Vererbung, das Slicing Problem und Polymorphismus

Thomas Hausberger, Matthias Panny
Ursprünglich erstellt von Sebastian Stabinger

SS2022

Das Problem

Beispiel

- Wir wollen Klassen für verschiedene Elemente eines Spiels deklarieren (z.B. Spieler, Ladenbesitzer, Objekt, . . .)
- Viele Aspekte dieser Klassen sind gleich oder zumindest ähnlich. Alle haben eine aktuelle Position und ein Bild das angezeigt werden soll
- Ein paar Aspekte sind aber anders. z.B. hat ein Spieler evtl. Lebenspunkte und kann sich bewegen. Ein Ladenbesitzer hat ein Inventar das er verkaufen kann und ein Objekt kann passierbar sein oder nicht
- Mit unserem aktuellen Wissen müssten wir die gemeinsamen Eigenschaften in der Spieler-, Ladenbesitzer- und Objekt-Klasse manuell wiederholen.

Beispiel Code — Spieler

```
class Player {
private:
  string _image;
  int _xpos, _ypos, _health;

public:
  Player(int xpos, int ypos, int health, string image) {
    _xpos = xpos;
    _ypos = ypos;
    _image = image;
    _health = health;
  }

  void move_up() { ypos--; }
  void move_down() { ypos++; }
  void move_left() { xpos--; }
  void move_right() { xpos++; }

  bool isdead() { return health <= 0; }
  void draw() { draw_img(_image, _xpos * 16, _ypos * 16); }
};
```

Beispiel Code — Ladenbesitzer

```
class Shopkeep {
private:
  string _image;
  int _xpos, _ypos;
  vector<string> _inventory;

public:
  Shopkeep(int xpos, int ypos, string image, vector<string> inventory) {
    _xpos = xpos;
    _ypos = ypos;
    _image = image;
    _inventory = inventory;
  }

  vector<string> get_inventory() { return _inventory; }

  void draw() { draw_img(_image, _xpos * 16, _ypos * 16); }
};
```

Beispiel Code — Objekt

```
class Object {
private:
  int _xpos, _ypos;
  string _image;
  bool _solid;

public:
  Object(int xpos, int ypos, string image, bool solid) {
    _xpos = xpos;
    _ypos = ypos;
    _image = image;
    _solid = solid;
  }

  bool issolid() { return solid; }
  void draw() { draw_img(_image, _xpos * 16, _ypos * 16); }
};
```

Die Lösung — Vererbung

Vererbung

- Mittels Vererbung kann eine Klasse (die abgeleitete Klasse) Variablen und Memberfunktionen einer anderen Klasse (der Basisklasse) erben und somit verwenden.
- Wir erweitern und/oder ändern also die Funktionalität einer bereits existierenden Klasse (die Basisklasse) und geben dieser geänderten Klasse (die abgeleitete Klasse) einen neuen Namen.
- Wir sagen die abgeleitete Klasse erweitert die Basisklasse

Syntax

```
class AbgeleiteteKlasse : public Basisklasse {
  // ... Definition der abgeleiteten Klasse
}
```

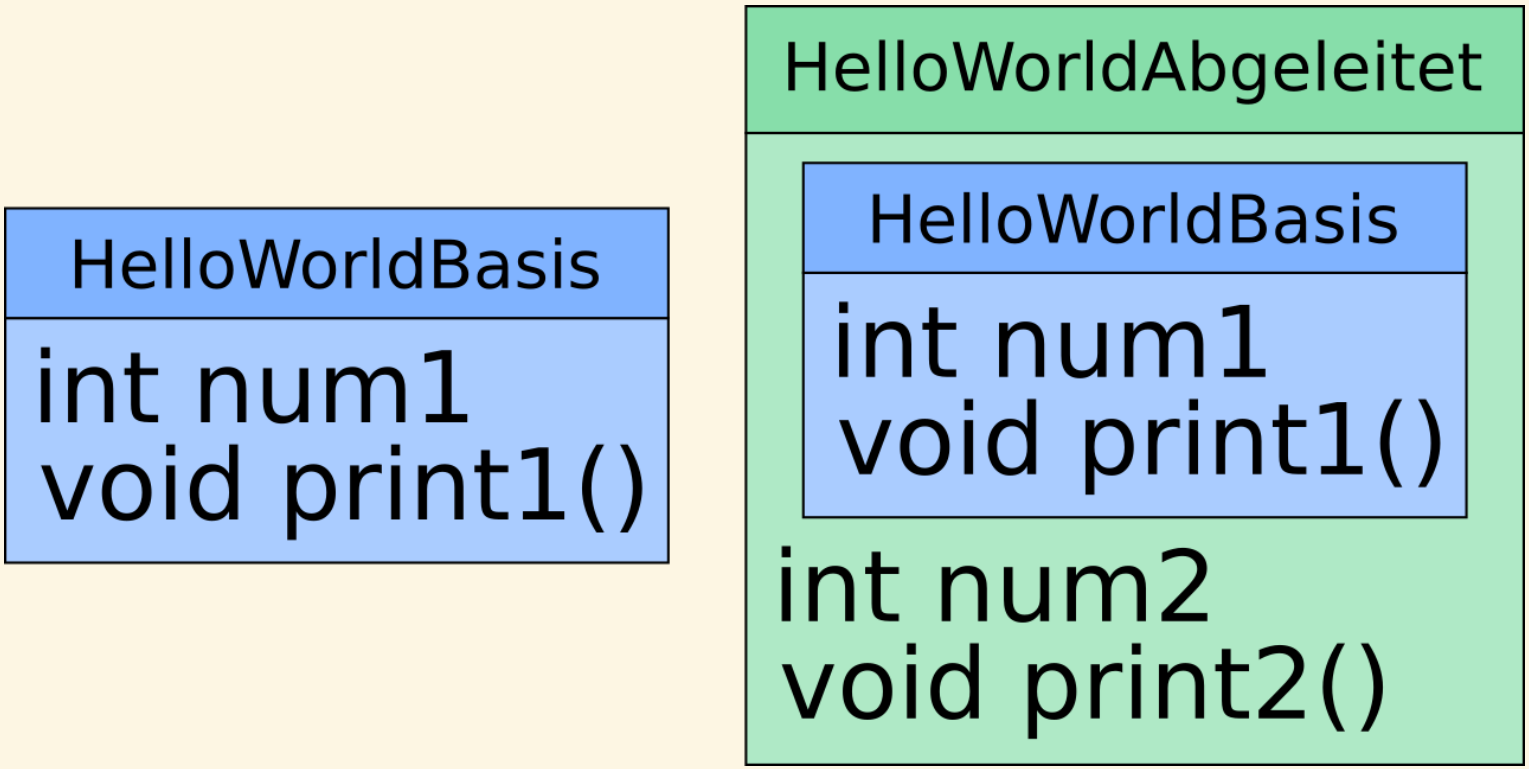
- Achtung: Die abgeleitete Klasse hat nur Zugriff auf Variablen und Memberfunktionen der Basisklasse welche `public` sind!

Vererbung — Kurzes Beispiel

```cpp
#include <iostream>
using namespace std;

class HelloWorldBasis {
public:
  int num1 = 42;
  void print1() { cout << "Hallo: Basisklasse " << num1 << endl; }
};

class HelloWorldAbgeleitet : public HelloWorldBasis {
public:
  int num2 = 23;
  void print2() { cout << "Hallo: Abgeleitete Klasse " << num1 << endl; }
};

int main() {
  HelloWorldBasis w1;
  w1.print1(); // Output: Hallo : Basisklasse 42
  HelloWorldAbgeleitet w2;
  w2.print2(); // Output: Hallo : Abgeleitete Klasse 42
  // Die Funktion print1 ist auch in der abgeleiteten Klasse
  w2.print1(); // Output: Hallo : Basisklasse 42
  cout << "num = " << w2.num1 << endl;  // 42
  cout << "num2 = " << w2.num2 << endl; // 23
}
```

Vererbung — Kurzes Beispiel

Grafisch kann man sich die Vererbung folgendermaßen vorstellen (Die Basisklasse ist in die abgeleitete Klasse eingebettet):



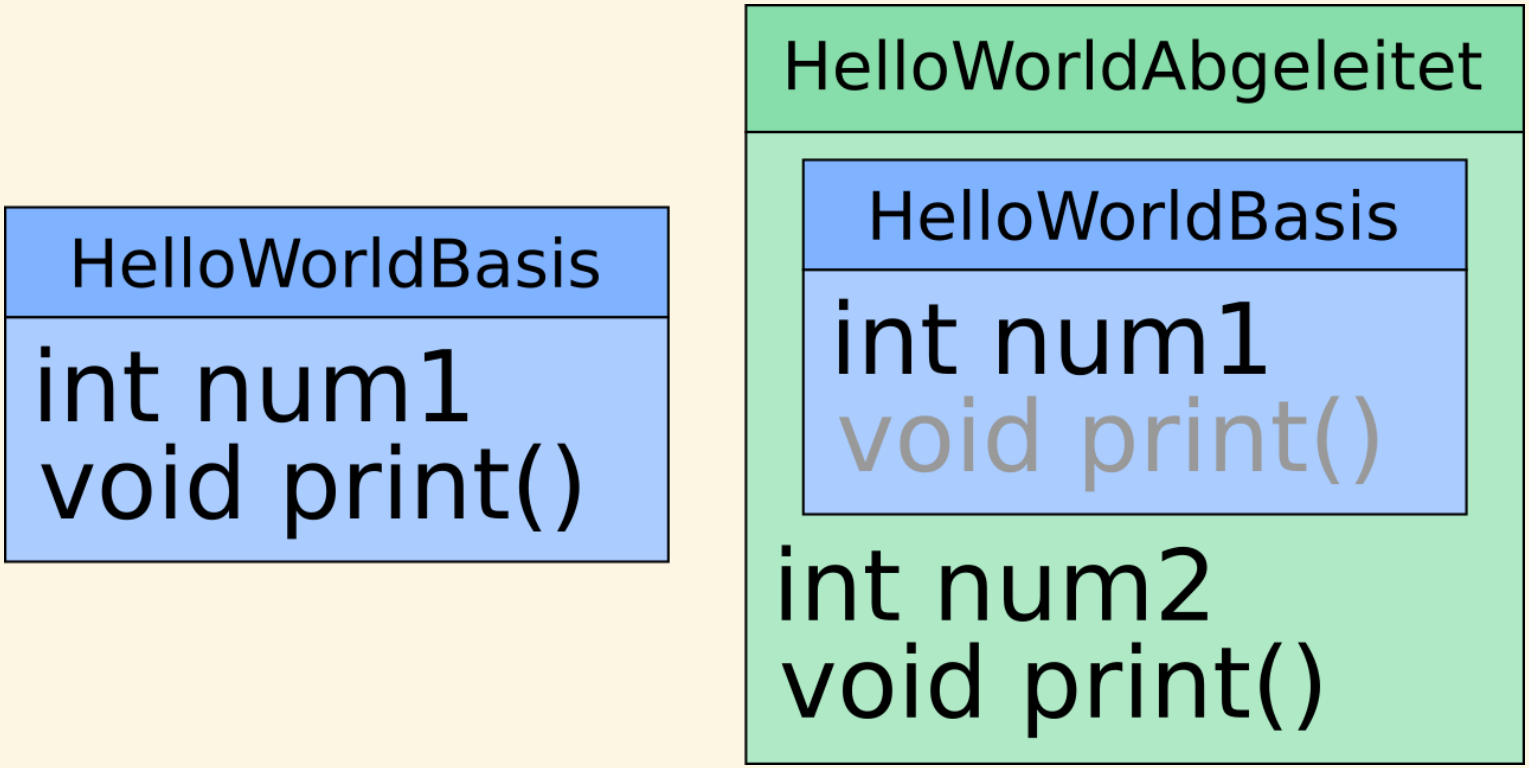
Überladen von Funktionen

Funktionsweise

Wir können eine Funktion der Basisklasse in der abgeleiteten Klasse überschreiben indem wir sie gleich benennen.

```cpp
#include <iostream>
using namespace std;

class HelloWorldBasis {
public:
  int num1 = 42;
  void print() { cout << "Hallo: Basisklasse " << num1 << endl; }
};

class HelloWorldAbgeleitet : public HelloWorldBasis {
public:
  int num2 = 23;
  void print() { cout << "Hallo: Abgeleitete Klasse " << num1 << endl; }
};

int main() {
  HelloWorldBasis w1;
  w1.print(); // Output: Hallo : Basisklasse 42
  HelloWorldAbgeleitet w2;
  w2.print();  // Output: Hallo : Abgeleitete Klasse 42
  cout << "num = " << w2.num1 << endl;  // 42
  cout << "num2 = " << w2.num2 << endl; // 23
}
```

Überladen von Funktionen



- Das System sucht sozusagen von Außen nach Innen und führt die erste Funktion aus welche hinsichtlich Namen und Parametern passt

Verwendung überschriebener Funktionen

Problem

- Wir haben in `HelloWorldAbgeleitet` das vererbte `print` von `HelloWorldBasis` überschrieben
- Wir wollen irgendwo in `HelloWorldAbgeleitet` trotzdem das `print` von `HelloWorldBasis` aufrufen
- Was z.B. häufig vorkommt: Wir erweitern eine Funktion der Basisklasse indem wir sie überschreiben. In der neuen Funktion rufen wir die überschiebene Funktion der Basisklasse auf.

Lösung

- Wir können auf die Basisklasse wie auf einen Namespace zugreifen.
- Bsp. `HelloWorldBasis::print()`

Verwendung überschriebener Funktionen

```cpp
#include <iostream>
using namespace std;

class HelloWorldBasis {
public:
  int num1 = 42;
  void print() { cout << "Hallo: Basisklasse " << num1 << endl; }
};

class HelloWorldAbgeleitet : public HelloWorldBasis {
public:
  int num2 = 23;
  void print() {
    HelloWorldBasis::print();
    cout << "Hallo: Abgeleitete Klasse " << num1 << endl;
  }
};

int main() {
  HelloWorldAbgeleitet w2;
  w2.print();
  // Output:
  // Hallo : Basisklasse 42
  // Hallo : Abgeleitete Klasse 42
}
```

Konstruktoren und Vererbung

Standardkonstruktoren

Der Standardkonstruktor einer Basisklasse wird automatisch aufgerufen wenn die abgeleiteten Klasse erzeugt wird.

```
#include <iostream>

class C1 {
public:
  int i;
  C1() { i = 23; }
};

class C2 : public C1 {
public:
  int j;
  C2() { j = 42; }
};

int main() {
  C2 c;
  std::cout << c.i << " " << c.j << std::endl; // Output: 23 42
}
```

Konstruktoren mit Parametern

Ein Konstruktor der Basisklasse welcher Parameter erwartet muss explizit aufgerufen werden. Dies geschieht durch Anhängen mittels : an den Konstruktor.

Syntax — Beispiel

```
class C1 {
public:
  int a;
  C1(int pa) { a = 2 * pa; }
};

class C2 : public C1 {
public:
  C2(int i) : C1(3 * i) { a = a + 10; }
};

int main() {
  C2 c(4); // c.a == 34 [(2*(3*4))+10]
  cout << c.a << endl;
}
```

Ein größeres Beispiel

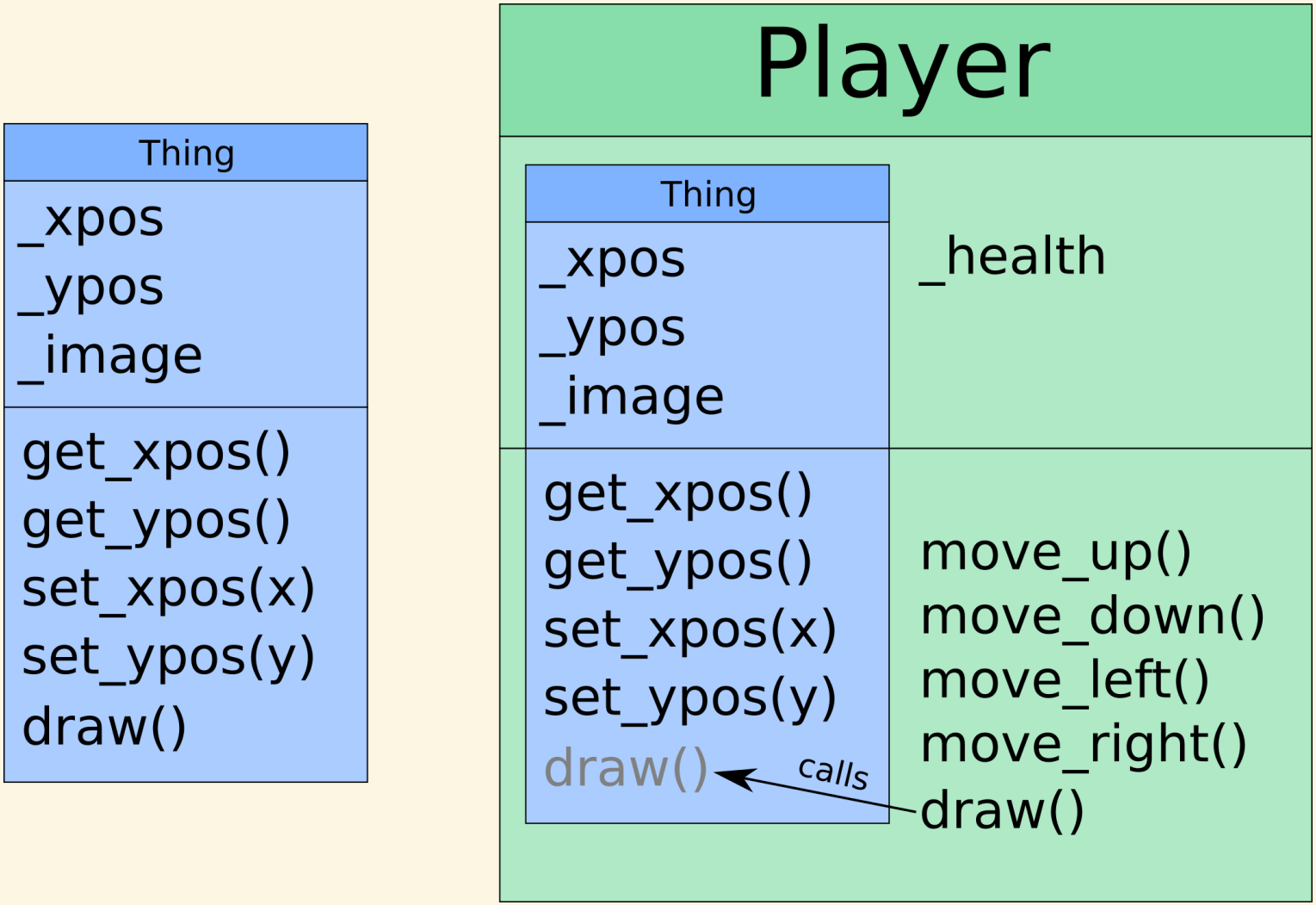
Code Beispiel — Ein Ding

```
class Thing {
private:
  int _xpos, _ypos;
  string _image;

public:
  Thing(int xpos, int ypos, string image) {
    _xpos = xpos;
    _ypos = ypos;
    _image = image;
  }

  int get_xpos() { return _xpos; }
  int get_ypos() { return _ypos; }
  void set_xpos(int x) { _xpos = x; }
  void set_ypos(int y) { _ypos = y; }

  void draw() { draw_image(_image, _xpos * 16, _ypos * 16); }
};
```

Code Beispiel — Spieler

```
class Player : public Thing {
private:
  int _health;

public:
  Player(int xpos, int ypos, int health, string image)
      : Thing(xpos, ypos, image) {
    _health = health;
  }

  void move_up() { set_ypos(get_ypos() - 1); }
  void move_down() { set_ypos(get_ypos() + 1); }
  void move_left() { set_xpos(get_xpos() - 1); }
  void move_right() { set_xpos(get_xpos() - 1); }

  void draw() { // Erweitere draw Funktion von Thing
    Thing::draw();
    // Zeichne Gesundheitsanzeige ...
  }
};
```

Graphische Visualisierung — Thing - Player



Code Beispiel — Ladenbesitzer

```
class Shopkeep : public Thing {
private:
  vector<string> _inventory;

public:
  Shopkeep(int xpos, int ypos, string image, vector<string> inventory)
      : Thing(xpos, ypos, image) {
    _inventory = inventory;
  }

  vector<string> get_inventory() { return _inventory; }
};
```

Code Beispiel — Objekt

```
class Object : public Thing {
private:
  bool _solid;

public:
  Object(int xpos, int ypos, string image, bool solid)
      : Thing(xpos, ypos, image) {
    _solid = solid;
  }

  bool issolid() { return _solid; }
};
```

Kontrolle der Sichtbarkeit

Das Problem

- Wir haben bereits gelernt: Die abgeleitete Klasse hat Zugriff auf alle Elemente der Basisklasse welche `public` sind. Auf die `private` Elemente hat sie keinen Zugriff.
- Wir wollen aber häufig in der abgeleiteten Klasse direkt auf Elemente der Basisklasse zugreifen welche von außerhalb nicht sichtbar sind.

Beispiel

```
class C1 {
private:
  int secret;
};

class C2 : public C1 {
public:
  void change_secret(int newsecret) { secret = newsecret; }
  // Error!! Kein Zugriff auf secret
};
```

Wir könnten `secret` in den `public` Bereich schieben, aber dann kann jeder darauf zugreifen . . .

Die Lösung: protected

- Um dieses Problem zu lösen gibt es einen dritten Sichtbarkeitsbereich innerhalb einer Klasse namens `protected`
- Er verhält sich von Außerhalb so wie `private`, aber eine abgeleitete Klasse kann direkt darauf zugreifen.

Beispiel

```
class C1 {
protected:
  int secret;
};

class C2 : public C1 {
public:
  void change_secret(int newsecret) { secret = newsecret; }
};

int main() {
  C2 c;
  c.change_secret(42); // Alles OK
  c.secret;            // Error!! Kein Zugriff auf secret
}
```

Übungen

Gemeinsame Übung

Wir schreiben die beim letzten Termin entwickelte Version des Spiels so um, dass es eine Spieler- und eine Monsterklasse gibt.

- Der Spieler bekommt eine Gesundheitsanzeige
- Die Monster verschwinden wie gehabt nach einer gewissen Zeit, wir lösen das dieses mal aber in der Monsterklasse selbst

Casten

Casten "normaler" Datentypen

Casten einer "normalen" Variable konvertiert (so gut wie möglich) den Inhalt einer Variable in einen anderen Datentyp

```
int i1 = 23;
double d1 = (double)i; // Konvertiert i explizit nach double
double d2 = 23.42;
int i2 = d2; // Hier wird implizit von double nach int konvertiert
```

Der Upcast

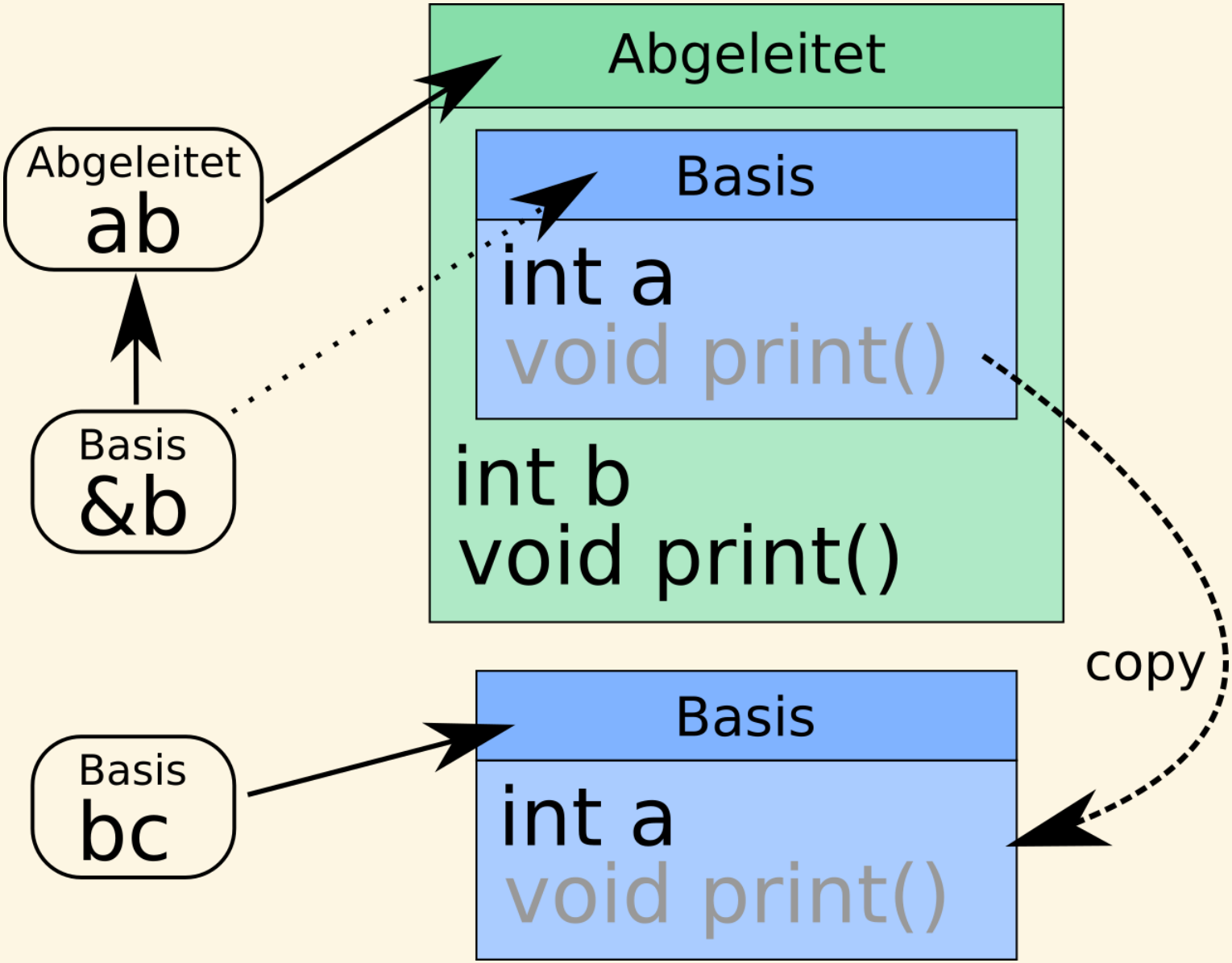
Upcast

- Wenn sich Klassen in einer Vererbungshierarchie befinden, kann ohne weiteres von einer abgeleiteten Klasse zu einer Basisklasse gecastet werden
- Das ist kein Problem, weil bei der abgeleiteten Klasse ja nur Sachen zur Basisklasse hinzugekommen sind
- Man bezeichnet so eine Konvertierung von einer abgeleiteten Klasse zu einer Basisklasse als upcast (weil man in der Klassenhierarchie nach Oben wandert)
- Wir werden uns solche Upcasts an Hand von impliziten casts anschauen (also Konvertierungen die automatisch passieren)

Der Upcast — Beispiel

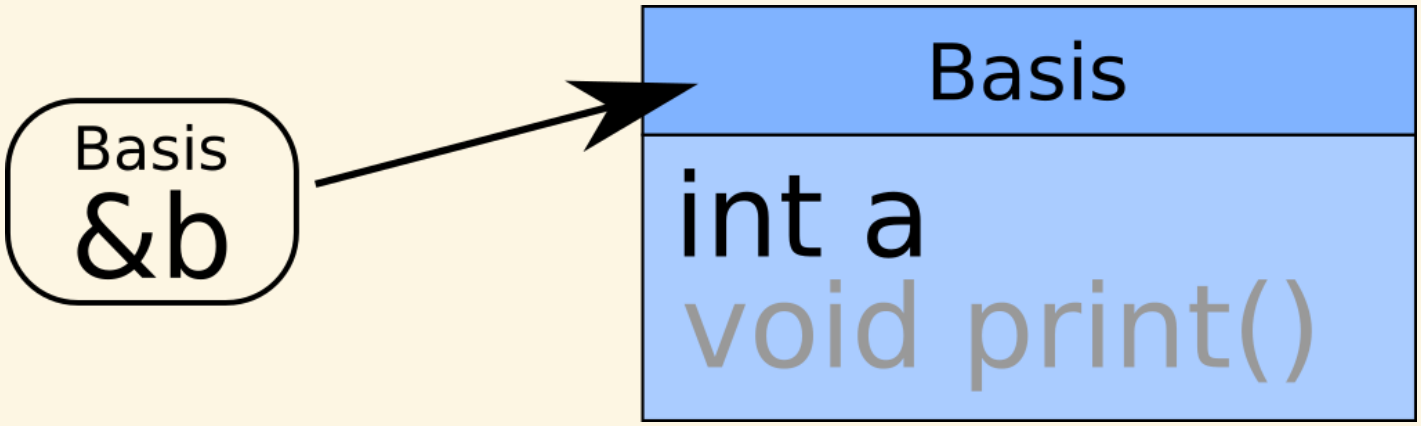
```
#include <iostream>
using namespace std;

class Basis {
public:
  int a;
  void print() { cout << "Basisklasse mit Nummer " << a << endl; }
};

class Abgeleitet : public Basis {
public:
  int b;
    void print() { cout << "Abgeleitete Klasse mit Nummern " << a
                        << " und " << b << endl; }
};

int main() {
  Abgeleitet ab;
  ab.a = 42; ab.b = 23;
  ab.print(); // Abgeleitete Klasse mit Nummer 42 und 23
  Basis bc = ab;
  bc.print(); // Basisklasse mit Nummer 42
  Basis &b = ab;
  b.print(); // Basisklasse mit Nummer 42
}
```

Der Upcast — Graphische Visualisierung



Das Slicing Problem

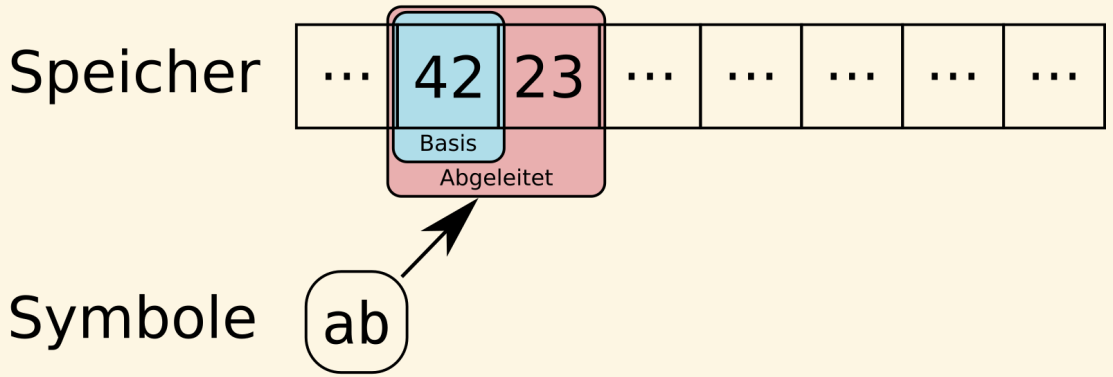
- Aus Sicht der Referenz `b` haben wir es nun mit einer Instanz der Klasse `Basis` zu tun.



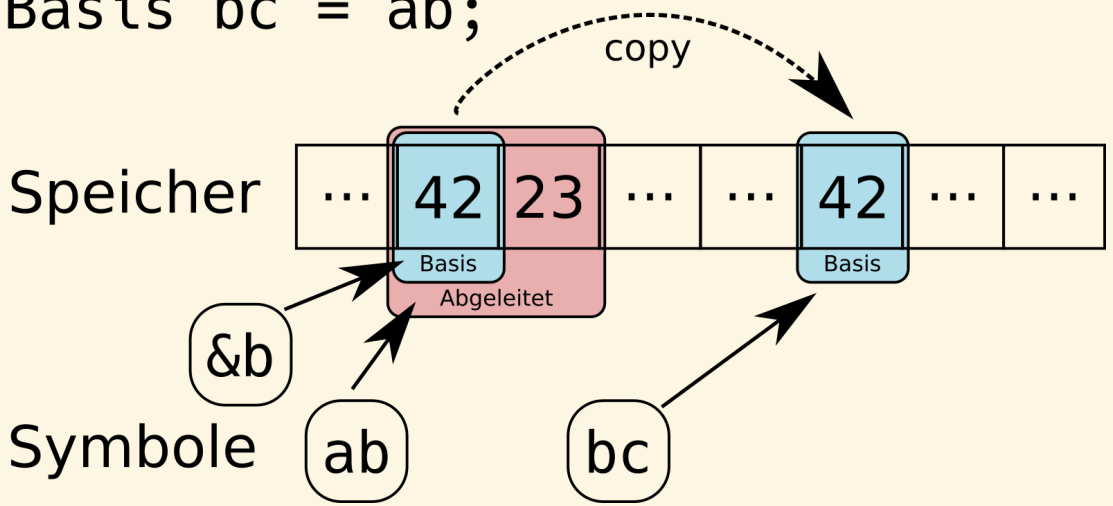
- `b` hat also "vergessen", dass es auf einen Teil einer `Abgeleitet`-Klasse verweist
- Beim kopieren für die Variable `bc` wurde nur der Basis-Teil kopiert
- Man bezeichnet das als Slicing-Problem, weil Teile einer Klasse tatsächlich, oder scheinbar abgeschnitten werden

Der Upcast — Was passiert

```
Abgeleitet ab;
ab.a = 42; ab.b = 23;
```




```
Basis bc = ab;
```



Der Upcast — Parameter Beispiel

```cpp
#include <iostream>
using namespace std;

class Basis {
public:
  int a;
  void print() { cout << "Basisklasse mit Nummer " << a << endl; }
};

class Abgeleitet : public Basis {
public:
  int b;
    void print() { cout << "Abgeleitete Klasse mit Nummern " << a
                        << " und " << b << endl; }
};

void callmyprint(Basis &b) { b.print(); }

int main() {
  Abgeleitet ab;
  ab.a = 42; ab.b = 23;
  ab.print(); // Abgeleitete Klasse mit Nummern 42 und 23
  callmyprint(ab); // Basisklasse mit Nummer 42
  // ab wird implizit in Basis konvertiert
}
```

Effekt eines Upcasts

- Wird eine abgeleitete Klasse in den Typ einer Basisklasse konvertiert, vergisst die Instanz was sie vorher einmal war und verhält sich dann als wäre es schon immer eine Basisklasse gewesen
- Dieses Verhalten bezeichnet mal als slicing und ist oft nicht was man will
- Auf den nächsten Slides werden wir uns das Problem näher ansehen und mit sogenanntem Polymorphismus eine Lösung finden

Polymorphismus

Ein kleines Beispiel

```
#include <iostream>
using namespace std;

class Basis {
public: void print() { cout << "Hallo von der Basisklasse" << endl; }
};
class Abgeleitet1 : public Basis {
public: void print() { cout << "Hallo von der abgeleiteten Klasse 1" << endl; }
};
class Abgeleitet2 : public Basis {
public: void print() { cout << "Hallo von der abgeleiteten Klasse 2" << endl; }
};

void print_with_introduction(Basis &a) {
  cout << "Unsere Klasse möchte folgendes sagen: " << endl;
  a.print();
}

int main() {
  Basis b; Abgeleitet1 a1; Abgeleitet2 a2;

  print_with_introduction(b);
  print_with_introduction(a1);
  print_with_introduction(a2);
}
```

Was ist das Problem?

- Wir hätten gerne, dass sich die Klassen mit ihren tatsächlichen `print` Funktionen melden und nicht mit der Standardimplementierung der Basisklasse
- Das funktioniert aber auf Grund des Slicing-Problems nicht

Lösung

Die Lösung ist das Schlüsselwort `virtual`. Wenn in einer Basisklasse vor einer Funktion `virtual` steht, so merkt sich die Klasse welche Funktion einer abgeleiteten Klasse tatsächlich aufgerufen werden muss.

Warum ist virtual nicht Standard?

Der Aufruf einer `virtual`-Funktion ist langsamer, weil das System zuerst nachsehen muss welche Funktion tatsächlich aufgerufen werden muss. Zudem verbraucht eine `virtual`-Funktion Speicher in jeder Instanz einer Klasse, weil irgendwo gespeichert werden muss welche Funktion wirklich aufzurufen ist.

Virtual — Beispiel

```
#include <iostream>
using namespace std;

class Basis {
public: virtual void print() { cout << "Hallo von der Basisklasse" << endl; }
};
class Abgeleitet1 : public Basis {
public: void print() { cout << "Hallo von der abgeleiteten Klasse 1" << endl; }
};
class Abgeleitet2 : public Basis {
public: void print() { cout << "Hallo von der abgeleiteten Klasse 2" << endl; }
};

void print_with_introduction(Basis &a) {
  cout << "Unsere Klasse mÃ¶chte folgendes sagen: " << endl;
  a.print();
}

int main() {
  Basis b; Abgeleitet1 a1; Abgeleitet2 a2;

  print_with_introduction(b);
  print_with_introduction(a1);
  print_with_introduction(a2);
}
```

Wann verwendet man Polymorphismus

- Man hat eine Basisklasse die ein bestimmtes Verhalten implementiert
- Davon leitet man mehrere Klassen ab die dieses Verhalten erweitern
- Man kann jetzt eine Funktion für die Basisklasse schreiben, und als Parameter alle abgeleiteten Klassen verwenden, oder ...
- wir können in einem Vektor oder in einem Array vom Typ der Basisklasse auch alle abgeleiteten Klassen speichern (als Zeiger oder Referenz)

Beispiel — Vektor mit Zeigern

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

class Basis {
public: virtual void print() { cout << "Hallo von der Basisklasse" << endl; }
};
class Abgeleitet : public Basis {
public: void print() { cout << "Hallo von der abgeleiteten Klasse 1" << endl; }
};

int main() {
  vector<Basis *> vec;
  Basis b1, b2;
  Abgeleitet a1, a2;
  vec.push_back(&b1);
  vec.push_back(&b2);
  vec.push_back(&a1);
  vec.push_back(&a2);

  for (auto &e : vec)
    (*e).print();
}
```

Beispiel — Vektor mit Referenzen

```cpp
#include <functional>
#include <iostream>
#include <vector>

using namespace std;

class Basis {
public:
  virtual void print() { cout << "Hallo von der Basisklasse" << endl; }
};
class Abgeleitet : public Basis {
public:
  void print() { cout << "Hallo von der abgeleiteten Klasse 1" << endl; }
};

int main() {
  vector<reference_wrapper<Basis>> vec;
  Basis b1, b2;
  Abgeleitet a1, a2;
  vec.push_back(b1); vec.push_back(b2);
  vec.push_back(a1); vec.push_back(a2);

  for (Basis &e : vec)
    e.print();
}
```

Übung — Employee und Manager

- Implementieren Sie die beiden Klassen `Employee` und `Manager`
- `Employee` ist die Basisklasse, `Manager` ist die abgeleitete Klasse. Ein Manager ist also auch ein Angestellter
- `Employee` hat eine Variable `salary` die speichert wie viel der jeweilige Mitarbeiten verdient
- `Employee` hat eine Funktion `raise` mit der man einem Mitarbeiter eine Gehaltserhöhung geben kann. Damit der unsere Mitarbeiter nicht zu viel verdienen, wird der maximale Gehalt auf `3500` beschränkt
- `Employee` hat auch eine Funktion `print` die ausgibt: `DerMitarbeiter hat einen Gehalt von ...!`
- Im `Manager` wird die Funktion `raise` überladen und das Gehalt ist nicht beschränkt

Verwendung

- Erzeugen Sie einen Vektor und füllen Sie ihn mit einigen `Employee` sowie `Manager` Instanzen (entweder mit Zeigern oder Referenzen)
- Iterieren Sie über den Vektor und rufen `raise` sowie `print` auf und beobachten Sie das Verhalten

Hinweis

- Polymorphismus ist ein weiterführendes Thema der Objektorientierten Programmierung und uns fehlen einige Grundlagen um das Konzept wirklich gut nützen zu können (z.B. dynamische Speicherverwaltung)
- Es geht in erster Linie darum, dass sie das Konzept gehört haben und sich zumindest ungefähr vorstellen können um was es geht
- Sie müssen Polymorphismus im Abschlussprojekt nicht verwenden.