4 Abgeleitete Klassen (derived classes)

4.1 Konzept

- Bei der Definition einer Klasse kann man eine oder mehrere *Basisklassen* (base classes) angeben, von denen die neue Klasse *abgeleitet* wird.
- Zusätzlich zu ihren eigenen Elementen, besitzt die abgeleitete Klasse automatisch alle Datenelemente und Elementfunktionen ihrer Basisklassen („Vererbung“).
- Jeder Konstruktor der abgeleiteten Klasse ruft entweder explizit (durch entsprechende Elementinitialisierer) oder implizit Konstruktoren aller Basisklassen auf, bevor die eigenen Elemente der Klasse initialisiert werden.
- Die Reihenfolge dieser Konstruktoraufrufe entspricht der *Reihenfolge der Basisklassen* in der Deklaration der abgeleiteten Klasse, die von der Reihenfolge der Elementinitialisierer abweichen könnte.
- Der Destruktor der abgeleiteten Klasse ruft nach Ausführung seines Rumpfs implizit die Destruktoren aller Elemente und Basisklassen auf.
- Die Reihenfolge dieser Destruktoraufrufe entspricht der *umgekehrten Reihenfolge* der entsprechenden Konstruktoraufrufe.

Beispiel

```
struct Person {           // Person.
  string name;            // Name.
  Person (string n) : name{n} {}
};

struct Student : Person {       // Student.
  int number;                   // Matrikelnummer.
  Student (string n, int m) : Person{n}, number{m} {}
};

struct Employee : Person {      // Arbeitnehmer.
  int number;                   // Personalnummer.
  Employee (string n, int p) : Person{n}, number{p} {}
};

struct EmployedStudent : Student, Employee { // Teilzeitstudent.
  EmployedStudent (string n, int m, int p)
                          : Student{n, m}, Employee{n, p} {}
};
```

Erläuterungen

- Der Konstruktor von `Student` ruft den Konstruktor von `Person` auf (der wiederum den Konstruktor von `string` aufruft), bevor er das Datenelement `number` initialisiert. Der Konstruktor von `string` beschafft dynamischen Speicherplatz zur Speicherung des Namens.

- Der implizit definierte Destruktor von `Student` ruft den (ebenfalls implizit definierten) Destruktor von `Person` auf, der wiederum den Destruktor von `string` aufruft. Dieser gibt den dynamischen Speicherplatz frei, den sein Konstruktor zur Speicherung des Namens beschafft hat.

- Analoges gilt für den Konstruktor und Destruktor von `Employee`.

- Der Konstruktor von `EmployedStudent` ruft nacheinander die Konstruktoren von `Student` und `Employee` auf, die ihrerseits beide den Konstruktor von `Person` aufrufen etc. (Tatsächlich besitzt ein `EmployedStudent`-Objekt zwei unabhängige `Person`-Teilobjekte, was vermutlich nicht erwünscht ist; vgl. § 4.4.1 und § 4.4.2.)

4.2 Untertyp-Polymorphie

- Ein Objekt einer abgeleiteten Klasse (bzw. ein Zeiger oder eine Referenz auf ein solches Objekt) kann überall verwendet werden, wo ein Objekt einer (eindeutigen und zugänglichen) Basisklasse (bzw. ein Zeiger oder eine Referenz auf ein solches Objekt) erwartet wird, d. h. bei Bedarf findet eine entsprechende implizite *Aufwärtsumwandlung* (up-cast) statt.

- Umgekehrt kann ein Zeiger (bzw. eine Referenz) auf ein Objekt einer (nicht-virtuellen) Basisklasse explizit in einen Zeiger (bzw. eine Referenz) auf ein Objekt einer abgeleiteten Klasse umgewandelt werden (*Abwärtsumwandlung*, down-cast), sofern das referenzierte Objekt tatsächlich zu dieser Klasse gehört. (Andernfalls ist das Verhalten undefiniert.)

- Wenn die Basisklasse *polymorph* (im Sinne der C++-Sprachdefinition) ist, d. h. wenn sie *virtuelle Elementfunktionen* besitzt (vgl. § 4.5), kann zur Laufzeit überprüft werden, ob ein Objekt zur abgeleiteten Klasse gehört (*dynamischer Typtest*, dynamic cast, vgl. `instanceof` in Java).

Beispiel

```
// Name und ggf. Matrikelnummer von p ausgeben.
void print (const Person& p, bool stud = false) {
  cout << "Name: " << p.name << endl;
  if (stud) {
    const Student& s = static_cast<const Student&>(p);
    cout << "Matrikelnummer: " << s.number << endl;
  }
}

// Aufruf von print mit einem Person-Objekt.
Person p{"Maier"};
print(p);

// Aufruf von print mit einem Student-Objekt.
Student s{"Maier", 123};
print(s, true);

// Aufruf von print mit einem EmployedStudent-Objekt schlägt fehl,
// da Person keine eindeutige Basisklasse ist (vgl. §4.4).
EmployedStudent es{"Maier", 123, 456};
print(es, true); // Fehler!
```

4.3 Namenskonflikte

- Elemente einer abgeleiteten Klasse *verbergen* gleichnamige Elemente von Basisklassen.
- Gleichnamige Elemente aus unterschiedlichen Basisklassen führen zu *Mehrdeutigkeiten* in der abgeleiteten Klasse.
- In beiden Fällen können Elementnamen durch ihren Klassennamen *qualifiziert* werden, um den Zugriff zu ermöglichen bzw. eindeutig zu machen.

Beispiel

```
cout << "Matrikelnummer: " << es.Student::number << endl;
cout << "Personalnummer: " << es.Employee::number << endl;
```

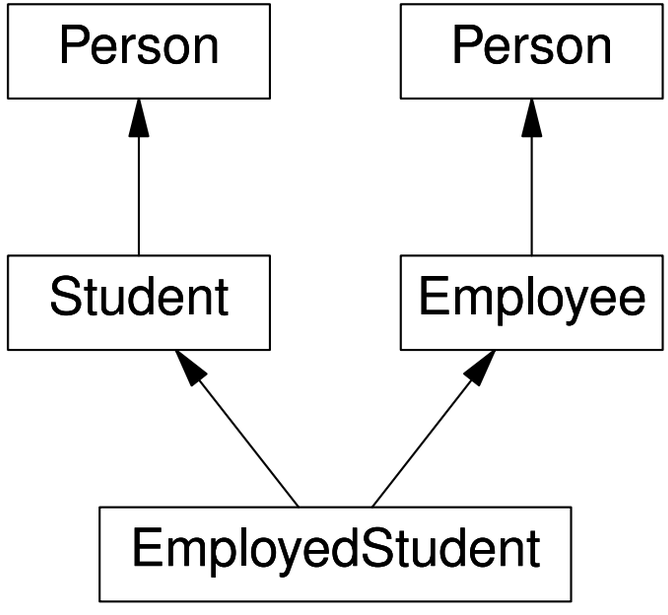
4.4 Replizierte und virtuelle Basisklassen

4.4.1 Replizierte Basisklassen

- Die *direkten* Basisklassen einer Klasse müssen paarweise verschieden sein.
- Die Menge der *indirekten* Basisklassen kann Klassen jedoch mehrfach enthalten (*replizierte Basisklassen*).
- In diesem Fall enthält ein Objekt der abgeleiteten Klasse mehrere *Teilobjekte* der entsprechenden Basisklassen, und die Elemente dieser Basisklassen sind zwangsläufig *mehrdeutig* (sofern sie nicht durch andere Elemente verborgen sind).
- Um die Elemente verwenden zu können, muss man ihren Namen mit dem Namen einer *eindeutigen* Zwischenklasse qualifizieren.

Beispiel

```
cout << es.Student::name << endl;
cout << es.Employee::name << endl;
```



V-förmige Vererbungsstruktur

4.4.2 Virtuelle Basisklassen

- Um die Replikation einer Basisklasse zu vermeiden, muss sie überall als *virtuelle Basisklasse* vereinbart werden.

- Wenn eine Klasse (direkte oder indirekte) virtuelle Basisklassen besitzt, ruft jeder Konstruktor dieser Klasse als erstes (explizit oder implizit) Konstruktoren der virtuellen Basisklassen auf, bevor Konstruktoren der (übrigen) direkten Basisklassen und Konstruktoren der eigenen Elemente aufgerufen werden.

- Die Reihenfolge der Konstruktoraufrufe für die virtuellen Basisklassen ergibt sich durch eine *Tiefensuche* im Ableitungsgraphen, wobei auf jeder Ebene die Deklarationsreihenfolge der Basisklassen berücksichtigt wird.

- Konstruktoraufrufe für die virtuellen Basisklassen werden nur von einem Konstruktor der *abgeleitetsten Klasse* (*most derived class*), d. h. vom Konstruktor eines *eigenständigen Objekts* ausgeführt.
  Entsprechende Aufrufe von Zwischenklassen-Konstruktoren werden *ignoriert*.

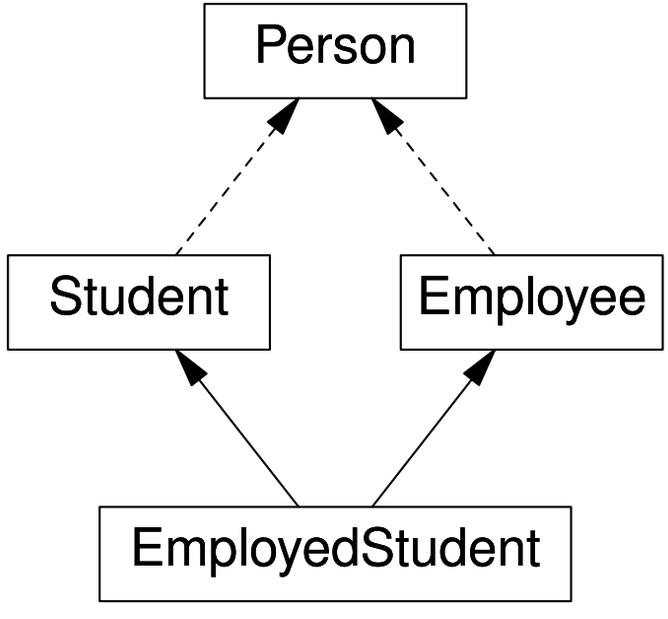
Beispiel

```
struct Person {                 // Person.
  string name;                  // Name.
  Person (string n) : name{n} {}
};

struct Student : virtual Person {       // Student.
  int number;                           // Matrikelnummer.
  Student (string n, int m) : Person{n}, number{m} {}
};
```

```
struct Employee : virtual Person {       // Arbeitnehmer.
  int number;                            // Personalnummer.
  Employee (string n, int p) : Person{n}, number{p} {}
};

struct EmployedStudent : Student, Employee { // Teilzeitstudent.
  EmployedStudent (string n, int m, int p)
                  : Person{n}, Student{n, m}, Employee{n, p} {}
}
```



Rautenförmige Vererbungsstruktur (diamond inheritance)

Erläuterungen

- Die Konstruktoren (und Destruktoren) von `Student` und `Employee` funktionieren wie zuvor.

- Der Konstruktor von `EmployedStudent` ruft als erstes den Konstruktor der (indirekten) virtuellen Basisklasse `Person` auf (der seinerseits den Konstruktor von `string` aufruft), bevor er die Konstruktoren der direkten Basisklassen `Student` und `Employee` aufruft.

- Obwohl diese Konstruktoren Aufrufe des Konstruktors von `Person` enthalten (müssen), werden diese hier ignoriert.
  (Sie werden nur ausgeführt, wenn ein *eigenständiges* `Student`- bzw. `Employee`-Objekt konstruiert wird.)

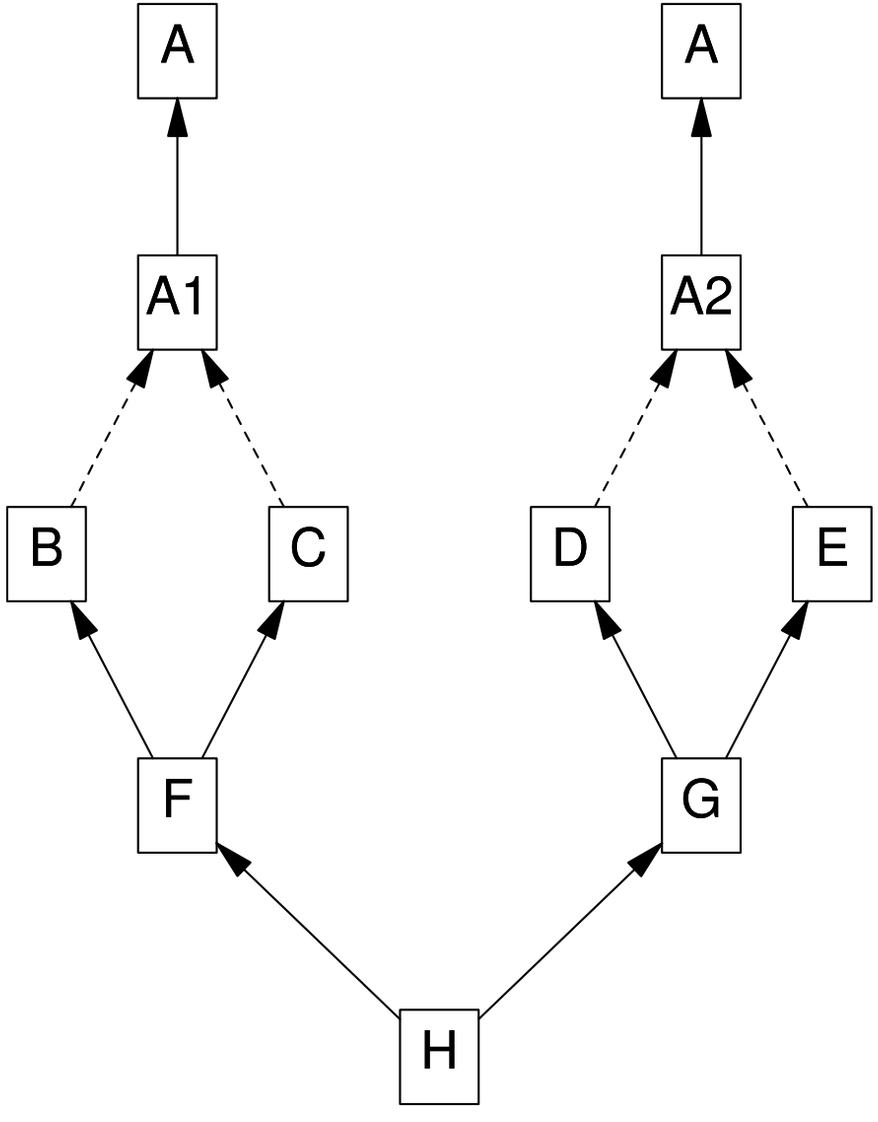
- Der (in diesem Beispiel implizite) Destruktor von `EmployedStudent` ruft (nach der Ausführung seines Rumpfs) die Destruktoren von `Employee`, `Student` und `Person` in dieser Reihenfolge auf.

4.4.3 Kombination virtueller und replizierter Basisklassen

- Wenn eine Klasse sowohl als virtuelle als auch als nicht-virtuelle Basisklasse auftritt, werden alle virtuellen Vorkommen zu einem gemeinsamen Teilobjekt zusammengefasst, während jedes nicht-virtuelle Vorkommen zu einem zusätzlichen Teilobjekt führt.

- Um mehrere Teilobjekte einer Klasse *teilweise* gemeinsam zu nutzen, muss man *von Anfang an* künstliche Zwischenklassen einführen.

Beispiel



4.4.4 Anmerkungen

- Das Vererbungskonzept von C++ zeigt, dass mehrfache Vererbung prinzipiell sprachlich beherrschbar ist.
- Replizierte Basisklassen ergeben sich technisch als Normalfall, obwohl sie in Anwendungen eher den Ausnahmefall darstellen.

- Virtuelle Basisklassen haben gewisse konzeptuelle Mängel:
  - Statische Abwärtsumwandlungen (vgl. § 4.2) von einer virtuellen Basisklasse (oder auch von einer ihrer Basisklassen) zu einer abgeleiteten Klasse sind nicht möglich.
  - Virtuelle Basisklassen einer Klasse können vor abgeleiteten Klassen nicht verborgen werden, weil jeder Konstruktor einer abgeleiteten Klasse Konstruktoren der virtuellen Basisklassen aufrufen muss.
    Dies verletzt das Modularitäts- und Geheimnisprinzip.
    (Beispielsweise müssten abgeleitete Klassen von `EmployedStudent` eigentlich nicht wissen, dass `Person` eine virtuelle Basisklasse dieser Klasse ist.)
  - Die Tatsache, dass Konstruktoraufrufe für virtuelle Basisklassen in Zwischenklassen-Konstruktoren ignoriert werden, ist verwirrend.
  - Die Entscheidung für eine virtuelle oder nicht-virtuelle Basisklasse muss an einer Stelle getroffen werden, wo sie zunächst noch keine Auswirkung hat.
    (Für `Student` und `Employee` besteht kein Unterschied, ob `Person` eine virtuelle oder nicht-virtuelle Basisklasse ist. Erst für `EmployedStudent` ist der Unterschied relevant.)
  - Replizierte virtuelle Basisklassen sind nicht direkt möglich.
    Dies verletzt wiederum das Modularitäts- und Geheimnisprinzip, weil man bei der Verwendung einer Basisklasse deren „Vorgeschichte“ kennen muss.
  - Um replizierte virtuelle Basisklassen indirekt zu erreichen, muss man – logisch wiederum an der „falschen“ Stelle – künstliche Zwischenklassen einführen.

4.5 Virtuelle Elementfunktionen

4.5.1 Konzept

- Wenn eine nichtstatische Elementfunktion `virtual` deklariert ist – und nur dann! –, kann sie in abgeleiteten Klassen *redefiniert* werden (Überschreiben).

- Die *Parametertypen* sowie eventuelle `const`- und `volatile`-*Qualifizierer* einer Redefinition müssen *exakt* mit denen der ursprünglichen Definition übereinstimmen; andernfalls wird eine *andere* gleichnamige Funktion definiert (Überladen).

- Durch das Wort `override` nach der Parameterliste kann explizit zum Ausdruck gebracht werden, dass diese Funktion eine Redefinition darstellt. Wenn es dann keine passende ursprüngliche Definition gibt, erhält man eine Fehlermeldung.

- Durch das Wort `final` nach der Parameterliste kann eine (weitere) Redefinition der Funktion in abgeleiteten Klassen verboten werden.

- Der *Resultattyp* einer Redefinition darf *kovariant* vom Typ der ursprünglichen Definition abweichen: Wenn der ursprüngliche Resultattyp ein Zeiger oder eine Referenz auf eine Klasse ist, darf der Resultattyp der Redefinition ein Zeiger bzw. eine Referenz auf eine davon abgeleitete Klasse sein.

- Eine virtuelle Elementfunktion kann mehrere geerbte virtuelle Funktionen gleichzeitig redefinieren, wenn diese alle den gleichen Namen, die gleichen Parametertypen und die gleichen Qualifizierer besitzen.
  Wenn mehrere dieser geerbten Funktionen selbst (direkte oder indirekte) Redefinitionen derselben ursprünglichen Funktion sind (was bei rautenförmiger Vererbung möglich ist), dann *müssen* sie am „Fußpunkt“ der Raute auf diese Weise gemeinsam redefiniert werden. (Diese Redefinition wird dann als „final overrider“ bezeichnet.)

- Beim Aufruf einer virtuellen Elementfunktion wird zur Laufzeit anhand des *dynamischen Typs* des Zielobjekts die passende Überschreibung der Funktion ausgewählt (*dynamisches Binden*).

- Wenn der Funktionsname beim Aufruf durch einen Klassennamen qualifiziert wird, wird der Aufruf jedoch *statisch* gebunden. Auf diese Weise kann eine Redefinition eine frühere Definition aufrufen (vgl. `super` in Java).

- Wenn eine virtuelle Elementfunktion in einer Klasse nur deklariert und dann außerhalb der Klasse definiert/implementiert wird, darf das Schlüsselwort `virtual` nur bei der Deklaration angegeben werden.

- Virtuelle Elementfunktionen und virtuelle Basisklassen sind voneinander unabhängige Konzepte.

Beispiel 1

```
// Basisklasse Person.
struct Person {
  string name; // Name.
  Person (string n) : name{n} {}

  // Virtuelle Elementfunktion.
  virtual void print () const {
    cout << "Name: " << name << endl;
  }
};
```

```
// Abgeleitete Klasse Student.
struct Student : virtual Person {
  int number; // Matrikelnummer.
  Student (string n, int m) : Person{n}, number{m} {}

  // Redefinition der virtuellen Funktion.
  virtual void print () const {
    Person::print(); // Aufruf der ursprünglichen Funktion.
    cout << "Matrikelnummer: " << number << endl;
  }
};

// Abgeleitete Klasse Employee.
struct Employee : virtual Person {
  int number; // Personalnummer.
  Employee (string n, int p) : Person{n}, number{p} {}

  // Redefinition der virtuellen Funktion.
  virtual void print () const {
    Person::print(); // Aufruf der ursprünglichen Funktion.
    cout << "Personalnummer: " << number << endl;
  }
};
```

```
// Mehrfach abgeleitete Klasse EmployedStudent.
struct EmployedStudent : Student, Employee {
  EmployedStudent (string n, int m, int p)
                  : Person{n}, Student{n, m}, Employee{n, p} {}

  // Redefinition der mehrfach geerbten virtuellen Funktion.
  virtual void print () const {
    Student::print(); // Aufruf der Funktion von Student.
    // Aufruf von Employee::print
    // würde den Namen noch einmal ausgeben.
    cout << "Personalnummer: " << Employee::number << endl;
  }
};
```

```
int main () {
  // Person.
  Person* p = new Person{...};

  // Polymorphe Verwendung von Student als Person.
  Person* s = new Student{...};

  // Polymorphe Verwendung von EmployedStudent als Student.
  Student* es = new EmployedStudent{...};

  // Aufrufe der Elementfunktion print.
  // Ausgeführte Funktion, wenn print virtual bzw. non-virtual ist:
                        // virtual                  | non-virtual
  p->print();           // Person::print            | Person::print
  s->print();           // Student::print           | Person::print
  es->print();          // EmployedStudent::print   | Student::print
  es->Student::print(); // Student::print           | Student::print
}
```

Beispiel 2

```
struct A {
  virtual void f1 ()          { cout << "A::f1" << endl; }
  virtual void f2 ()          { cout << "A::f2" << endl; }
  virtual void f3 ()          { cout << "A::f3" << endl; }
};

struct B : virtual A {
  virtual void f2 () override { cout << "B::f2" << endl; }
  virtual void f3 () override { cout << "B::f3" << endl; }
  virtual void f4 ()          { cout << "B::f4" << endl; }
};

struct C : virtual A {
  virtual void f3 () override { cout << "C::f3" << endl; }
  virtual void f4 ()          { cout << "C::f4" << endl; }
};
```

```
struct D : B, C {
  // f3 braucht in D einen "final overrider",
  // f1, f2 und f4 jedoch nicht.
  virtual void f3 () override { cout << "D::f3" << endl; }
};

int main () {
  D* d = new D;
  d->f1();       // A::f1
  d->f2();       // B::f2
  d->f3();       // D::f3
  d->f4();       // Mehrdeutig!
  d->B::f4();    // B::f4
  d->C::f4();    // C::f4
}
```

4.5.2 Polymorphe und abstrakte Klassen

- Eine Klasse, die virtuelle Elementfunktionen besitzt (oder erbt), heißt *polymorph* und kann in dynamischen Typtests (`dynamic_cast`) verwendet werden.
- Eine Klasse heißt *abstrakt*, wenn sie *rein virtuelle* Funktionen besitzt, d. h. virtuelle Elementfunktionen, die durch den Zusatz `=0` am Ende ihrer Deklaration gekennzeichnet sind.
- Von einer abstrakten Klasse können keine Objekte erzeugt werden.

Beispiel

```
// Abstrakte Basisklasse: Arithmetischer Ausdruck.
struct Expr {
  // Rein virtuelle Elementfunktion.
  virtual double eval () const = 0;
};

// Konkrete abgeleitete Klasse: Konstanter Ausdruck.
struct Const : Expr {
  const double val; // Wert des konstanten Ausdrucks.
  Const (double v) : val{v} {}

  // (Re)definition der virtuellen Funktion eval.
  virtual double eval () const {
    return val;
  }
};
```

```
// Abstrakte abgeleitete Klasse: Binärer Ausdruck.
struct Binary : Expr {
  Expr* const left;  // Linker und rechter Teilausdruck
  Expr* const right; // des binären Ausdrucks.
  Binary (Expr* l, Expr* r) : left{l}, right{r} {}

  // Die virtuelle Funktion eval bleibt rein virtuell.
};

// Konkrete abgeleitete Klasse: Addition.
struct Add : Binary {
  Add (Expr* l, Expr* r) : Binary{l, r} {}

  // (Re)definition der virtuellen Funktion eval.
  virtual double eval () const {
    return left->eval() + right->eval();
  }
};

// Analog für Subtraktion, Multiplikation und Division.
......
```

```
int main () {
  Expr* x = new Add{new Const{1}, new Const{2}};

  // Aufruf der virtuellen Elementfunktion.
  cout << x->eval() << endl;

  // Dynamische Typtests.
  if (dynamic_cast<Const*>(x)) { // Nein.
    cout << "constant expression" << endl;
  }
  if (dynamic_cast<Binary*>(x)) { // Ja.
    cout << "binary expression" << endl;
  }
  if (Add* a = dynamic_cast<Add*>(x)) { // Ja.
    cout << "left: " << a->left->eval() << endl;
    cout << "right: " << a->right->eval() << endl;
  }
}
```

4.5.3 Dynamische Typtests und -umwandlungen

- Für einen Zeiger `x` auf ein Objekt eines polymorphen Typs `X` und einen Zeigertyp `T` mit Zieltyp `Y` ist `dynamic_cast<T>(x)` *erfolgreich*, wenn eine der folgenden Bedingungen erfüllt ist:
  - Das Objekt `*x` ist ein öffentliches Teilobjekt eines eindeutigen Objekts des Typs `Y` (Abwärtsumwandlung, down-cast).
  - Das Objekt `*x` ist ein öffentliches Teilobjekt seines Gesamtobjekts (most derived object), und dieses besitzt ein eindeutiges öffentliches Teilobjekt des Typs `Y` (Querumwandlung, cross-cast).
- Im Erfolgsfall liefert `dynamic_cast<T>(x)` einen Zeiger auf das genannte Objekt des Typs `Y`, andernfalls einen Nullzeiger. Wenn `x` ein Nullzeiger ist, ist das Resultat ebenfalls ein Nullzeiger.
- Wenn `Y` gleich `void` ist, ist `dynamic_cast<T>(x)` immer erfolgreich und liefert einen Zeiger auf das Gesamtobjekt, zu dem das Objekt `*x` gehört.
- Wenn der Typ `X` `const`- und/oder `volatile`-qualifiziert ist, muss der Typ `Y` ebenso qualifiziert sein, aber nicht umgekehrt.

- Wenn `Y` eine Basisklasse von `X` ist, ist `dynamic_cast<T>(x)` eigentlich ein `static_cast`, der genau dann erfolgreich ist, wenn `Y` eindeutig und zugänglich ist.
  Da dies bereits zur Übersetzungszeit überprüft wird, erhält man ggf. eine entsprechende Fehlermeldung.
  In diesem Fall muss der Typ `X` nicht polymorph sein.

- Wenn `Y` gleich `X` ist, ist `dynamic_cast<T>(x)` immer erfolgreich und liefert einfach `x`.
  Auch in diesem Fall muss der Typ `X` nicht polymorph sein.

- Für einen L-Wert `x` des Typs `X` und einen L-Wert-Referenztyp `T` mit Zieltyp `Y` bzw. für einen beliebigen Wert `x` des Typs `X` und einen R-Wert-Referenztyp `T` mit Zieltyp `Y` ist `dynamic_cast<T>(x)` jeweils analog wie für Zeigertypen definiert.
  Im Erfolgsfall erhält man eine L- bzw. R-Wert-Referenz auf das genannte Objekt des Typs `Y`, andernfalls wird eine Ausnahme des Typs `bad_cast` geworfen.

- Die Begriffe „öffentlich“ und „zugänglich“ werden in § 4.7 definiert.

Beispiel

```
struct A {
  int a;
  virtual ~A () {} // Virtueller Destruktor, damit A polymorph ist.
};
struct B {
  int b;
};
struct C {
  int c;
  virtual ~C () {} // Virtueller Destruktor, damit C polymorph ist.
};

struct D : A, B, virtual C {};

D* d = new D;
A* a = d;                          // Implizite Aufwärtsumwandlung.

d = dynamic_cast<D*>(a);           // Erfolgreiche Abwärtsumwandlung.
B* b = dynamic_cast<B*>(a);        // Erfolgreiche Querumwandlung.
C* c = dynamic_cast<C*>(a);        // Erfolgreiche Querumwandlung.
d = dynamic_cast<D*>(c);           // Erfolgreiche Abwärtsumwandlung.
```

Fortsetzung des Beispiels

```
struct D1 : D {};
struct D2 : D {};
struct DD : D1, D2 {};

DD* dd = new DD;
a = (D2*)dd;          // Explizite und implizite Aufwärtsumwandlung.

d = dynamic_cast<D*>(a);  // Erfolgreiche Abwärtsumwandlung.
b = dynamic_cast<B*>(a);  // Nicht erfolgreiche Querumwandlung.
c = dynamic_cast<C*>(a);  // Erfolgreiche Querumwandlung.
d = dynamic_cast<D*>(c);  // Nicht erfolgreiche Abwärtsumwandlung.
```

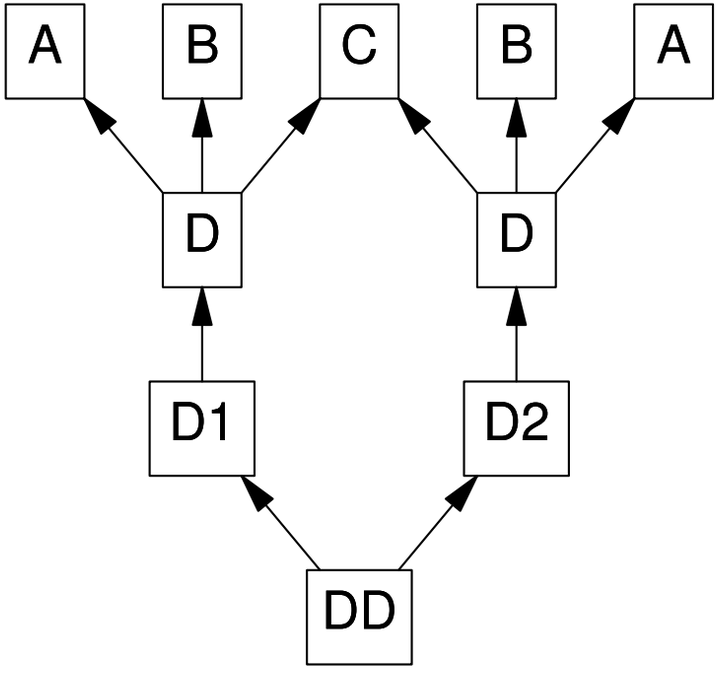
Erläuterungen

- Die Querumwandlung von `a` nach `B*` ist jetzt nicht erfolgreich, weil das Gesamtobjekt `*dd`, zu dem `*a` gehört, zwei `B`-Teilobjekte enthält und `*a` kein Teilobjekt von einem von ihnen ist.

- Die Querumwandlung von `a` nach `C*` ist jedoch erfolgreich, weil das Gesamtobjekt `*dd` (aufgrund der virtuellen Vererbung) nur ein `C`-Teilobjekt enthält.

- Auch die Abwärtsumwandlung von `a` nach `D*` ist erfolgreich, obwohl das Gesamtobjekt `*dd` zwei `D`-Teilobjekte enthält, weil `*a` ein Teilobjekt von genau einem von ihnen ist.

- Die Abwärtsumwandlung von `c` nach `D*` ist jedoch nicht erfolgreich, weil `*c` ein Teilobjekt von zwei `D`-Teilobjekten ist.



4.5.4 Virtuelle Destruktoren

- Wenn mittels `delete` ein Objekt gelöscht wird, dessen dynamischer Typ von seinem statischen Typ abweicht, muss der statische Typ einen *virtuellen Destruktor* besitzen, damit durch dynamisches Binden der Destruktor des dynamischen Typs ausgeführt werden kann.
  (Beim Löschen dynamischer Reihen müssen statischer und dynamischer Typ jedoch immer übereinstimmen.)

- In den Beispielen in § 4.5.3 könnten die dynamischen Objekte `*d` und `*dd` daher auch mittels `delete a` oder `delete c` gelöscht werden (weil die Klassen `A` und `C` jeweils einen virtuellen Destruktor besitzen), aber nicht mittels `delete b` (weil `B` keinen virtuellen Destruktor besitzt).

- Wenn eine Klasse polymorph sein soll, obwohl sie keine virtuellen Elementfunktionen besitzt, kann ihr Destruktor „pro forma“ virtuell definiert werden (vgl. die Beispiele in § 4.5.3).