**3) Getränkeautomat**

Automat akzeptiert 1€-Münzen

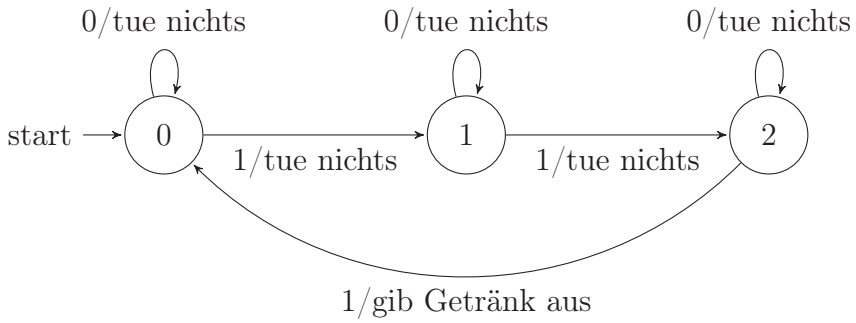
Ein Getränk kostet 3€

Operationen: Init (Reset), Akzeptiere 1€

Init $\rightarrow$ Zustand Semantik: Automat geht in Zustand 0

Akzeptiere_Euro: Zustand $\times\{0,1\} \rightarrow$ Zustand $\times$ {tue nichts, gebe Getränk aus}

Semantik: Beschreibung durch endlichen Automaten




**Bemerkungen:**

- Operationen können als partielle Funktion definiert sein. Man gibt dann den Definitionsbereich oft in einer Vorbedingung (Precondition) an.
  Beispiel: Stack und die Pop-Operation mit der Bedingung der Nicht-Leerheit des Stacks

- Operationen, bei der der Datentyp selbst auf der linken Seite nicht vorkommt, heißen Konstruktoren. Sie erzeugen eine neues Objekt (bzw. versetzen den Typ in einen bestimmten Zustand).
  Beispiele: create: $\rightarrow$ stack<T>
  create: int $\rightarrow$ Vektor (einer bestimmten Dimension)
  create int $\rightarrow$ stack <T> (Maximalgröße)

- Objekt- und Zustandssicht sind beide nützlich.
  - Stack/Getränkeautomat: Haben beide einen internen Zustand. Operationen können diesen ändern.
  - Integer: Hier ist die Objektsicht besser, da hier Operationen (z.B. ADD: integer × integer → integer) neue Objekte erzeugen und dabei existierende nicht geändert werden.
  - Syntax mit Parametern
    stack<T> ist ein sogenannter parametrisierter Datentyp, ein Stack mit Elementen vom Typ T (eventuell gibt es Anforderungen an den Typ T z.B. dictionary<T>: T muss eventuell linear geordnet sein ($x \leq y$)). Ein Stack hingegen braucht z.B. keine speziellen Anforderungen.
- Man kann nun eigentlich schon programmieren, obwohl über die Implementierung noch nichts bekannt ist.

### Anwendung von Stack<int>

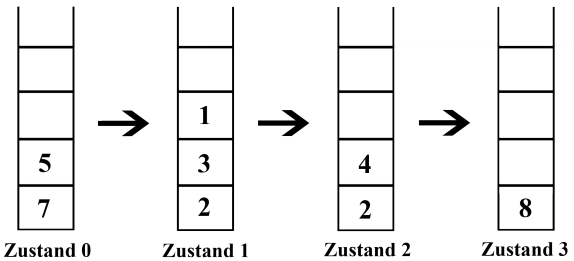
Auswertung von Postfix-Ausdrücken.
Vereinfachung:

- alle Operationen binär $+, -, *, \backslash$ (kein unäres Minus)
- Eingabe nur Zahlen 0,... ,9

### Beispiel 2.3

$(7-5)*(3+1) \rightarrow 75-31+*$ (polnische Notation) Bei Stacks ist dies gut machbar, da immer nur die zwei obersten Elemente verwendet werden.

Übung: Schreiben Sie ein Programm, das diese Ausdrücke berechnet mit dem Datentyp stack<int>.



## 2.2 Definition eines Datentyps in einer objektorientierten Programmiersprache

Im Folgenden wurde C++ als objektorientierten Programmiersprache gewählt.

**class Typename**{
// Die Definition der Menge der Objekte/Zustände (als Kommentar)

**private:** // Deklaration von Variablen zur Darstellung der Objekte/Zustände

**public:** Operation 1
...
Operation k

// Kommentar: Weitere Angaben z.B. über Effizienz

## 2.3 Operationen → Methoden/Memberfunktionen

Syntax: Ergebnistyp Name(Arg 1, Arg 2, ..., Arg k);
z.B. void push(int x);
int pop();

Spezielle Methoden (create → Konstruktoren) haben keinen Ergebnisty.
Name = Typname
z.B. stack();
stack(int size);

in C++: Destruktoren
syntax ∼Typename();
z.B. ∼stack(); → Speicherfreigabe

**Beispiel 2.4**

int_stack → stack<int>

Eine Instanz vom Typ int_stack ist eine Folge von ganzen Zahlen (int). Eine Folge der Länge 0 heißt der leere Stack.

```
class int_stack{
   private: // Implementierung
   public:
      // Konstruktor
      stack(int sz); // Erzeugt einen Stack mit
            // maximaler Anfangsgröße sz.
      // Destruktor
      ~stack()
      void push(int x); // Fügt x als letztes
         // (Top-)Element an die Folge an.
      int top() const; // liefert das letzte
         //(Top-) Element
         // Precondition: Stack ist nicht leer.
      int pop(); // entfernt das letzte Element der
         // Folge und gibt es zurück.
         // Precondition: Stack ist nicht leer.
      bool empty() const; // liefert true zurück,
         // falls der Stack leer ist, sonst false
};
```

`const` ist ein Schlüsselwort. Steht es hinter einer Funktion, so gibt es an, dass dies eine Funktion ist die den Zustand des Objekts, für das sie aufgerufen wird, nicht ändert.

## 2.4 Trennung von Spezifikation (Definition) und Implementierung

Im Gegensatz zu Java existiert in C++ eine Trennung zwischen Spezifikation und Implementierung. Hier werden die Deklarationen einer Methode (ohne Rumpf) in einer speziellen Header-Datei (z.B. int_stack.h) angegeben. Die Implementierung (dh. der C++-Code der Methoden) wird in einer anderen Datei (z.B. int_stack.cpp) angegeben. In dieser wird über #include „int_stack.h“ die Header-Datei eingebunden.
In C++ ist es, im Gegensatz zu Java, möglich mehr als eine Klasse pro Datei zu implementieren.
Diese und weitere Unterschiede zwischen Java und C++ entstehen durch die unterschiedlichen Ziele der beiden Programmiersprachen. So ist beispielsweise Effizienz ein wichtiges Ziel von C++ im Gegensatz zur Sicherheit, die ein wichtiges Ziel in Java ist.

**Beispiel 2.5**

**C++** (int_stack.h)

```
class int_stack{
   private int* A; // Feld
   int sz; // Länge von A
   int t; // Pointer (int*) auch möglich
}
```

**C++** (int_stack.cpp)

```
#include "int_stack.h"
// Konstruktor
int_stack::int_stack(int n){
   sz = n;
   A = new int[sz];
   t = -1; // Stack leer
}
```

**Java**

```
private int[] A;
int t;
```

## 2.5 Integer-Stack: dynamisch

### 2.5.1 Zwei Implementierungen

Es gibt mehrere Möglichkeiten die Klasse int_stack zu implementieren.

### I) Array-Datenstruktur

(int_stack.h)

```
class int_stack{
   public:
      // Konstruktor
      int_stack(); // Erzeugt leeren Stack
      // Destruktor
      ~int_stack();
   private:
      int* A;
      int sz; // Anfangsgröße. Kann später
            //vergrößert werden.
      int t;
      void push(int x);
}
```

**Implementierung der Operationen**

(int_stack.cpp)

```
int_stack::int_stack(){
   sz = 2; // Am Anfang Feld der Länge 2
   A = new int[sz];
   t = -1;
}
int_stack::~int_stack(){
   delete[] A; // Speicherfreigabe
}
void int_stack::push(int x){
   if(t == sz-1){ // Stack ist voll
      int* B = new int[2*sz];
      sz = 2*sz;
   }
   for(int i=0; i<=t; i++){
```

```
        B[i] = A[i]; // Werte kopieren
    }
    delete[] A;
    A = B;
    A[++t] = x; // Eigentlicher push
}
int int_stack::pop(){
    if(t == -1){
        EXCEPTION 'Pop auf leerem Stack';
    }
    if(t == sz/4){
        int* B = new int[sz/2];
        sz = sz/2;
    }
    for(int i=0; i<=t; i++){
        B[i] = A[i]; // Werte kopieren
    }
    delete[] A;
    A = B;
    }
    return A[t--];
}
```

**Bemerkungen**

- Der Destruktor wird automatisch aufgerufen, wenn eine Variable nicht mehr gültig ist.
- In Java löscht der Garbage Collector eine Variable automatisch, wenn er merkt, dass sie nicht mehr genutzt wird.
- Push-Operation: Verdopple das Feld, falls es zu klein wird.

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

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

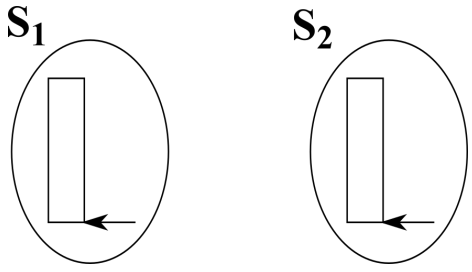

**Java**
S → Referenz/Pointer
↳ S = new int_stack();


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

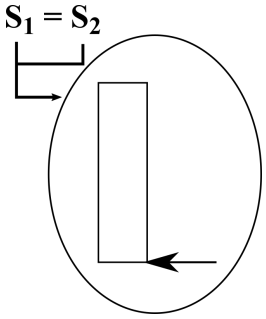
Value-Semantik (C++) ↔ Referenz-Semantik (Java)

### 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.