Einschub: Variablen, Konstruktoren, Wertzuweisungen (C++ ↔ Java)

C++
int_stack S; // Konstruktor
Konstruiert Objekt (leeren Stack)

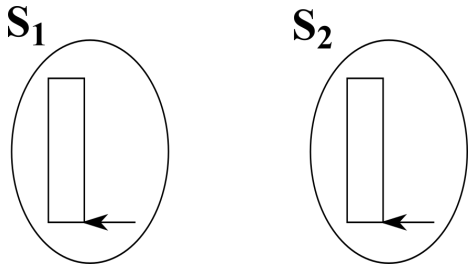
Java
S → Referenz/Pointer
↳ S = new int_stack();


```
void func(){
    int_stack S; // lokal
    S.push(17);
    return S.pop();
    // Destruktor wird automatisch aufgerufen
}
```

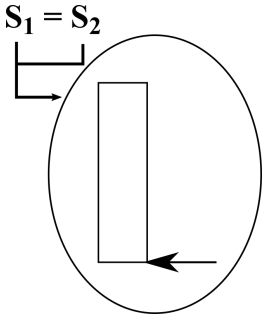
Value-Semantik (C++) ↔ Referenz-Semantik (Java)

Wertzuweisung

C++
$s_1 = s_2$;
Zwei Stacks mit gleichem Inhalt.
s_1 ist Kopie von s_2.



Java
int_stack s_1, s_2;
Die beiden Referenzen zeigen auf das gleiche Objekt.



Referenz-Semantik in C++

Um die Referenz-Semantik in C++ zu nutzen, müssen Pointer verwendet werden.

```
int_stack* sp1 = new int_stack();
int_stack sp2 = sp1;
```

Weitere Aspekte

- Test auf Gleichheit == Operator
- Parameterübergabe (und Rückgabe)

Beispiel 2.6

Parameterübergabe „by value“

```
int func(int_stack S){
   S.push(17);
   return S.top();
}
```

Java

```
int_stack S = new int_stack();
func(S);
```

S wird verändert!

C++

```
int_stack S;
func(S);
```

Die Übergabe wird nicht verändert, da sie „by value“ erfolgt d.h. der lokale Parameter S ist eine Kopie des aktuellen Parameters.

Parameterübergabe „by reference“
C++

```
int func(int_stack& S){
   S.push(17);
   return S.top();
}
```

S ist identisch mit dem aktuellen Parameter. Also wie in Java.

Pointer

```
int func(int_stack* S){
   (*S).push(17); // oder S -> push(17);
}
int_stack S;
func(&S);
int_stack * p = new int_stack();
func(p);
```

II) Einfach verkettete Liste

siehe Übung

2.5.2 Korrektheit einer Implementierung

(hier Array-Implementierung von int_stack)

Eigentlich hat man zwei Datentypen

1) Den abstrakten Datentyp int_stack (→ Datei stack.h)
2) Den konkreten Datentyp (Array-Implemtierung)

Abstrakter Zustand $s \in S$: Folge von int's
Konkreter Zustand $z \in Z$: Werte der (Member-)Variablen A, t, sz.
Durch die Invariante wird garantiert, dass nicht alle Kombinationen von A, t, sz möglich sind, sondern nur die gültigen konkreten Zustände:

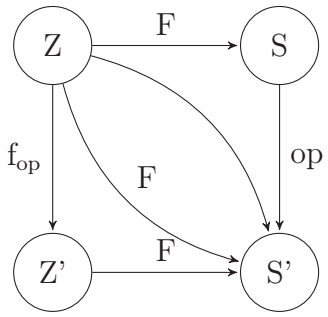
1) A ist ein Feld der Länge sz
2) $-1 \leq t \leq sz - 1$

Noch zu zeigen:

1) Konstruktoren erzeugen nur gültige konkrete Zustände
2) Für jede abstrakte Operation op (push, pop, ...) und die dazugehörige konkrete Operation f_{op} (Methode) zeige:

$$F(f_{op}(z)) = op(F(z))$$

Kommutatives Diagramm

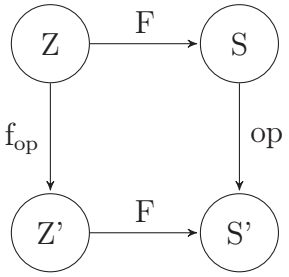


Funktion F (Semantik)

F: Z → S

hier:

$$F(A,t,sz) = \begin{cases} A[0], \ldots, A[t], & \text{für } t \geq 0 \\ \text{leer}, & \text{für } t = -1 \end{cases}$$



Beispiel 2.7 pop-Operation

$F(f_{op}(A, t, sz)) = pop(F(A, t, sz))$

Beweis (für jede Operation) → siehe Übung

3 Vererbung $\leftrightarrow$ Templates

Situation:

Es wird ein Datentyp A benötigt, der sehr ähnlich ist einem bereits definierten Datentyp B ($\rightarrow$ Klassen).

A soll

- einen Teil der Daten/Operationen wiederverwenden
- andere hinzufügen
- einige verändern (auf andere Wege implementieren)

Beispiel 3.1

Eine existierende Klasse: Polygon (B)
Eine neue Klasse: Rechteck (A)
Es handelt sich hierbei um eine Spezialisierung, da ein Rechteck ein spezielles Polygon ist.
Rechteck könnte alle Polygon-Operationen übernehmen (z.B. draw(), translate(dx,dy), rotate(x,y,ρ), area(), ...)

Einige Operationen können effizienter implementiert werden: Z.B. Flächenberechnung: area()
Andere Operationen sind nur für Rechtecke definiert: Z.B. Seitenverhältnis: ratio()

Allgemein: Klasse A (im Beispiel Rechteck) wird von Klasse B (im Beispiel Polygon) abgeleitet. B nennt man dabei die Basisklasse und A die abgeleitete Klasse.

Java

```
class A extends B{
   // Erweiterung: A-Teil
}
```

C++

```
class A: public B{
   // B-Teil:
   // Alle Daten und Methoden werden übernommen

   // A-Teil:
   // Zusätzliche Daten/Methoden,
   // modifizierte Operationen
}
```

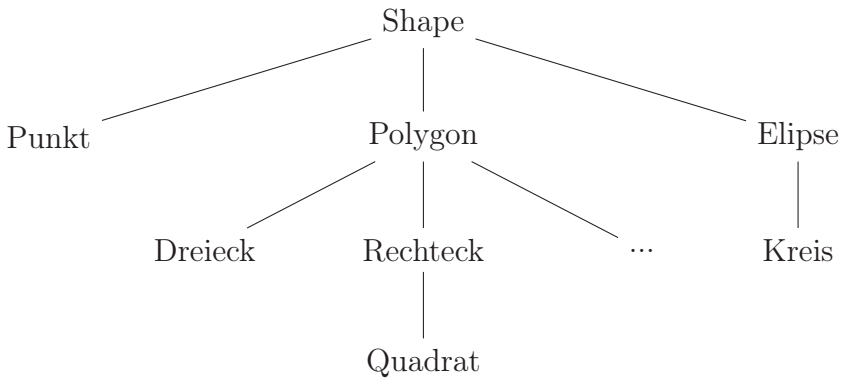
```
class Polygon{
   private // Daten

   double area();
   double umfang();
   void translate(double dx, double dy);
   void draw(...);
   ...
}
```

```
class Rechteck: public Polygon{
   // Konstruktoren -> Übung
   private:
      double b, h; // A-Teil
      // Überschreiben:
         double area(){return b*h;}
         double umfang(){return 2+(b+h);}
      // Hinzufügen:
      double ratio(){return b/h;}
}
```

Jedes Rechteck ist ein spezielles Polygon.

3.1 Allgemeiner: Ableitungsbaum



3.1.1 Typverträglichkeit

In C++ kann einer Variable von Typ B* (Pointer) oder B& (Referenz) ein Objekt (Pointer/Referenz) vom Typ A zugewiesen werden.
In Java sind alle Variablen vom Typ B.

C++

```
Polygon* p = new Rechteck(...);
void func(Polygon& poly){...}
Rechteck rect(...);
func(rect);
```

Java

```
Polygon p = new Rechteck(...);
```

Genauer: Alle im Ableitungsbaum verfügbaren Typen können zugewiesen werden.

Variable/Parameter hat zwei Typen:

- statischer Typ (zur Compile-Zeit bekannt (z.B. Polygon))
  → bestimmt, welche Methoden zur Verfügung stehen
- dynamischer Typ (zur Laufzeit bekannt (z.B. Rechteck))
  → bestimmt die Implementierung der Methoden

3.2 Polymorphe Datenstrukturen

Ein Beispiel wäre eine Feld von Polygonen.

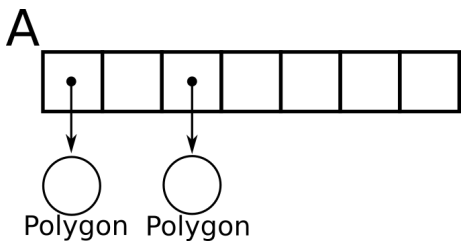
C++

```
Polygon** A = new Polygon*[100];
```

Java

```
Polygon[] A = new Polygon[100];
```

Polgon** A entspricht einem Feld von Pointern.




```
A[0] = new Polygon(...);
A[1] = new Rechteck(...);
A[2] = new Dreieck(...);
A[3] = new Quadrat(...);
```

→ polymorphes Array

Beispiel 3.2 Berechnung der Gesamtfläche

```
double total_area(Polygon** A){ // Java: (Polygon[] A)
   double total = 0;
   for (int i = 0; i < 100; i++){
      total += A[i]->area();
      // Oder: total += (*A[i]).area();
   }
   return total;
}
```

Der dynamische Typ bestimmt, welche Area-Implementierung verwendet wird.
→ Dynamische Bindung

Dynamische Bindung

```
polygon * p = new rechteck(...)
```

oder

```
double func(polygon& poly){
   return poly.area();
}
rechteck rect(...)
func(rect)
```

In C++ ist die statische Bindung der Standard. Für die dynamische Bindung wird eine virtuelle Methode benötigt.

```
virtual double area(){...}
```

`virtual` ist dabei das Schlüsselwort für die dynamische Bindung.

Beispiel 3.3 Grafik-Editor (erweiterbar)

Der Grafik-Editor speichert eine Menge von allgemeinen Flächen (→ shape)
Abstrakte Klasse „shape“

C++

```
class chape{
   virtual void draw(...) = 0
   virtual void area() = 0
   virtual void rotate(double phi, ...) = 0
}
```

Java

```
abstract class shape{
   abstract void draw(...);
   ...
}
```

Abstrakte Klassen werden verwendet, um Interfaces zu definieren.
Der Grafik-Editor speichert eine Liste von `shape*`.
`shape * p = new shape();` ergibt einen Fehler, da `shape` abstrakt ist.

Wenn `polygon` von `shape` abgeleitet ist:
C++

```
shape * p = new polygon(...)
```

```
class polygon: public shape{
   double area(){...}
}
```

Alle rein virtuellen (abstrakten) Methoden müssen definiert werden.

Java:

```
class polygon implements shape{...}
```

```
Interface shape{
   void draw();
}
```

Im Editor
`void draw_objects(){`
Iteriere über alle shapes s und rufe `s.draw()` auf.

Anwendung auf Algorithmen
Lineare Ordnung durch ein Interface (Java comparable)

C++

```
class comparable{
   virtual int compare (comparable& x) = 0;
```

Die Funktion vergleicht das Objekt, welches die Funktion aufruft, mit x und liefert

$$\begin{cases} +1, & \text{falls dieses } > x \\ 0, & \text{falls dieses } = x \\ -1, & \text{falls dieses } < x \end{cases}$$

Anwendung in generischen Sortieralgorithmen
Quicksort(comparable * A[])
zum Vergleich if(x.compare(y) ≤ 0) ...
Sortiere ein Feld von point:

```
class point : public comparable{
   point ...
   int compare(comparable & p){}
      // p ist tatsächlich ein Point!
   double px = ((point&) p).x;
   double py = ((point&) p).y;
   (x,y) < (px,py)
   // lexikographische Ordnung
   return ... ;
};
```

Aufruf von Quicksort

```
point* A [] = new point*[100];
(point**)
for( i=o; i<100; i++){
   A[i] = new point(i, i*i);
}
Quicksort(A, 100);
```

Datenstrukturen: → binäre Suchbäume, Schlüssel vom Typ „comparable“