Zeiger, Referenzen, das Slicing Problem und Polymorphismus

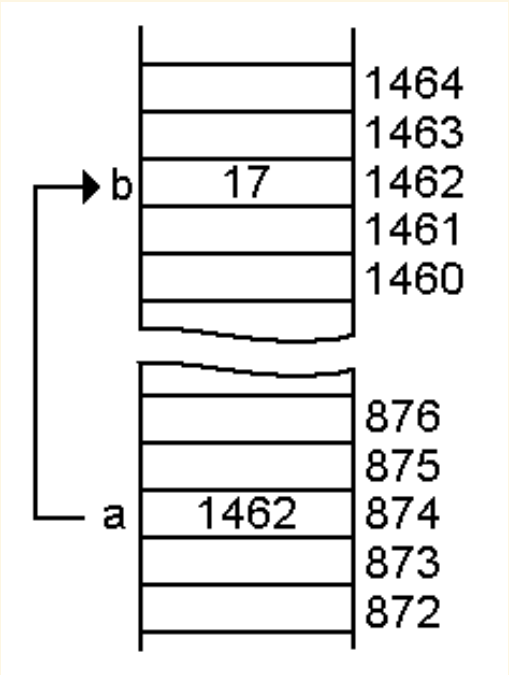
Sebastian Stabinger

SS2019

Zeiger

Zeiger in C

- Zeiger sind Variablen welche Adressen speichern können (also normalerweise 64bit Integer)
- Zeiger haben einen Typ
  - Der Typ eines Zeigers gibt an, was an der Adresse auf die gezeigt wird gespeichert ist.
  - Im Zeiger selbst wird unabhängig vom Typ immer das gleiche gespeichert (eine Adresse)



Beispiel und Probleme

Beispiel

```
int intVar = 10;
int *intPtr = &intVar;
cout << intVar << endl;
cout << *intPtr << endl;
intVar = 20;
cout << *intPtr << endl;
```

Probleme

```
int *intPtr2;
// zufaelliger speicherzugriff (pointer wird nicht initialisiert)
cout << *intPtr2 << endl;
int *intPtr3 = NULL;
// zugriff auf speicheradresse 0 (ueblicherweise segmentation fault)
cout << *intPtr3 << endl;
```

Probleme mit Zeigern

Probleme

1. Ein Zeiger muss nicht auf ein gültiges Objekt im Speicher zeigen
2. Ein Zeiger kann auch auf nichts zeigen (wenn `NULL` als Adresse gespeichert ist)
3. Die Syntax von Zeigern ist am Anfang verwirrend (`*`, `&`, ...)

Vorteile

Nachdem Zeiger ein sehr einfaches Konzept sind (eine Variable welche eine Adresse speichert) ist es auch ein extrem mächtiges Werkzeug

Referenzen

Eigenschaften

Referenzen verweisen wie Zeiger auf Objekte im Speicher

Vorteile von Referenzen

- Referenzen lösen einige der Probleme mit Pointern
- Typprüfung des Compilers kann nicht mehr so leicht übergangen werden
- Referenzen können wie normale Variablen verwendet werden

Nachteile von Referenzen

- Referenzen sind weniger flexibel als Zeiger
  - Keine Zeigerarithmetik
  - Keine Möglichkeit direkt auf die Adresse zuzugreifen

Referenzen — Syntax

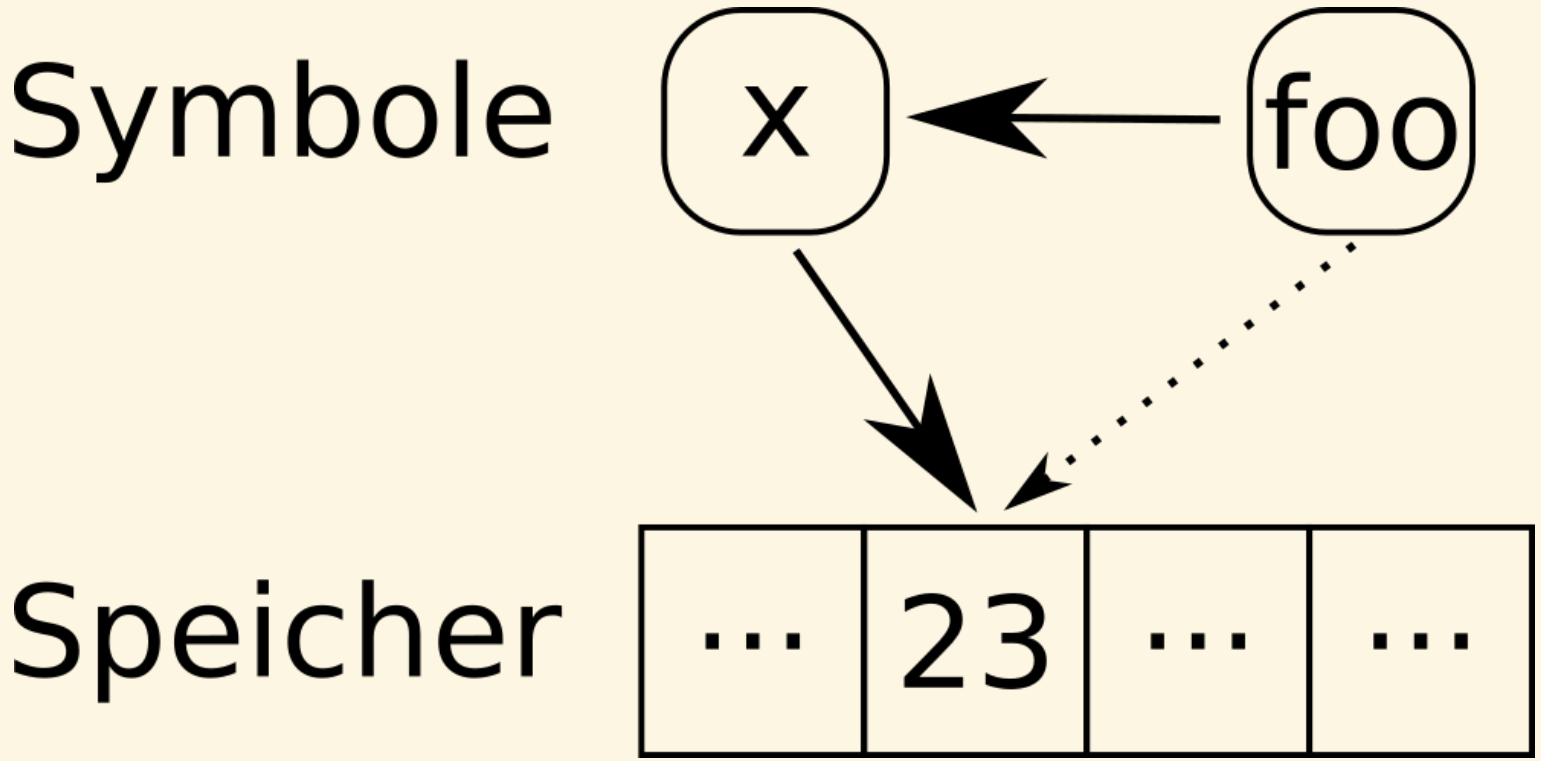
- Um eine Referenz zu erzeugen stellt man ein & an den Anfang eines Variablen- oder Parameternamens (Äquivalent zu dem * bei einem Zeiger)
- Bei Variablen muss zudem in der gleichen Zeile eine Referenz auf eine andere Variable zugewiesen werden!
- Um eine Referenz auf eine Variable zeigen zu lassen muss nicht wie bei Zeigern zuerst die Variable mittels & referenziert werden

```
int x = 23;
int &foo = x;
// foo ist jetzt eine Referenz auf x
// (foo und x enthalten immer den gleichen Wert)
```

- Auf eine Referenz wird genauso zugegriffen wie auf eine gewöhnliche Variable (es ist kein Dereferenzieren mit einem * notwendig wie bei einem Zeiger)

```
foo = 42;
std::cout << foo << " " << x << std::endl;
```

Referenzen — Graphische Visualisierung



Vergleich zwischen Zeiger und Referenz

```
// Erzeugen von Variable, Zeiger und Referenz
int intVar = 10;
int *intPtr = &intVar;
int &intRef = intVar;
// Auslesen von Variable, Zeiger und Referenz
cout << intVar << endl;
cout << *intPtr << endl;
cout << intRef << endl;
// Zuweisen an Variable, Zeiger und Referenz
intVar = 20;
*intPtr = 30;
intRef = 40;

cout << intVar << endl; // Ausgabe = 40
```

Man sieht also, dass sich Referenzen genauso verwenden lassen wie Variablen, aber zu großen Teilen die Funktionalität eines Zeigers haben

Referenzen als Parameter

- Referenzen als Variablen in "normalem" Code sind eher unüblich
- Referenzen werden am häufigsten bei Parametern von Funktionen verwendet:
  - Die übergebenen Parameter müssen dadurch nicht kopiert werden was gerade bei großen Klassen schneller ist
  - Innerhalb der Funktion können Änderungen an den Parametern vorgenommen werden welche auch ausserhalb der Funktion sichtbar sind. Normalerweise funktioniert das nicht, weil die Änderungen nur an einer Kopie vorgenommen werden.

Swap mit normalen Parametern, Zeigern, Referenzen

Normale Parameter

```
void swap(int p1, int p2) {
  int temp = p1; p1 = p2; p2 = temp;
}
int a = 2, b = 3; swap(a, b);   // Verwendung
```

Sieht einfach aus, funktioniert aber auch einfach nicht ...

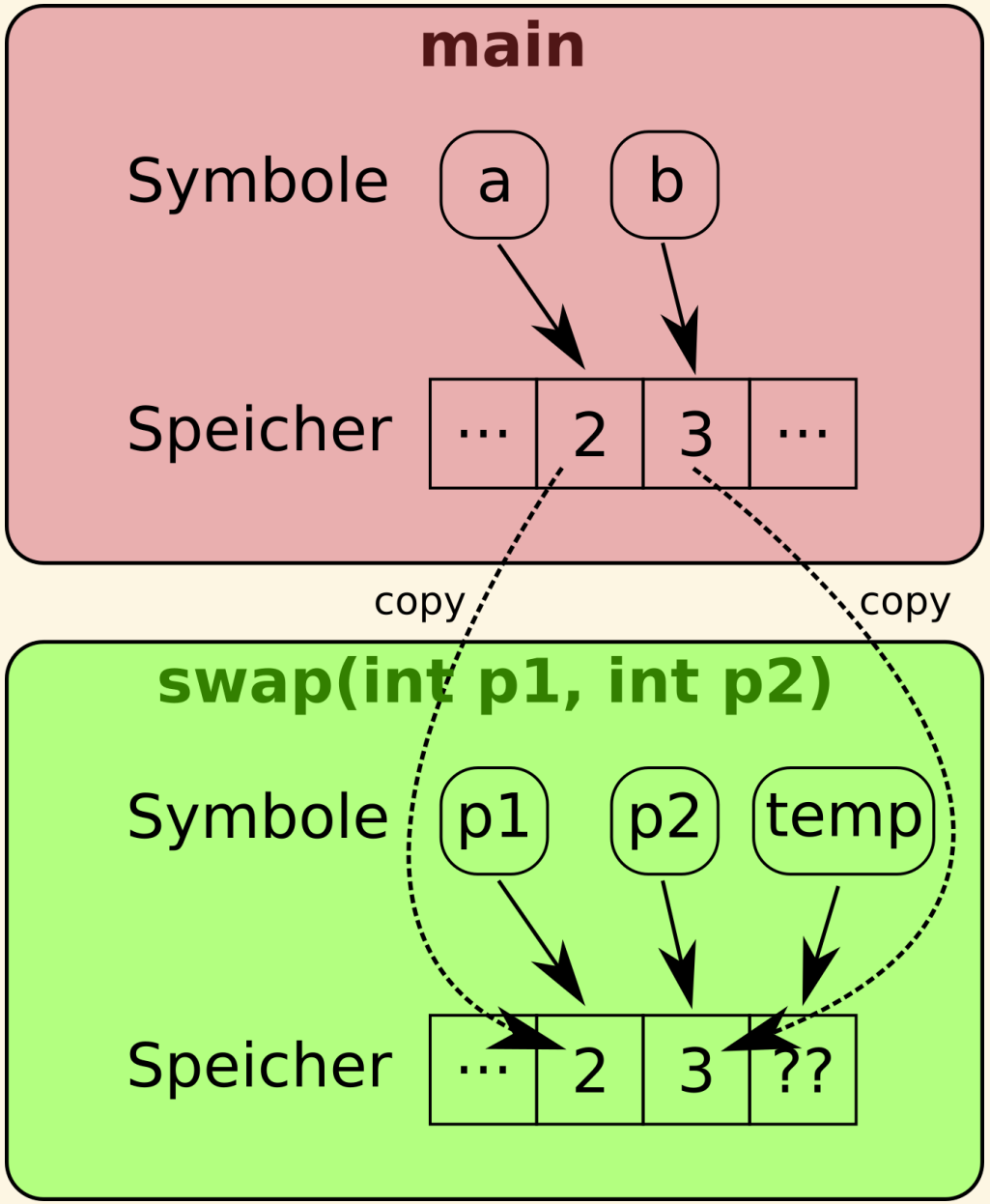
Zeiger

```
void swap(int *p1, int *p2) {
    int temp = *p1; *p1 = *p2; *p2 = temp;
}
int a = 2, b = 3; swap(&a, &b); // Verwendung
```

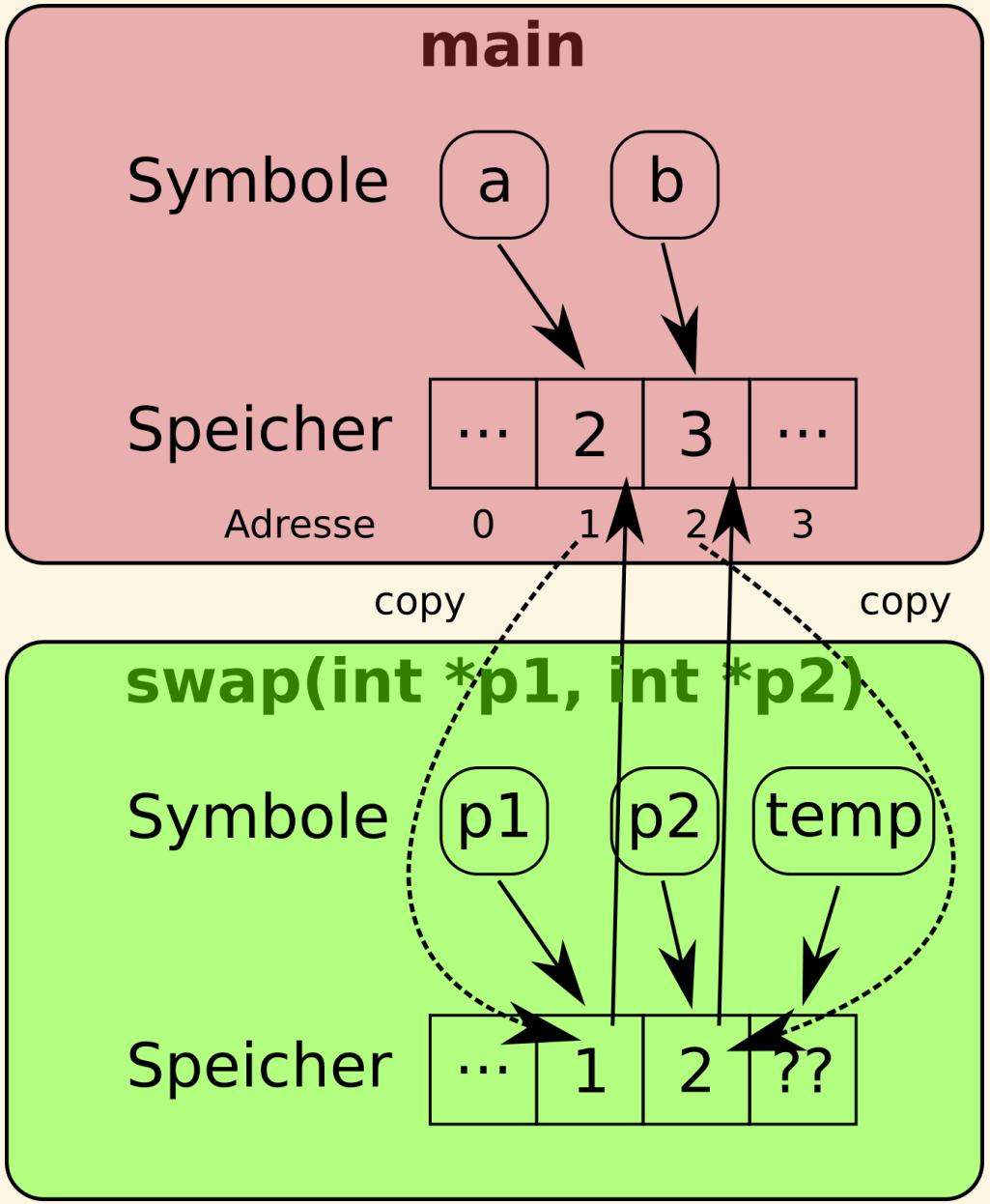
Referenzen

```
void swap(int &p1, int &p2) {
  int temp = p1; p1 = p2; p2 = temp;
}
int a = 2, b = 3; swap(a, b);  // Verwendung
```

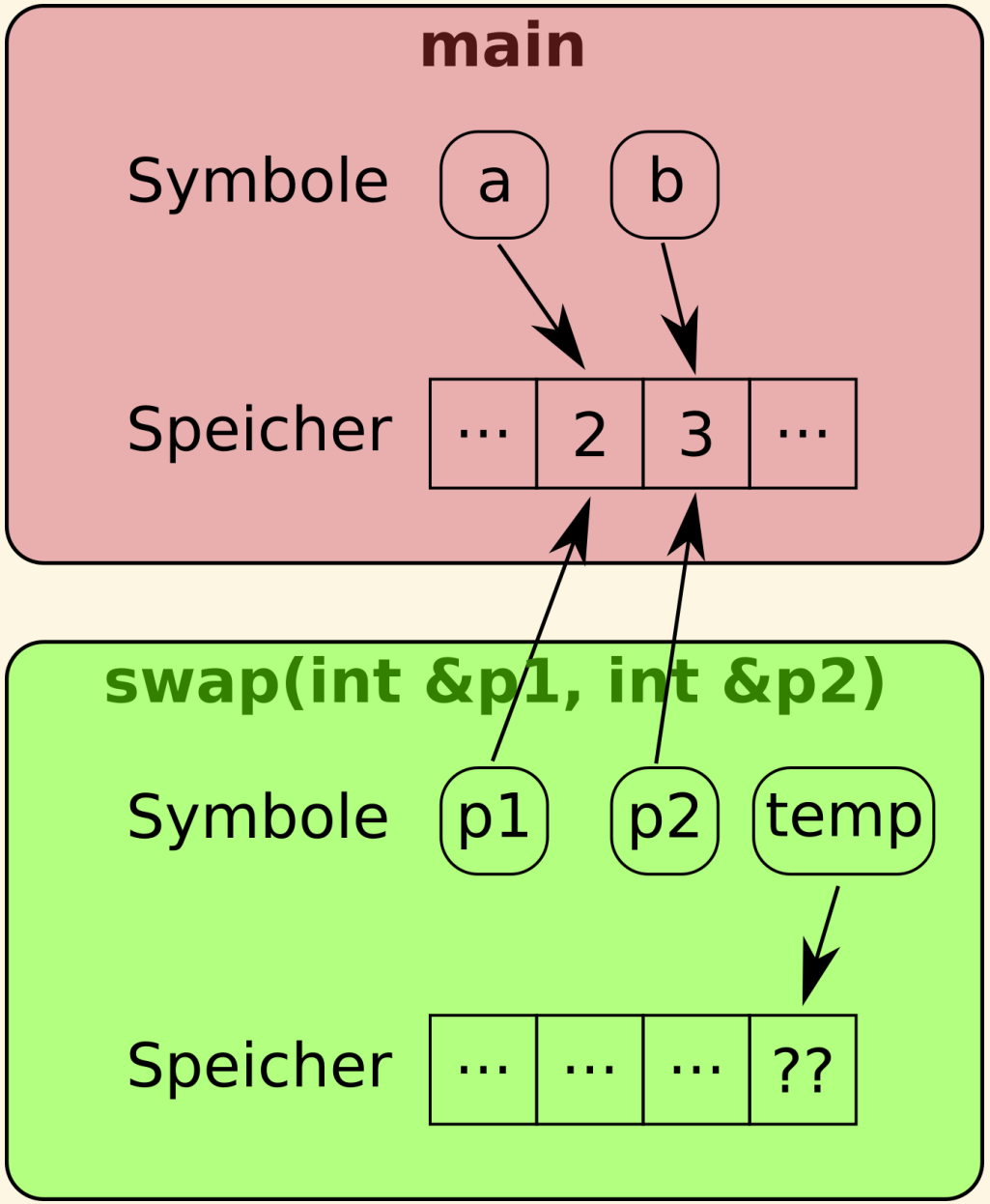
Swap – Graphische Visualisierung



Swap – Graphische Visualisierung



Swap – Graphische Visualisierung



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

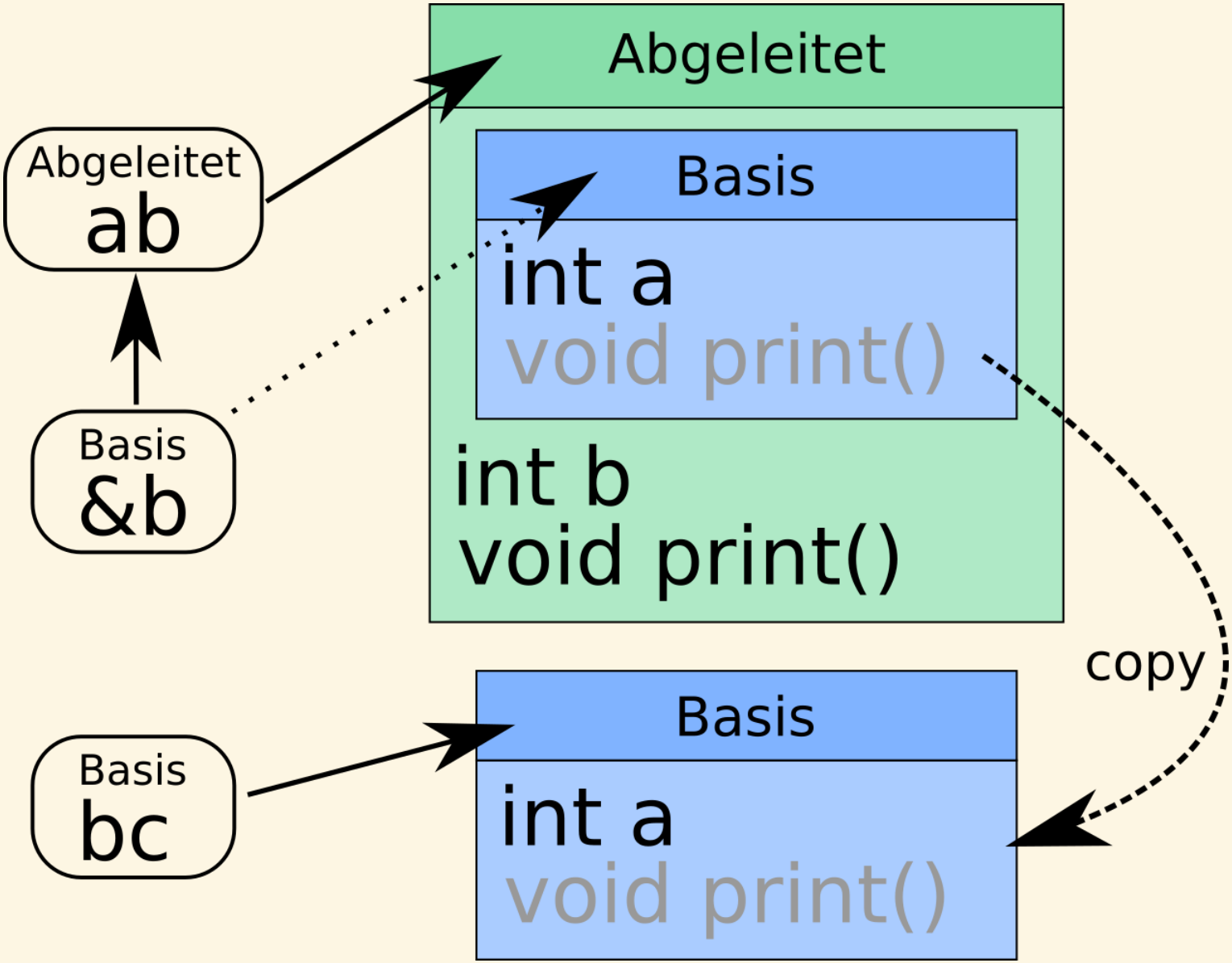
Upcast

- 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

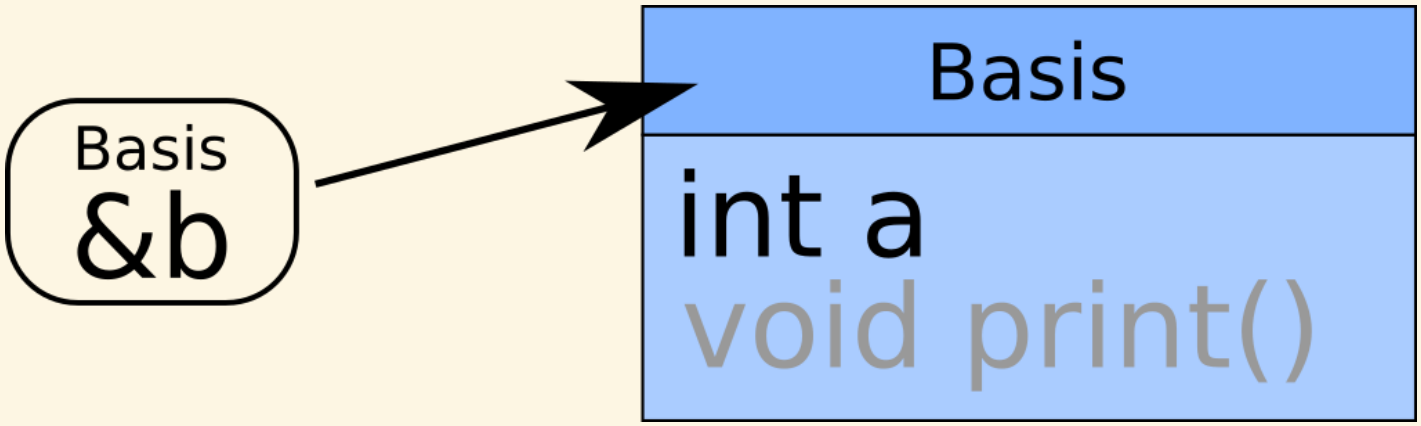
```
#include <iostream>
using namespace std;

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

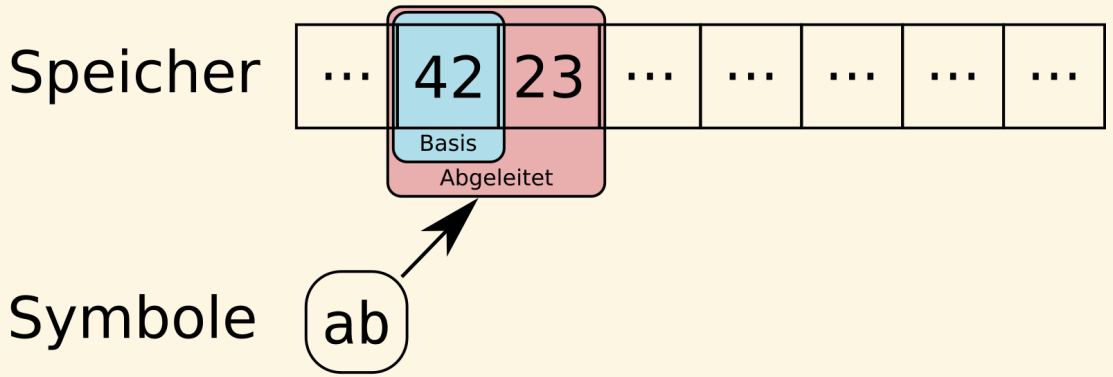
- Aus Sicht der Referenz `b` haben wir es nun mit einer Instanz der Klasse `Basis` zu tun.



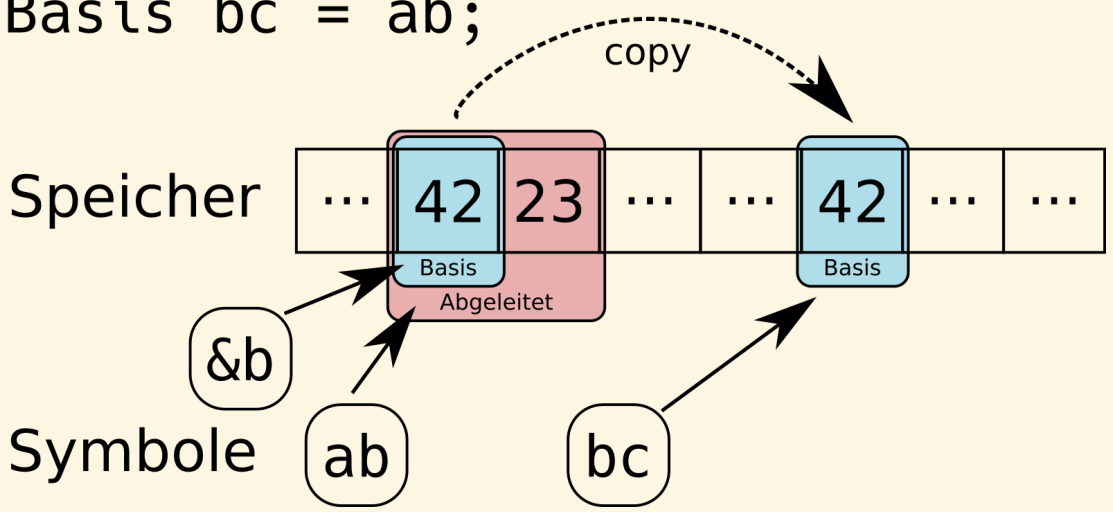
- `b` hat also "vergessen", dass es auf einen Teil einer `Abgeleitet`-Klasse verweist
- Beim kopieren für die Variable `bc` wurde nur der Basis-Teil kopiert
- Man bezeichnet das als Slicing-Problem, weil Teile einer Klasse tatsächlich, oder scheinbar abgeschnitten werden

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 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);
}
```

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: `Der Mitarbeiter 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.