Vorlesung

Objektorientiertes Programmieren in C++

Teil 10 - WS 2025/26

Detlef Wilkening
www.wilkening-online.de

17 Vererbung & Polymorphie

17.1 Intermezzo

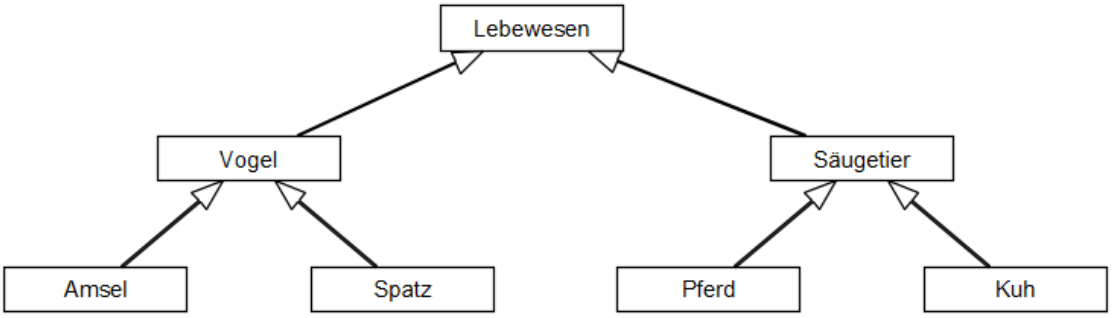
Wenn wir mehr Zeit für die Vorlesung hätten, dann kämen jetzt erstmal einige andere Themen an die Reihe. Aber wir nähern uns dem Ende der Vorlesung, darum müssen wir fokussieren. Und C++ hat zwei große zentrale Themen. Da ist zum einen die generische Programmierung mit Templates, die in der Vorlesung leider viel zu kurz gekommen sind. Und zum anderen die objekt-orientierte Programmierung mit Klassen, Vererbung und Polymorphie – und dieses Thema muß in der Vorlesung rein. Darum sind wir jetzt in diesem Kapitel.

Leider ist es so, dass wir eigentlich die Kapitel dazwischen benötigen würden. Dann bei „dynamic_cast“ (siehe Kap. 18.8) können Exceptions fliegen, also bräuchten wir sie eigentlich. Vererbung und Polymorphie geht in C++ fast immer mit dynamischer Speicherverwaltung einher, und die basiert in modernem C++ auf Smart-Pointern, und die basieren auf den normalen C-Zeigern. Also bräuchten wir das alles. Und dafür wären auch Exceptions wieder notwendig.

Aber all diese Themen passen zeitlich nicht mehr. Darum machen wir den großen Sprung. Wenn Sie in den Beispielen Zeiger, dynamische Speicherverwaltung oder Exceptions sehen, dann ignorieren Sie sie im Detail. Die wichtigen Grundlagen von Vererbung und Polymorphie sind auch ohne dieses Wissen verständlich, und die meisten Beispiele kommen ohne diese Dinge aus. Und außerdem haben ja die meisten von Ihnen im letzten Semester C gehört, und da gehören Zeiger zum Inhalt. Das heißt, Sie kennen zumindest das prinzipielle Konzept mehr als gut genug.

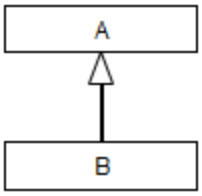
17.2 Vererbung

Vererbung (genau genommen „öffentliche Vererbung“) ist die Modellierung einer "*ist-ein*" Beziehung. Eine *„ist-ein“* Beziehung meint, dass jedes Objekt der abgeleiteten Klasse ein Objekt der Basis-Klasse ersetzen kann, ohne dass es sematische Probleme gibt.



Eine „ist-ein“ Beziehung in der normalen Welt

Immer dann, wenn die zu modellierende Domaine „ist-ein“ Beziehungen enthält, bietet sich damit „öffentliche Vererbung“ als eine sinnvolle Modellierungs-Strategie an. Bevor wir hier aber tiefer einsteigen, lassen sie uns das Konzept der Vererbung noch mal detaillierter betrachten, und einige Begriffe definieren.



Eine abstrakte ganz allgemeine „ist-ein“ Beziehung

Hierbei ist:

- A Basis-Klasse (von B und C)
- B ist abgeleitet von A => B ist ein A => alles was für A gilt, gilt auch für B
- C ist abgeleitet von A => C ist ein A => alles was für A gilt, gilt auch für C

In Richtung der abgeleiteten Klassen findet eine Spezialisierung statt:

- B ist eine Spezialisierung von A
- Ein Pferd ist eine Spezialisierung eines Säugetiers
- Alles, was für Säugetiere gilt (z. B. Alter, Gewicht, ...), gilt auch für Pferde. All diese Attribute und Funktionen erbt Pferd von Säugetier.

In Richtung der Basis-Klassen findet eine Generalisierung oder Verallgemeinerung statt – Basis-Klassen fassen gemeinsame Dinge der abgeleiteten Klassen zusammen:

- A enthält alle Gemeinsamkeiten von B und C.
- Vogel enthält alles Vogel-typische, unabhängig, ob es sich um eine Amsel oder eine Möve handelt.

Hinweise

- Die abgeleitete Klassen (z. B. B und C) sind unabhängig voneinander.
- Von einer Klasse können beliebig viele andere Klassen abgeleitet werden.
- Eine Klasse kennt die von ihr abgeleiteten Klassen nicht.
- Umgekehrt kennt die abgeleitete Klasse natürlich ihre Basis-Klasse.
- Ein Basis-Klasse wird oft auch Super-Klasse genannt.
- Eine abgeleitete Klasse wird oft auch Sub- oder Unter-Klasse genannt.

Hinweis – alle Vererbungen in diesem Kapitel sind Einfach-Vererbungen („single inheritance"), d. h. eine Klasse hat **genau eine** Basis-Klasse. C++ unterstützt auch Mehrfach-Vererbung („multiple inheritance"), die aus Zeitmangel leider nicht behandelt wird.

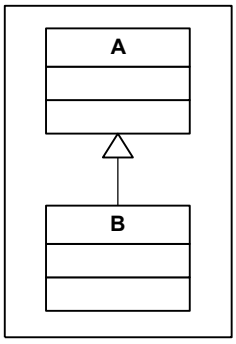
17.2.1 Implementation

Wie wird Vererbung in C++ implementiert?

Syntax

class Klassen-Name **: [Vererbungs-Spezifizierer]** Basis-Klasse { ... };

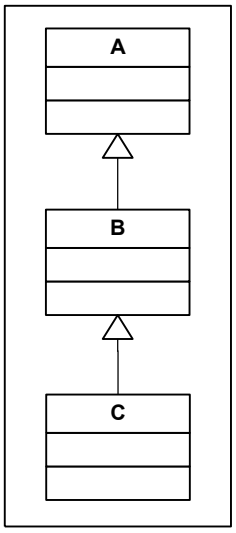
```
class A
{
public:
   int ai;
   void af();
};

class B : public A         // Definition Klasse B oeffentlich abgeleitet von A
{
public:
   int bi;
   void bf();
};

int main()
{
   A a;
   B b;

   a.ai=7;        // okay
   a.af();        // okay
   a.bi=8;        // Compiler-Fehler - A hat keine Varibale bi
   a.bf();        // Compiler-Fehler - A hat keine Funktion bf()

   b.ai=9;        // okay - B hat Variable ai von A geerbt
   b.af();        // okay - B hat Funktion af() von A geerbt
   b.bi=10;       // okay
   b.bf();        // okay
}
```



Klassen-Hierarchie des Beispiels

Hinweis – die abgeleitete Klasse erbt die Funktionalität der Basis-Klasse, was man daran sieht, dass das Objekt der abgeleiteten Klasse „B“ die Funktionen und Attribute von „A“ besitzt, ohne dass sie explizit programmiert werden mussten.

Die Vererbungshierarchie lässt sich beliebig fortsetzen, z.B. in dem man zusätzlich eine Klasse „C“ von „B“ ableitet:



Klassen-Hierarchie des erweiterten Beispiels

```
// Klasse A und B wie eben

class C : public B
{
public:
   int ci;
   void cf();
};

int main()
{
   C c;
   c.ai=11;      // okay - C hat Variable ai von B (wiederum von A) geerbt
   c.af();       // okay - C hat Funktion af() von B (wiederum von A) geerbt
   c.bi=12;      // okay - C hat Variable bi von B geerbt
   c.bf();       // okay - C hat Funktion bf() von B geerbt
   c.ci=13;      // okay
   c.cf();       // okay
}
```

Welche Elemente enthalten die einzelnen Klassen nun?

Klasse	Attribut	Element-Funktionen
A	ai	af()
B	ai bi	af() bf()
C	ai	af()

bi	bf()
ci	cf()

Denn
- B ist ein A
- C ist ein B
- C ist ein A (Vererbung ist transitiv)

17.2.2 Konstruktoren

Konstruktoren müssen meist neu definiert werden, da sie Default mäßig nicht vererbt werden. Dies ist vernünftig, da ein ererbter Konstruktor nicht wissen kann, wie er die neuen nicht-ererbten Attribute initialisieren soll.

Wird ein Objekt einer abgeleiteten Klasse erzeugt und ist für die Basis-Klasse kein spezieller Konstruktor angegeben, so wird automatisch der Standard-Konstruktor der Basis-Klasse für den Basis-Klassenanteil des Objekts genommen.

```
class A
{
public:
   A() { cout << "Konstruktor A\n"; }
};

class B : public A                      // B abgeleitetet von A
{
public:
   B() { cout << "Konstruktor B\n"; }
};

int main()
{
   B b;
}
```

Ausgabe
```
Konstruktor A
Konstruktor B
```

Hat die Basis-Klasse keinen Standard-Konstruktor, so muss der gewünschte Konstruktor in der Initialisierungsliste der abgeleiteten Klasse angegeben werden.

```
class A
{
public:
   A(int);
};

A::A(int i)
{
}

class B : public A              // B abgeleitetet von A
{
public:
   B();
   B(int);
};

B::B()                          // Compiler-Fehler, kein Standard-Konstruktor
{
}
```

```
B::B(int i)                          // okay, explizite Angabe des Konstruktors
  : A(i)
{
}
```

Beim Konstruieren eines Objekts wird **vor** den Konstruktoren der Attribute der Konstruktor der Basis-Klasse aufgerufen. Die Basis-Klasse verhält sich bzgl. der Erstellung eines Objekts quasi wie ein Attribut, das als erstes in der Klassen-Definition steht.

```
class attribut
{
public:
   attribut(int i) { cout << "attribut(" << i << ")\n"; }
};

class base
{
public:
   base() : attribut_(1) { cout << "base()\n"; }

private:
   attribut attribut_;
};

class derived : public base
{
public:
   derived() : attribut_(2), base() { cout << "derived()\n"; }

private:
   attribut attribut_;
};

int main()
{
   derived d;
}
```

```
Ausgabe
attribut(1)
base()
attribut(2)
derived()
```

Hinweis – obwohl in der Initialiisierung von „B()“ das Attribut „attribut_“ vor der Basis-Klasse „A“ angegeben ist, wird die Basis-Klasse vor dem Attribut der abgeleiteten Klasse konstruiert. Die Reihenfolge der Konstruktion ist durch die Vererbungsbeziehung und die Klassen-Definition festgelegt.

Was passiert genau?

1. Speicherplatz reservieren
2. Aufruf des Konstruktors der Basis-Klasse:
   a) Aufruf der Konstruktoren der Attribute der Basis-Klasse (Reihenfolge Definition)
   b) Ausführen des Konstruktor-Rumpfs der Basis-Klasse
3. Aufruf der Konstruktoren der Attribute der abgeleiteten Klasse (Reihenfolge Definition)
4. Ausführen des Konstruktor-Rumpfs der abgeleiteten Klasse

Durch diese Reihenfolge ist gewährleistet, dass jedes Objekt immer einen stabilen Zustand der ihm zugrunde liegenden Teil-Objekte sieht.

17.2.3 Destruktoren

Auch Destruktoren werden nicht vererbt.

Die Destruktoren werden umgekehrt abgearbeitet, d. h. von der abgeleiteten Klasse bis hin zur Basis-Klasse. Es werden automatisch **alle** Destruktoren des Vererbungszweiges durchlaufen.

```
class attribut
{
public:
   attribut(int i) : n_(i) { cout << "attribut(" << n_ << ")\n"; }
   ~attribut() { cout << "~attribut(" << n_ << ")\n"; }
private:
   int n_;
};

class base
{
public:
   base() : att_(1) { cout << "base()\n"; };
   ~base() { cout << "~base()\n"; };

private:
   attribut att_;
};

class derived : public base
{
public:
   derived() : att_(2) { cout << "derived()\n"; };
   ~derived() { cout << "~derived()\n"; };

private:
   attribut att_;
};

int main()
{
   derived d;
}
```

Ausgabe

```
attribut(1)
base()
attribut(2)
derived()
~derived()
~attribut(2)
~base()
~attribut(1)
```

Durch diese Reihenfolge ist gewährleistet, dass auch bei der Zerstörung einer Objekts jedes Teil-Objekt immer einen stabilen Zustand der ihm zugrunde liegenden Objekte sieht.

17.2.4 Qualifizierter Name

Manchmal ist es nötig, Symbole einer Klasse anzusprechen, die eigentlich nicht sichtbar sind, da sie überschrieben oder verdeckt sind. In diesem Fall muss das Symbol über den Namen der Klasse, den Scope-Resolution Operator und den eigentlichen Namen referenziert

werden.

```
class A
{
public:
   void f();
};

class B : public A
{
public:
   void g();
};

void B::g()
{
   A::f();       // expliziter Aufruf der Element-Funktion f der Klasse A
}

int main()
{
   B b;
   b.g();
   b.A::f();     // expliziter Aufruf der Element-Funktion f der Klasse A
}
```

Hinweis – da wir Überschreiben und Verdeckung noch nicht kennen, ist das Beispiel etwas hergeholt, aber mit dem nächsten Kapitel wird sich das ändern.

17.2.5 Überschreiben von Funktionen I

Eine abgeleitete Klasse erbt von der Basis-Klasse u.a. ihr Verhalten - die Funktionen. Das ererbte Verhalten muss für die abgeleitete Klasse aber nicht korrekt sein. In manchen Fällen ist es komplett falsch, in anderen stimmt das Prinzip, aber im Detail gibt es Abweichungen. In solchen Fällen kann die ererbte Funktion von der abgeleiteten Klasse überschrieben werden.

Achtung – das *„Überschreiben“*, wie es in diesem Kapitel vorgestellt wird, ist noch nicht das Richtige Überschreiben, und arbeitet in manchen Situationen fehlerhaft. Erst mit virtuellen Funktionen wird Überschreiben vollständig – das folgt gleich.

Falls die Implementierung einer Basis-Klassen Element-Funktion nicht passend ist, kann sie in einer abgeleiteten Klasse neu implementiert, d. h. komplett überschrieben werden.

```
class A
{
public:
   void f() { cout << "A::f()\n"; }
};

class B : public A
{
public:
   void f() { cout << "B::f()\n"; }

   void g()
   {
      f();              // ruft B::f() auf
      A::f();           // ruft A::f() auf
   }
};

int main()
```

```
{
   A a;
   B b;
   a.f();               // ruft A::f() auf
   b.f();               // ruft B::f() auf
   b.A::f();            // ruft A::f() auf
   b.g();
}
```

Ausgabe
```
A::f()
B::f()
A::f()
B::f()
A::f()
```

Auf die Art und Weise kann eine nicht passende Implementierung einer Basis-Klasse in einer abgeleiteten Klasse neu implementiert, d. h. überschrieben werden.

Ein Beispiel wäre eine Funktion „get_salary“ in einer Klasse „employee“ und in der abgeleiteten Klasse „sales_manager“. Ein Vertriebsleiter ist sicherlich ein Angestellter, d.h. für ihn gelten die Funktionen get_name(), get_personnel_no(), usw – von daher sieht Vererbung nach der korrekten Modellierung aus. Aber während Angestellte meist ein Fest-Gehalt beziehen, wird bei einem Vertriebsleiter oft eine Umsatz-Beteiligung eingerechnet. Von daher ist die „get_salary“ Implementierung der Basis-Klasse sicher nicht richtig.

```
class employee
{
public:
   const std::string& get_name() const;
   int get_personnel_no() const;
   money get_salary() const;
   ...
};

class sales_manager : public employee
{
public:
   money get_salary() const;
   ...
};
```

Hinweis – oft ist es so, daß die Basis-Klassen Implementierungen gar nicht so schlecht sind, aber eben nicht 100% passen. Vielleicht bekommt der Vertriebsleiter zusätzlich zu einem Festgehalt einen variablen Anteil hinzu – in diesem Fall wäre die Basis-Klassen Implementierung mit dem Festgehalt ja nicht falsch, sondern eben nur ein Teil der korrekten Implementierung. Darum ist es oft sinnvoll in einer Neu-Implementierung auf die Basis-Klassen Implementierung zurückzugreifen.

Dafür wird dann immer die vollständige Referenzierung (oder Qualifizierung) mit dem Basis-Klassen Namen benötigt – ansonsten würde eine Endlos-Rekursion entstehen, denn dann würde die Funktion sich ja immer selbst aufrufen.

```
void derived::fct()
{
   ...
   base::fct();            // expliziter Aufruf der Original-Element-Funktion
   ...
}
```

17.2.6 Zugriffsbereich protected

Zusätzlich zu den Zugriffsbereichen **public** und **private** gibt es Außerdem noch **protected**. Der Zugriffsbereich **protected** liegt in seiner Wirkung zwischen **public** und **private**.

Im Gegensatz zu **private** kann auf Elemente im Zugriffsbereich **protected** auch noch von abgeleiteten Klassen zugegriffen werden.

```
class A
{
public:
   void fpublic();

protected:
   void fprotected();

private:
   void fprivate();
};

class B : public A
{
public:
   void f();
};

void B::f()
{
   fpublic();          // okay - Aufruf von A::fpublic()
   fprotected();       // okay - Aufruf von A::fprotected()
   fprivate();         // Compiler-Fehler - A::fprivate() ist nicht erreichbar
}

int main()
{
   A a;
   a.fpublic();        // okay - Aufruf von A::fpublic()
   a.fprotected();     // Compiler-Fehler - A::fprotected() ist nicht erreichbar
   a.fprivate();       // Compiler-Fehler - A::fprivate() ist nicht erreichbar
}
```

Achtung – beachten Sie bitte, dass bzgl. aller abgeleiteten Klassen der **protected**-Bereich mit zur öffentlichen Schnittstelle, d. h. zum Interface gehört und entsprechend designt werden sollte. Legen Sie d.h. auch in den **protected**-Bereich keine Datenelemente.

17.2.7 Vererbungs-Spezifikationen

In C++ gibt es drei Vererbungs-Spezifikationen:

public
protected
private default bei class

Normalerweise wird nur die **public**-Vererbung benutzt, die semantisch einer *"ist-ein"* Beziehung entspricht. Dem gegenüber modellieren **protected**- und **private**-Vererbungen eine ***„ist-implementiert-mit"*** bzw. eine ***„hat-ein"*** Beziehung, die spezielle Möglichkeiten bietet. Außerdem wird mit der Vererbungs-Spezifikationen wird bestimmt, wie die

Zugriffsbereiche der Basis-Klasse den Zugriffs-Bereichen der abgeleiteten Klasse zugeordnet werden.

Hinweis – in der Praxis sind 99,9% aller Fälle öffentliche Vererbung – fast alle anderen Sprachen kennen auch nur diese Art der Vererbung.

Achtung – ein gern gemachter Fehler in C++ ist das Vergessen des Vererbungs-Spezifizierers „public“, wodurch eine private Vererbung mit anderer Semantik und anderem Verhalten entsteht.

17.3 Konsequenzen aus der „ist-ein“ Beziehung

Ganz im Sinne der *„ist-ein“* Semantik können in C++ Objekte einer öffentlich-abgeleiteten Klasse auch immer für Objekte der Basis-Klasse stehen. Dies hat mehrere Konsequenzen.

17.3.1 Konsequenz 1

Wird in einem Ausdruck ein Objekt einer Basis-Klasse erwartet, so kann auch immer ein abgeleitetes Objekt als Argument benutzt werden. Dies betrifft z.B. Funktions-Aufrufe oder Zuweisungen, gilt aber für alle Arten von Ausdrücken.

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

void f(A)
{
}

int main()
{
   B b;
   f(b);          // okay - B Objekt wird als A benutzt, da B ein A ist

   A a;
   a = b;          // okay - B Objekt wird als A benutzt, da B ein A ist
}
```

In diesem Beispiel wird „b“ in beiden Ausdrücken automatisch in ein A-Objekt gewandelt - hierbei geht jede Information über den eigentlichen Typ verloren, d.h. in der Funktion „f“ ist der Parameter wirklich ein A-Objekt, und auch das „a“ ist nur ein „A“ und mehr nicht.

17.3.2 Konsequenz 2

Da ein Objekt einer abgeleiteten Klasse immer für ein Objekt einer Basis-Klasse stehen kann, muss dies auch für jegliche Art von Referenzen auf Basis-Klassen-Objekte stimmen.

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

int main()
{
   B b;
   A* p = &b;       // okay - denn ein B "ist ein" A
   A& r = b;        // okay - denn ein B "ist ein" A
```

```
}
```

Auch dies ist ganz im Sinne der „ist-ein“ Semantik. Es ist ja auch korrekt, wenn sie z.B. auf ein Auto zeigen und sagen: „Dies ist eine Maschine“, obwohl sie dabei eine sehr allgemeine Abstraktion benutzen, aber ein Auto ist eben eine Maschine.

Hierbei verliert das Objekt auch nicht seine Identität, das es nicht kopiert, zugewiesen oder sonstwie verändert wird. Es bleibt im Speicher ganz normal bestehen, und wird nur von außen über verschiedene Typen referenziert.

17.3.3 Statischer und dynamischer Typ

Man unterscheidet in C++ den sogenannten statischen und den dynamischen Typ.

- Der statische Typ ist der Typ, den der Compiler sieht, da er ohne wenn und aber zur Compilezeit feststeht und eindeutig bekannt ist. Im Beispiel sind dies die Typ-Varianten von “A“ der Variablen „p“ und „r“.
- Dem gegenüber ist der dynamische Typ der echte Typ des Objekts, auf das verwiesen wird. Dieser ist zur Compilezeit nicht zwingend bekannt, und muss nicht dem statischen Typ entsprechen. Im Beispiel ist der dynamische Typ des referenzierten Objekts „B“, obwohl der statische Typ „A“ ist.

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

int main()
{
   B b;            // statischer Typ ist "B", dynamischer Typ auch
   A* p = &b;      // statischer Typ ist "A*", der dynamische Typ kann mehr sein...
   A& r = b;       // statischer Typ ist "A&", der dynamische Typ kann mehr sein...

   int n;          // statischer Typ ist "int", dynamischer Typ auch
   int& ri = n;    // statischer Typ ist "int&", dynamischer Typ auch
}
```

Der dynamische Typ kann sich nur dann vom statischen Typ unterscheiden, wenn:

- die Variable ein Zeiger oder eine Referenz ist, und
- der statische Typ eine Klasse ist, von der es Ableitung geben kann.

17.3.4 Diskussion

Falls Sie das Ganze etwas verwundert, machen Sie sich mal von der ganzen Computerei frei, und betrachten das Ganze mit einem *normalen* Beispiel: Wenn Sie z.B. auf einen Stuhl zeigen und sagen „das ist ein Stuhl“, dann wird Ihnen wohl niemand widersprechen. Aber auch die Aussage „das ist ein Möbelstück“ wäre ohne Frage richtig.

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

int main()
{
   B b;
   A* p = &b;      // "p" zeigt auf ein "A", und vielleicht auch auf mehr...
```

```
    A& r = b;       // "r" referenziert ein "A", und vielleicht auch mehr...
}
```

Und genau das gleiche passiert hier: Die Referenz-Variable sagt mit ihrem statischen Typ „A“ das sie ein Möbelstück referenziert (*darauf zeigt*), obwohl sie doch in Wirklichkeit einen Stuhl (ein Objekt vom Typ „B“) referenziert (*darauf zeigt*). Aber daran ist nichts Falsches und unwahres - sie sagt nur nicht alles. Aber in vielen Kontexten reicht das. Wir sagen zu unserem Besuch auch „Nimm dir einen Stuhl“, und lassen offen ob er sich in einen Sessel, die gute Coach oder den normalen Holzstuhl setzen soll. Warum auch? Im Prinzip würde es sogar reichen zu sagen „Nimm doch bitte Platz“.

17.3.5 Etwas komplexerer Fall

Manch einer bringt den Einwand, dass die Unterscheidung in statische und dynamische Typen doch sinnlos ist – jeder sieht doch bei dem obigen Beispiel, dass „p“ und „r“ auf das B Objekt „b“ verweisen. Bei dem obigen Beispiel ist dies richtig – es ist halt als einführendes Beispiel sehr einfach. Schon bei einem nur etwas komplexeren Fall läßt sich der dynamische Typ prinzipiell erst zur Laufzeit bestimmen – und hat damit seine Berechtigung.

```
class A { };
class B : public A { };
class C : public A { };

void fct(A& r)          // Ginge auch mit Zeigern - genau der gleiche Effekt
{
   ...                  // Welchen Objekt-Typ referenziert "r" hier? Ein "A", "B" oder "C"?
}

int main()
{
   A a;
   B b;
   C c;
   fct(a);
   fct(b);
   fct(c);
}
```

Beim ersten Durchlauf von „fct“ referenziert „r“ ein A-Objekt, beim Zweiten ein „B“ Objekt und beim Dritten ein „C“ Objekt. Der dynamische Typ von „r“ ändert sich hier zur Laufzeit und seine Bestimmung macht daher auch immer erst während des konkreten Aufrufs Sinn. Er läßt sich weder von uns noch vom Compier vorher bestimmen.

In diesem Beispiel können wir vorher noch sagen, bei welchem Durchlauf welcher dynamische Typ vorkommt. Wenn wir die Funktions-Aufrufe aber von einem Zufallszahlen-Generator oder Benutzer-Eingaben abhängig machen – dann ist selbst das vorher nicht mehr möglich.

In realen Programmen können wir (und der Compiler) nur den statischen Typ exakt bestimmen – über den dynamischen Typ kann man erst zur Laufzeit in der konkreten Situation reden. Und genau hier kommt jetzt die Polymorphie ins Spiel…

17.4 Polymorphie

17.4.1 Bisheriges Verhalten

Bisher werden Element-Funktions-Aufrufe des Compilers über den statischen Typ einer Variablen aufgelöst.

```
class A
{
public:
   void f() const { cout << "A::f\n"; }
};

class B : public A
{
public:
   void f() const { cout << "B::f\n"; }
};

class C : public B
{
public:
   void f() const { cout << "C::f\n"; }
};

int main()
{
   C c;
   A& ra = c;
   B& rb = c;
   C& rc = c;
   ra.f();       // => A::f
   rb.f();       // => B::f
   rc.f();       // => C::f
}
```

```
Ausgabe
A::f
B::f
C::f
```

Dieses Verhalten kommt daher, dass der Compiler die Funktions-Aufrufe zum Compile-Zeitpunkt fest *verdrahtet* (*statische Bindung*, *frühe Bindung, static binding*). Hierbei wird nicht der echte dynamische Objekt-Typ benutzt, da der Compiler diesen nicht wissen kann. So bindet der Compiler den Funktions-Aufruf fest über den statischen Typ, über den den der Funktions-Aufruf vorgenommen wird. Wir haben das schon detailliert in den Kapiteln über die Arbeitsweise von Compilern und Linkern kennen gelernt. Diese direkte Verdrahtung erklärt ja auch die Performance von Funktions-Aufrufen in C++.

Anders ist dies bei virtuellen Funktionen.

17.4.2 Virtuelle Funktionen, bzw. Überschreiben von Funktionen II

Bei virtuellen Funktionen wird der Funktions-Aufruf **erst zur Laufzeit** festgelegt (*dynamische Bindung, späte Bindung, late binding*). Hierbei wird zur Laufzeit quasi der echte dynamische Typ des Objekts bestimmt und die Funktion dieses Typs aufgerufen.

Um eine Element-Funktion zu einer virtuellen Funktion zu machen, muss das Schlüsselwort „virtual“ vor die Element-Funktions-Deklaration geschrieben werden. Bei der Definition einer virtuellen Funktion darf das Schlüsselwort **virtual** nicht auftauchen.

Syntax (Element-Funktions-Deklaration)

virtual rueckgabetyp funktionsname (parameterliste);
virtual rueckgabetyp funktionsname (parameterliste) const;

Bei den überschreibenden Funktionen kann das Schlüsselwort “virtual” weggelassen werden, und wird es auch fast immer. Stattdessen wird das optionale Schlüsselwort “override” hinter die Funktion eingefügt. Der Compiler überprüft dann, ob diese Funktion wirklich eine Funktion der Basisklasse überschreibt.

```
class A
{
public:
   virtual void f() const { cout << "A\n"; }
};

class B : public A
{
public:
   void f() const override { cout << "B\n"; }
};

class C : public A
{
public:
   virtual void f() const override { cout << "C\n"; }
};

class D : public C
{
public:
   virtual void f() const override;
};

void D::f() const
{
   cout << "D\n";
}

class E : public A { };

class F : public E
{
public:
   virtual void f() const override { cout << "F\n"; }
};

int main()
{
   A a;
   B b;
   C c;
   D d;
   E e;
   F f;
   A* p;

   p=&a;
   p->f();       // p zeigt auf ein A-Objekt  => A::f() -> Ausgabe A

   p=&b;
   p->f();       // p zeigt auf ein B-Objekt  => B::f() -> Ausgabe B
```

```
   p=&c;
   p->f();       // p zeigt auf ein C-Objekt  => C::f() -> Ausgabe C

   p=&d;
   p->f();       // p zeigt auf ein D-Objekt  => D::f() -> Ausgabe D

   p=&e;
   p->f();       // p zeigt auf ein E-Objekt  => A::f() -> Ausgabe A
                 // da E keine eigene Fkt. f() hat
   p=&f;
   p->f();       // p zeigt auf ein F-Objekt  => F::f() -> Ausgabe F
}
```

```
Ausgabe
A
B
C
D
A
F
```

Eine Funktion ist ab der Klasse in der Vererbungs-Hierarchie virtuell, wo sie in der Klassendefinition zum ersten Mal mit **virtual** deklariert wurde. Wird in den weiteren abgeleiteten Klassen die **virtual** Deklaration weggelassen, so ist die Funktion trotzdem weiterhin virtuell.

Achtung – nur Element-Funktionen können virtuell sein und überschrieben werden. Weder Klassen-Funktionen noch freie Funktionen können virtuell sein, da sie keinen Objekt-Bezug haben, über den die Funktions-Auswahl („Funktions-Dispatch“) zur Laufzeit möglich ist.

17.4.3 Diskussion

Mit Polymorphie ist gemeint, dass eine Funktion vielgestaltig ist, d. h. in Abhängigkeit vom Kontext unterschiedlich (angepasst) reagiert. Genau genommen reagiert natürlich nicht eine Funktion unterschiedlich, sondern es werden unterschiedliche Funktionen aufgerufen, ohne dass sich der Entwickler um die echten Objekt-Typen und deren verschiedene Funktionen-Implementierungen kümmern muss. Dies ermöglicht es ihm, ähnliche Objekte[1] gleich zu behandeln, ohne Details kennen zu müssen (z.B. welche Klassen es gibt, wie sie heissen, wie sie zu behandeln sind, usw...).

Hinweis – im ersten Augenblick sieht Polymorphie nicht nach etwas Besonderem aus, sondern eher nur nach einem kleinen Sprachgag – aber dies ist falsch. Es ist **das Schlüsselkonzept** der Objektorientierung. Seine wahre Mächtigkeit erkennt man meist erst in praktischen Einsätzen, von denen in den weiteren Kapiteln leider nur ein paar folgen werden.

[1] Ähnliche Objekte sind Objekte, die eine gemeinsame Basisklasse haben.

17.5 Beispiel „Obstkorb“

17.5.1 Aufgabe

Nehmen wir an, sie wollen einen Obstkorb implementieren:

- Ein Obstkorb soll einfach mehrere Früchte verschiedener Obstsorten aufnehmen können. Der Obstkorb übernimmt **nicht** den Besitz der Früchte.
- Der Obstkorb hat einen Namen.
- Außerdem soll der Obstkorb einen Konsolen Ausgabe folgender Form haben:
  - Name vom Obstkorb
  - Anzahl der Früchte im Obstkorb
  - Darstellung alle Früchte – alphabetisch sortiert nach dem Namen der Frucht
- Jede Frucht hat einen Namen.
- Die Darstellung einer Frucht besteht aus Name und Obstsorte.
- Für den Anfang begnügen wir uns mit den zwei Obstsorten „Apfel“ und „Birne“.

Hier eine mögliche Beispiel-Ausgabe eines Obstkorbs mit 5 Früchten:

```
Gewünschte Ausgabe - wenn denn der Obstkorb fertig wäre...
Ich bin der Obstkorb "Geschenk" und enthalte 5 Fruechte:
- Bauchiger Adler (Birne)
- Dickes Schwein (Apfel)
- Fetter Kohl (Birne)
- Gruener Baum (Apfel)
- Saftiger Schmatz (Apfel)
```

17.5.2 Lösung

Fangen wir mal wieder mit der Main-Funktion an – wie sollte sie denn aussehen? Z.B. so:

```
int main()
{
   basket b("Geschenk");

   pear  fr1("Bauchiger Adler");
   apple fr2("Dickes Schwein");
   pear  fr3("Fetter Kohl");
   apple fr4("Gruener Baum");
   apple fr5("Saftiger Schmatz");

   b.insert(fr1);
   b.insert(fr2);
   b.insert(fr3);
   b.insert(fr4);
   b.insert(fr5);

   b.print();
}
```

Hiermit ist klar, dass wir eine Klasse „basket“ für den Obstkorb brauchen:

```
class basket
{
public:
   basket(const string& name);

   void insert(const apple&);
```

```
    void insert(const pear&);

    void print() const;
};
```

Implementieren wir die Klasse schnell:

- für den Namen brauchen wir einen String

```
class basket
{
public:
    basket(const string& name) : name_(name) {}

    ...

private:
    string name_;
};
```

- Und für die Früchte einfach einen Vektor.

```
class basket
{
    ...

private:
    string name_;
    vector<???> fruits_;
};
```

Aber einen Vektor für was? Wenn er Äpfel aufnehmen kann, dann passen keine Birnen hinein. Und kann er Birnen aufnehmen, dann bleiben die Äpfel außen vor.

Eine Möglichkeit wäre die Verwendung zweier Vektoren – einen für Äpfel, und den anderen für Birnen. Aber das wird kompliziert, und bei weiteren Obstsorten wird das immer furchtbarer. Eine andere Lösung wäre sicher besser...

Nutzen wir Vererbung und Polymorphie. Sowohl Äpfel als auch Birnen sind Obstsorten („ist-ein“ Beziehung) – also erzeugen wir eine Basis-Klasse „fruit“ und abgeleitete Klassen „apple“ und „pear“, die mit Namen („std::string“) umgehen können.

```
class fruit
{
public:
    fruit(const string& name) : name_(name) {}

private:
    string name_;
};

class apple : public fruit
{
public:
    apple(const string& name) : fruit(name) {}
};

class pear : public fruit
{
public:
    pear(const string& name) : fruit(name) {}
};
```

Jetzt können wir Äpfel und Birnen (und später auch alle anderen von „fruit“ abgeleiteten Obstsorten) gemeinsam behandeln. Implementieren können wir den Obstkorb jetzt mit einem Vektor für „const-fruit-Zeigern“. Machen sie sich klar, dass wir hier Zeiger benötigen:

- „fruit“ Objekte gehen nicht, da beim Einfügen in den Container aus z.B. einem Apfel eine Frucht werden würde – ohne Informationen über den Apfel.
- Das eigentliche Obst-Objekt wird nur dann nicht angetastet, wenn wir mit Zeigern oder Referenzen arbeiten. Aber Referenzen können wir nicht nehmen, da Container keine Referenzen aufnehmen können. Also bleiben nur Zeiger übrig.
- Und const-Zeiger, da wir die Früchte nicht verändern wollen.

```
class basket
{
public:
   basket(const string& name) : name_(name) {}

   void insert(const apple&);
   void insert(const pear&);

   void print() const;

private:
   string name ;
   vector<const fruit*> fruits_;
};
```

Mit der Basis-Klasse „fruit“ kann die Klasse „basket“ noch mehr vereinfacht werden, da sie nun nur noch eine Insert-Funktion benötigt – typisiert auf „const fruit&“. Diese läßt sich jetzt auch direkt inline implementieren:

```
class basket
{
public:
   ...

   void insert(const fruit& fr) { fruits_.push_back(&fr); }

   ...

private:
   string name_;
   vector<const fruit*> fruits_;
};
```

Bleibt noch die Ausgabe offen – beginnen wir mit dem Obstkorb:

```
void basket::print() const
{
   cout << "Ich bin der Obstkorb \"" << name_ << "\" und enthalte "
        << fruits_.size() << " Fruechte:\n";
   for (const fruit* p : fruits_)
   {
      ???
   }
}
```

In der Schleife könnte man jetzt für die jeweiligen Obstsorten eine entsprechende Ausgabe einbauen. Aber mit jeder Veränderung einer Obstsorte (inkl. Löschen bzw. Hinzufügen) müßte die For-Schleife angepasst werden – keine schöne Vorstellung. Statt dessen lassen wir sich jedes Objekt selber ausgeben, und dank Polymorphie ist das ganz einfach: Wir

definieren einfach eine virtuelle Print-Funktion in der Basis-Klasse „fruit“, und überschreiben sie angepasst in den abgeleiteten Klassen „apple“ und „pear“ – z.B. so:

```
class fruit
{
public:
   fruit(const string& name) : name_(name) {}

   virtual void print() const {}

protected:
   const string& name() const { return name_; }

private:
   string name ;
};

class apple : public fruit
{
public:
   apple(const string& name) : fruit(name) {}

   void print() const override { cout << name() << " (Apfel)"; }
};

class pear : public fruit
{
public:
   pear(const string& name) : fruit(name) {}

   void print() const override { cout << name() << " (Birne)"; }
};
```

Und die Print-Funktion in „basket“ sieht jetzt so aus:

```
void basket::print() const
{
   cout << "Ich bin der Obstkorb \"" << name_ << "\" und enthalte "
        << fruits_.size() << " Fruechte:\n";
   for (const fruit* p : fruits_)
   {
      cout << "- ";
      p->print();
      cout << '\n';
   }
}
```

Und wo wir schon mal dabei sind, eine schöne Lösung zu basteln... äh, zu entwickeln – machen wir es doch ordentlich, und liefern dabei gleich noch ein Beispiel für eine virtuelle Indirektion für eine freie Operator-Funktion mit.

Wir wollen die Schleife in der Print-Funktion von „basket“ schöner machen, d.h. den Ausgabe-Operator überladen. Da der erste Operand des Ausgabe-Operators ein „std::ostream“ ist, müssen wir eine freie Operator-Funktion nehmen. Freie Funktionen können nicht virtuell sein – das können nur Element-Funktionen (Kap. 17.4.2). Also müssen wir eine Indirektion nehmen.

```
void basket::print() const
{
   cout << "Ich bin der Obstkorb \"" << name_ << "\" und enthalte "
        << fruits_.size() << " Fruechte:\n";
   for (const fruit* p : fruits_)
   {
```

```
        cout << "- " << *p << '\n';
    }
}
```

```
class fruit
{
public:
    fruit(const string& name) : name_(name) {}

    virtual ostream& print(ostream& out) const { return out; }

protected:
    const string& name() const { return name_; }

private:
    string name ;
};

inline ostream& operator<<(ostream& out, const fruit& fr)
{
    return fr.print(out);
}

class apple : public fruit
{
public:
    apple(const string& name) : fruit(name) {}

    ostream& print(ostream& out) const override { return out << name() << " (Apfel)"; }
};

class pear : public fruit
{
public:
    pear(const string& name) : fruit(name) {}

    ostream& print(ostream& out) const override { return out << name() << " (Birne)"; }
};
```

17.5.3 Zusammenfassung

Und hier nochmal die gesamte Lösung in einem:

```
#include <string>
#include <vector>
#include <ostream>
#include <iostream>
using namespace std;

class fruit
{
public:
    fruit(const string& name) : name_(name) {}

    virtual ostream& print(ostream& out) const { return out; }

protected:
    const string& name() const { return name_; }

private:
    string name_;
};

inline ostream& operator<<(ostream& out, const fruit& fr)
{
    return fr.print(out);
}

class apple : public fruit
{
```

```
public:
   apple(const string& name) : fruit(name) {}

   ostream& print(ostream& out) const override { return out << name() << " (Apfel)"; }
};

class pear : public fruit
{
public:
   pear(const string& name) : fruit(name) {}

   ostream& print(ostream& out) const override { return out << name() << " (Birne)"; }
};

class basket
{
public:
   basket(const string& name) : name_(name) {}

   void insert(const fruit& fr) { fruits .push back(&fr); }

   void print() const;

private:
   string name_;
   vector<const fruit*> fruits ;
};

void basket::print() const
{
   cout << "Ich bin der Obstkorb \"" << name_ << "\" und enthalte "
        << fruits_.size() << " Fruechte:\n";
   for (const fruit* p : fruits )
   {
      cout << "- " << *p << '\n';
   }
}

int main()
{
   basket b("Geschenk");

   pear  fr1("Bauchiger Adler");
   apple fr2("Dickes Schwein");
   pear  fr3("Fetter Kohl");
   apple fr4("Gruener Baum");
   apple fr5("Saftiger Schmatz");

   b.insert(fr1);
   b.insert(fr2);
   b.insert(fr3);
   b.insert(fr4);
   b.insert(fr5);

   b.print();
}
```

17.5.4 Diskussion

Betrachten Sie mal die Abhängigkeiten (Kennen-Beziehungen) der Klassen untereinander, und ihre Konsequenzen bzgl. Wiederverwendung und Veränderbarkeit:

- Die Klasse „fruit“ als sehr einfach Klasse ist vollkommen entkoppelt von ihrer Benutzung (hier dem Obstkorb) und ihren konkreten Ausprägungen (hier Apfel und Banane). Daher sie ist von nichts und niemandem abhängig. Damit ist diese Klasse problemlos wiederverwendbar. Sie stellt einfach eine allgemeine Sicht auf die Abstraktion Obst dar.
- Da die Klasse Obstkorb nur die Klasse Obst kennt, ist hier eine kleine Kopplung vorhanden. Trotzdem kann der Obstkorb mit beliebigen Obstsorten umgehen, ohne diese

kennen zu müssen. Damit können Obstsorten entfernt, hinzugefügt oder ihr Verhalten verändert werden, ohne dass der Obstkorb umprogrammiert werden muss.
- Das Standard-Verhalten von Obst kann in der Basis-Klasse einmal gesammelt werden, kann aber in konkreten Obstsorten verändert werden.
- Alles apfel-spezifische ist in der Klasse Apfel zusammengefasst. Diese Klasse implementiert nur die Klasse Obst - die einzige Kopplung.
- Um das Beispiel um weitere Obstsorten zu erweitern, muss nur eine neue Obstsorten-Klasse von Obst abgeleitet werden und in das Haupt-Programm eingefügt werden. **Das restliche Programm ist vollkommen von dieser Änderung entkoppelt** und muss nicht verändert werden. Analoges gilt beim Entfernen oder Verändern von Obstsorten.

Mit Polymorphie ist es jetzt noch viel viel wichtiger geworden, die Implementierungen auf ihre Abstraktions-Ebenen zu beschränken und die Aufgaben zu delegieren. War es vorher schon sehr sehr schlechter Stil, Objekt-Zustände abzufragen und selber damit zu arbeiten, so wird diese Vorgehensweise jetzt essentiell.

17.6 Destruktoren

Destruktoren sind normale Element-Funktionen:

```
class A
{
public:
   ~A() { cout << "- De. A\n"; }
};

class B : public A
{
public:
   ~B() { cout << "- De. B\n"; }
};

int main()
{
   cout << "Zeiger vom Typ B*\n";
   B* pb = new B();
   delete pb;

   cout << "Zeiger vom Typ A*\n";
   A* pa = new B();
   delete pa;                        // Achtung - undefiniertes Verhalten
}
```

Ausgabe
```
Zeiger vom Typ B*
- De. B
- De. A
Zeiger vom Typ A*
- De. A
```

Wie man an der Ausgabe des Beispiels sieht, wird daher bei Objekten die über Basis-Klassen-Zeiger gelöscht werden, nur der Destruktor der Basisklasse aufgerufen. Ein Destruktor ist halt eine normale Element-Funktion, und wird daher defaultmäßig statisch gebunden, d.h. der Destruktor-Aufruf wird über den statischen Typ festgelegt. Hierbei

können aber auch noch andere Fehler auftreten, die unser Beispiel nicht zeigt. Letztlich ist das Löschen über einen Basisklassen-Zeiger mit einem nicht-virtuellen Destruktor nicht erlaubt und erzeugt „undefined behaviour“.

Die Lösung besteht natürlich darin, den Destruktor virtuell zu machen. Da der implizite Destruktor nicht virtuell ist, müssen wir in diesem Fall immer selber einen erzeugen - und sei er auch leer. Im einfachsten Fall wird in die Basis-Klasse ein leerer virtueller Destruktor eingefügt. Dann wird auch der Destruktor dynamisch gebunden, und damit immer der Destruktor der abgeleiteten Klasse aufgerufen, und außerdem funktioniert die Speicherfreigabe korrekt.

```
class A
{
public:
   virtual ~A() {}
};

class B : public A
{
public:
   virtual ~B() { cout << "- De. B\n"; }
};

int main()
{
   cout << "Zeiger vom Typ B*\n";
   B* pb = new B();
   delete pb;

   cout << "Zeiger vom Typ A*\n";
   A* pa = new B();
   delete pa;
}
```

Ausgabe

```
Zeiger vom Typ B*
- De. B
Zeiger vom Typ A*
- De. B
```

Regeln

- Da Sie nie wissen können, wie Ihre Klasse mal benutzt wird, sollten Sie sich nur von Klassen ableiten, die einen virtuellen bzw. protected Destruktor haben.
- Umgekehrt sollten Sie immer, wenn Sie eine Basis-Klasse entwickeln, einen virtuellen Destruktor implementieren – auch wenn dieser leer ist.
- Sie sollten jetzt nicht übertreiben und bloß nicht jeder Klasse einen virtuellen Destruktor geben. Sobald eine Klasse mindestens eine virtuelle Funktion hat, wird sie größer, ihre Benutzung wird unter Umständen langsamer, und falls sie vorher ein POD war so würde sich das nun ändern.

Hinweis – im Prinzip könnte man auf den virtuellen Destruktor verzichten, wenn man sicher sein könnte, dass Objekte nie dynamisch über einen Basis-Klassen-Zeiger gelöscht werden. In der STL gibt es einige solcher Fälle, bei denen die Basis-Klassen nur z.B. Typedefs zur Verfügung stellen, und ansonsten keine semantische Bedeutung haben. Aber dies sind Ausnahmen – halten Sie sich lieber an obige Regeln, dann sind sie auf der sicheren Seite.

17.7 Abstrakte Basis-Klassen

Wir erinnern uns an unseren Obstkorb mit vielen Früchten, die durch eine Basis-Klasse *„fruit“* repräsentiert wurden. Nur gibt es kein „Obst“ Objekt an sich, sondern nur z. B. Bananen, Äpfel, Birnen, usw. Darum sollten sich von der Klasse *„fruit“* eigentlich keine Objekte erzeugen lassen, da sie keinen Sinn machen.

Außerdem hat man häufig das Problem, dass man nicht weiß, wie man Funktionen in der Basis-Klasse implementieren soll. Die Print-Funktion z.B. in „fruit“ wurde leer gelassen, da jede abgeleitete Klasse sie überschreiben soll, und die Basis-Klasse keine gute Default-Implementierung anbieten kann.

Mit unseren aktuellen Kenntnissen können wir das Erzeugen von Basis-Klassen-Objekten schon sehr einschränken – ist ihnen klar wie? Machen sie doch einfach den oder die Konstruktoren „protected“, dann können sie nur aus der Klasse selber, aus abgeleiteten Klassen, oder von Friends aus aufgerufen werden.

```
class A
{
public:
   virtual ~A() {}

protected:
   A() {}
};
```

Aber diese Lösung hat halt Lücken, und sie adressiert nicht das Problem der *unmöglichen* Funktions-Implementierungen. Aber C++ hat ein Sprachmittel für beide Probleme:

Abstrakte Klassen mit rein virtuellen Funktionen

Abstrakte Klassen sind Klassen, von denen keine Objekte erzeugt werden können. Eine Klasse ist dann abstrakt, wenn sie mindestens eine rein-virtuelle Funktion besitzt (auch durch Vererbung).

Rein-virtuelle Funktionen sind normale virtuelle Funktionen, die in der Basis-Ebene (im Normallfall) keine Implementierung haben und zwingend in den abgeleiteten Klassen überschrieben werden müssen. Deklariert werden sie in der KlassenDefinition mit einem **= 0**.

```
class A
{
public:
   virtual void f() = 0;
};

A a;     // Compiler-Fehler - von abstrakten Klassen kann kein Objekt erzeugt werden
```

```
class A
{
public:
   virtual void f() = 0;
};
```

```
class B : public A { };

class C : public B
{
public:
   virtual void f() {}
};

A a;     // Compiler-Fehler, da abstrakte Klasse
B b;     // Compiler-Fehler, da abstrakte Klasse (durch Vererbung)
C c;     // okay
```

Was für einen Sinn haben abstrakte Klassen?

- Sie verhindern, dass von diesen Klassen Objekte gebildet werden.
- Sie erzwingen, dass bestimmte Funktionen (die rein-virtuellen) überschrieben werden.
- Außerdem stellen sie ein allgemeines Interface für verschiedene Implementierungen dar.

Man bezeichnet Basis-Klassen, vor allem abstrakte Basis-Klassen – im Extrem solche mit nur rein-virtuellen-Funktionen – gerne als Interface. Sie stellen eine allgemeine Sicht auf eine Abstraktion dar (z.B. fruit für alle Obstsorten), und bilden quasi die Schnittstelle zu den konkreten Klassen.

17.8 Dynamic-Cast

In sehr seltenen Fällen ist es nötig, einen Objekt-Zeiger oder eine Objekt-Referenz in der Klassen-Hierarchie hinauf zu den abgeleiteten Klassen zu casten.

Dieser Cast ist – wie eigentlich alle Casts – nicht unproblematisch. Hier ist der Cast aber besonders problematisch, denn es ist zur Compilezeit oft nicht bestimmbar, ob sich der Quell-Ausdruck überhaupt in den Ziel-Typ umwandeln läßt.

```
class A
{
public:
   virtual ~A() {}
   virtual void f();
};

class B : public A {};

class C : public A
{
public:
   void g();
};

void fct(A* pa)
{
   pa->g();      // Wie kann ich daraus einen C-Zeiger machen?
                 // Und was ist, wenn es kein Objekt vom Typ C ist, sondern z.B. ein B?
}
```

Mit **dynamic_cast** existiert in C++ ein Cast für Klassen-Hierarchien. Für ihn gilt:

- Quell- und Ziel-Typ müssen in einer gemeinsamen Klassen-Hierarchie liegen, oder der Zieltyp muss *'void*'* sein. Mit Quell-Typ ist der statische Typ des Quell-Ausdrucks

gemeint.

- Der Quell-Typ muss polymorph sein, d.h. er muss mindestens eine virtuelle Funktion enthalten.
- Der Ziel-Typ muss nicht polymorph sein.
- Ein Dynamic-Cast kann nur auf Zeiger und Referenz-Typen ausgeführt werden.
- Der Dynamic-Cast wird nur ausgeführt, wenn der Quell-Ausdruck dem Ziel-Typ entspricht.
- Die Konvertierung findet zur Laufzeit statt - sie ist daher langsamer als andere Casts.
- Ein Dynamic-Cast geht korrekt mit virtuellen Adressen bei Mehrfach-Vererbung um.
- Ein Dynamic-Cast kann keinen const Modifizierer entfernen.

Wird ein nicht korrekter Cast versucht, so wird:

- bei Zeigern ein Null-Zeiger zurückgegeben, und
- bei Referenzen eine **std::bad_cast** Exception geworfen.

```
// Klassen A, B und C wie oben

void f1(A* pa)
{
   cout << "> f1\n";
   if (C* pc = dynamic_cast<C*>(pa))
   {
      pc->g();
   }
   else
   {
      cout << "  Null-Zeiger\n";
   }
}

void f2(A& ra)
{
   cout << "> f2\n";
   try
   {
      dynamic_cast<C&>(ra).g();
   }
   catch (const std::bad_cast& x)
   {
      cout << "  Exception: " << x.what() << '\n';
   }
}

int main()
{
   A a;
   cout << "A\n";
   f1(&a);
   f2(a);

   B b;
   cout << "\nB\n";
   f1(&b);
   f2(b);

   C c;
   cout << "\nC\n";
   f1(&c);
   f2(c);
}
```

Ausgabe

```
A
> f1
  Null-Zeiger
```

```
> f2
  Exception: Bad dynamic_cast!

B
> f1
  Null-Zeiger
> f2
  Exception: Bad dynamic_cast!

C
> f1
  C::g()
> f2
  C::g()
```

Bemerkung – auch mit den Sicherheiten von „dynamic_cast“ ist ein Cast ein Cast und bleibt ein Cast. Auch wenn er relativ schön und sicher ist. Cast’s sind und bleiben ein Werkzeug für absolute Ausnahme-Situationen. Im Normallfall benötigen sie bei einem ‘vernünftigen’ Design keine Cast’s, auch kein „dynamic_cast“.

17.9 Vererbung & Polymorphie

17.9.1 Semantik

Über die Semantik dieser neuen Sprachmittel lassen sich folgende Faustregeln aufstellen:

- Eine gemeinsame Basis-Klasse bedeutet gemeinsame Aufgaben.
- Öffentliche Erblichkeit bedeutet *"ist-ein"*.
- Eine rein-virtuelle Funktion bedeutet, dass die Schnittstelle der Funktion geerbt wird.
- Eine virtuelle Funktion bedeutet, dass die Schnittstelle und eine Standardimplementierung geerbt werden.
- Eine nicht-virtuelle Funktion bedeutet, dass die Schnittstelle inkl. obligatorischer Funktionen geerbt wird.
- Oberbegriffe (Basis-Klassen) sind ein Hilfsmittel zur Abstraktion und bilden ein gemeinsames Interface aller abgeleiteten Klassen.
- Polymorphie bedeutet, dass eine Funktion, je nach Objekt, angepasst reagiert.

17.9.2 Begriff „Polymorphie“

Mit Vererbung und virtuellen Funktionen wird in C++ Polymorphie (Vielgestaltigkeit) realisiert. Mit Polymorphie ist gemeint, dass eine Funktion vielgestaltig ist, d. h. in Abhängigkeit vom Kontext unterschiedlich (angepasst) reagiert. Genau genommen reagiert nicht eine Funktion unterschiedlich, sondern es werden unterschiedliche Funktionen aufgerufen.

Der Begriff Polymorphie wird in der Literatur sehr unterschiedlich benutzt:

1. Einige bezeichnen schon jeden Aufruf einer Element-Funktion als Polymorphie, da jede Klasse die gleichen Funktions-Namen enthalten kann, und daher in Abhängigkeit vom Objektbezug unterschiedliche Funktionen aufgerufen werden.

2. Manche bezeichnen Überladen als Polymorphie, da hier unterschiedliche Funktionen in Abhängigkeit von den Parametern aufgerufen werden.

3. Ich beschränke mich hier bei dem Begriff Polymorphie (im Einklang mit dem Grossteil der OO Literatur) auf die Wirkungsweise von dynamisch gebundenen (in C++ also virtuellen) Funktionen. Manchmal wird dies in C++ auch dynamische Polymorphie genannt.

17.9.3 Schlüsselkonzepte

Mit Vererbung und vor allem Polymorphie haben wir **die Schlüsselkonzepte** der objektorientierten Programmierung kennengelernt. Immer, wenn sie eine Menge an ähnlichen Dingen, verwalten, bearbeiten, oder sonstwas müssen, bietet sich Polymorphie als eine elegante Lösung an.

Ob Sie nun
- ein Spiel mit *unterschiedlichen* Spielern entwickeln, die auf *unterschiedlichen* Planeten leben, in dem *unterschiedliche* Raumschiffe *mit unterschiedlichen* Waffen vorkommen,...
- oder ein Programm zur Kontoführung *unterschiedlicher* Konten,
- oder, oder, oder....

Immer bieten sich Vererbung und Polymorphie als einfache, elegante und leistungsfähige Designmittel an. Daher finden sie sich auch immer wieder als zentrale Konzepte in Pattern, Frameworks, Bibliotheken und Programmen wieder.

Wenn Sie es schaffen, dass eine Programm-Struktur (die Architektur) nur auf abstrakten Klassen beruht, können Sie die konkreten Implementierungen verändern, ersetzen, löschen, ergänzen und umstrukturieren, ohne dass Sie die Programm-Struktur nur ein einziges Mal ändern müssen. Damit hätten Sie einen sehr hohen Grad an Erweiterbarkeit und Änderbarkeit erreicht.

