
class A { public: int x; };
class B : public A {
  public: int x;
};

int main() {
  B b;
  b.x = 7; b.A::x = 3;
  cout << b.x << " " << b.A::x
       << endl;
}
```

```
7 3
```

Einschub: Visualisierung von Objekten abgeleiteter Klassen als DAG



Kanten entsprechen ist-Unterobjekt-von Beziehung

Mehrfach abgeleitete Klasse

```
class A { public: int x; };
class B1 : public A {};
class B2 : public A {};
class C : public B1, public B2 {};
```

Eindeutigkeit von Namen bei mehrfacher Ableitung

- Objekte abgeleiteter Klasse *C* mit mehrfachem Vorkommen gleicher Basisklasse *A* in Ableitungs-DAG enthalten mehrere Subobjekte von *A*
- `C.x` hier nicht zulässig
  - Kürzester Pfad zu Komponente gleichen Namens wird vorgezogen
  - Bei Gleichstand: Kompilerfehler
  - Sonst: Qualifikation mit `::` bis Pfad eindeutig

Verhalten mehrfach abgeleiteter Klasse

```
class A { public: int x; };
class B1 : public A {};
class B2 : public A {};
class C : public B1, public B2 {};

int main() {
  C c;
  c.B1::x = 3; c.B2::x = 7;
  cout << c.B1::x << " " << c.B2::x
       << endl;
}
```

```
3 7
```

Implizite Typkonvertierung von Objektzeigern/-referenzen

Sei A Basisklasse von B.

- Implizite Typumwandlung möglich: „Zeiger auf B“ $\longrightarrow$ „Zeiger auf A“, analog für Referenzen
- A priori keine implizite Typumwandlung zwischen Objekten – Indirektion (Zeiger/Referenz) ist notwendig

Implizite Typumwandlung abgeleiteter Klassen

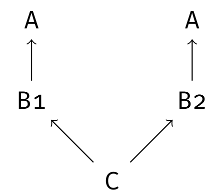
```
class A { public: int x; };
class B : public A {};

int main() {
  B b;
  b.x = 7;
  A& a = b;
  cout << a.x << endl;
}
```

```
7
```

Motivation: Virtuelle Basisklassen

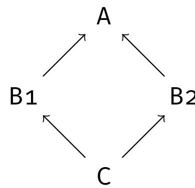
Semantische Bedeutung mehrere Vorkommnisse von `A`?
Mehrere *unabhängige* Weisen in denen `C` Anforderungen von `A` erfüllt, Verhalten von `A` implementiert.
Analogie: Motorsegler ist *sowohl* Segelschiff wie auch Motorschiff.



Oft jedoch nur eine Implementierung von `A`.
Analogie: Motorsegler ist trotzdem nur auf eine Weise ein Schiff.

Virtuelle Basisklassen

- Qualifikation `virtual` in ⟨BasisSpec⟩
- Für Objekte einer Klasse C, alle Klassen A mit letzter Kante `virtual` im Pfad von C zu A: nur *eine* Kopie von A in Objekten von C
- Wenn zusätzlich Pfade ohne `virtual` von C zu A: zusätzliche Kopien



Virtuell abgeleitete Klasse

```
class A { public: int x; };
class B1 : public virtual A {};
class B2 : public virtual A {};
class C
  : public B1, public B2 {};

int main() {
  C c;
  c.x = 7;
  cout << c.x << endl;
}
```

Exkurs: Implementierung virtueller Basisklassen

- Implementierungsdetails im Standard nicht vorgeschrieben
- Typumwandlung „Zeiger auf `C`“ ⟶ „Zeiger auf `A`“ soll möglichst geringe Kosten zur Laufzeit verursachen (Rechenzeit, Speicher, …)
- Daher: Objekte von `C` enthalten ihre Subobjekte von `A` in exakt gleicher Form (Speicherlayout) wie auch Objekte von `A` angeordnet sind
  ⟹ Typumwandlung allein durch Addition einer Konstante (Beginn des Subobjekts im Speicher des Objekts)
- Problem: Wenn Subobjekte von `B1` und `B2` beide eigentlich `A` enthalten sollen, `C` insgesamt aber nur ein `A`, dann Finden eines Speicherlayouts mit dieser Eigenschaft *unmöglich*
- Daher: (Sub-)Objekte von `B1`, `B2` enthalten Zahl (*offset*) wo Subobjekt von `A` jeweils startet.
  Objekte von `C` passen offsets in ihren Subobjekten an sodass sie zeigen auf selbes Subobjekt von `A`
- Nachschlagen von Zeigern *im Objekt selbst*, statt anhand des Typs: *virtual dispatch*

Virtuelle Methoden

- Methoden können mit `virtual` qualifiziert werden
- Auswahl unter überschattenden Methoden nicht anhand von Typ, sondern letztem auf Objekt angewandtem Konstruktor (Funktionszeiger für jede virtuelle Methode Teil des Objekts, werden im Konstruktor implizit gesetzt)

Polymorphe Objekte

```
class A { public:
  virtual char f() { return 'A'; }
};
class B : public A { public:
  virtual char f() { return 'B'; }
};
int main() {
  B b; A& ap = b;
  cout << b.f() << ap.f() << endl;
}
```

```
BB
```

`override`, `final`

- Methode mit selbem Namen und Parametertypliste in Basisklasse (transitiv) virtuell ⟹ Methode implizit virtuell
- `override`: muss Methode aus Basisklasse überschatten
- `final`: darf nicht überschattet werden

override und final

```
class A { public:
  virtual char f() { return 'A'; }
};
class B : public A { public:
  virtual char f() override final
    { return 'B'; }
};
int main() {
  B b; A& ap = b;
  cout << b.f() << ap.f() << endl;
}
```

```
BB
```

Fehlerquelle *object slicing*: Kopierkonstruktor, Zuweisungsoperator

Referenz als Parameter (Kopierkons., Zuw'operator) ⟶ implizite Typkonvertierung

Konvertierung bei Zuweisungsoperator

```
class A { public:
  virtual char f() { return 'A'; }
};
class B : public A { public:
  virtual char f() { return 'B'; }
};
int main() {
  B b; A a; A& ap = a; ap = b;
  cout << b.f() << ap.f() << endl;
}
```

```
BA
```

Konvertierung bei Kopierkonstruktor

```
class A { public:
  virtual char f() { return 'A'; }
};
class B : public A { public:
  virtual char f() { return 'B'; }
};
int main() {
  B b; A a = b;
  cout << b.f() << a.f() << endl;
}
```

```
BA
```

Fehlerquelle: Zuweisungsoperator

violated_invariant_assignment.cpp

```
#include <iostream>

using namespace std;

class A {
  protected:
    double x;

  public:
    A(double x_) : x(x_) {}
};

class B : public A {
  private:
    int approx_x;

  public:
    B(double x_) : A(x_), approx_x(x_) {}

    friend ostream& operator<<(ostream& stream, const B& b) {
      return stream << b.x << ” =~ ” << b.approx_x;
    }
};

int main() {
  B b1{3.0}, b2{7.0};
  A& ap = b1;
  ap = b2;

  cout << b1 << endl;
}
```

```
7 =~ 3
```

Rein virtuelle Methoden

- Spezielle Syntax: `= 0` statt Körper der Methode
- Keine Objekte von Klassen die rein virtuelle Methode nicht überschatten (auch nicht Parameter-, Ergebnistyp von Funktionen/Methoden)
- Zeiger und Referenzen jedoch erlaubt (auch als Parameter-, Ergebnistyp)
- Klasse mit min. einer rein virtuellen Methode: *abstract base class (ABC)*

Rein virtuelle Methode

```
class A { public:
  virtual char f() = 0;
};
class B : public A { public:
  virtual char f() { return 'B'; }
};
int main() {
  B b; cout << b.f() << endl;
}
```

```
B
```

Einschub: Vereinbarung von Klassenkomponenten als *explicitly defaulted*

- Sonst implizite Standardimplementierung von Kopierkonstruktor, Zuweisungsoperator, Destruktor, … kann explizit gefordert werden
- Spezielle syntax: `= default` statt Körper der Methode

Explizite Standardimplementierung

```
class A { public:
  A() = default;
  void f() { cout << "A" << endl; }
};

int main() {
  A a{}; a.f();
}
```

```
A
```

Einschub: Vereinbarung von Klassenkomponenten als *deleted*

- Sonst implizite Standardimplementierung von Kopierkonstruktor, Zuweisungsoperator, Destruktor, … kann explizit ausgeschlossen werden
- Spezielle syntax: `= delete` statt Körper der Methode

Gelöschter Kopierkonstruktor

```
class A {
  A(const A&) = delete;
};

int main() {
  A a1{};
  A a2{a1};
}
```

```
…
… error: use of deleted function
↪  'A::A(const A&)'
…
```

Motivation: virtuelle Destruktoren

Speicherleck durch falschen Destruktor

```
class A {};
class B : public A { public:
  double* b;
  B() { b = new double[100]; }
  ~B() { delete b; }

  void f() {
    for (int i = 0; i < 100; i++)
      cout << b[i] << endl;
  }
};

int main() {
  B* bp = new B{};
  for (int i = 0; i < 100; i++)
    bp->b[i] = i;
  bp->f();

  A* ap = bp;
  delete ap;
}
```

```
==…== Memcheck, a memory error detector
==…== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward
↪ et al.
==…== Using Valgrind-3.20.0 and LibVEX; rerun with -h for
↪ copyright info
==…== Command: destructor_leak_demo
==…==
==…==
==…== HEAP SUMMARY:
==…==     in use at exit: 800 bytes in 1 blocks
==…==   total heap usage: 4 allocs, 3 frees, 77,608 bytes
↪ allocated
==…==
==…== 800 bytes in 1 blocks are definitely lost in loss record 1
↪ of 1
==…==    at 0x48440F3: operator new[](unsigned long) (in …)
==…==    by 0x4012CF: B::B() (destructor_leak_demo.cpp:8)
==…==    by 0x4011BD: main (destructor_leak_demo.cpp:18)
==…==
==…== LEAK SUMMARY:
==…==    definitely lost: 800 bytes in 1 blocks
==…==    indirectly lost: 0 bytes in 0 blocks
==…==      possibly lost: 0 bytes in 0 blocks
==…==    still reachable: 0 bytes in 0 blocks
==…==         suppressed: 0 bytes in 0 blocks
==…==
==…== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
↪ from 0)
```

Virtuelle Destruktoren

Aufruf von Destruktor mit `delete`-Operator (oder auch implizit) wählt Destruktor nach selben Regeln wie Methode.
⟹ Destruktoren können `virtual` sein

Beispiel: virtuelle Destruktoren

Kein Speicherleck durch virtuellen Destruktor

```
class A { public: virtual ~A() = default; };
class B : public A { public:
  double* b;
  B() { b = new double[100]; }
  ~B() { delete[] b; }

  void f() {
    for (int i = 0; i < 100; i++)
      cout << b[i] << endl;
  }
};

int main() {
  B* bp = new B{};
  for (int i = 0; i < 100; i++)
    bp->b[i] = i;
  bp->f();

  A* ap = bp;
  delete ap;
}
```

```
==…== Memcheck, a memory error detector
==…== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward
↪ et al.
==…== Using Valgrind-3.20.0 and LibVEX; rerun with -h for
↪ copyright info
==…== Command: virtual_destructor_demo
==…==
==…==
==…== HEAP SUMMARY:
==…==     in use at exit: 0 bytes in 0 blocks
==…==   total heap usage: 4 allocs, 4 frees, 77,616 bytes
↪ allocated
==…==
==…== All heap blocks were freed -- no leaks are possible
==…==
==…== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
↪ from 0)
```