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)

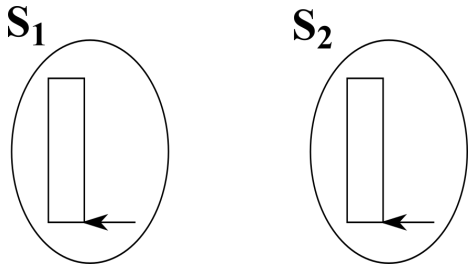
Java
S → Referenz/Pointer
↳ S = new int_stack();


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

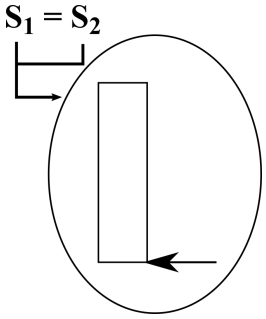
Value-Semantik (C++) ↔ Referenz-Semantik (Java)

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.

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:

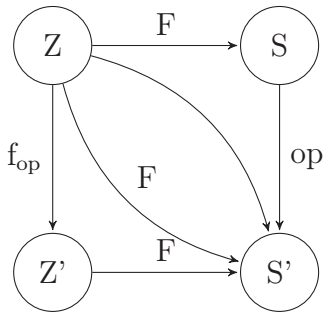
1) A ist ein Feld der Länge sz
2) $-1 \leq t \leq sz - 1$

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

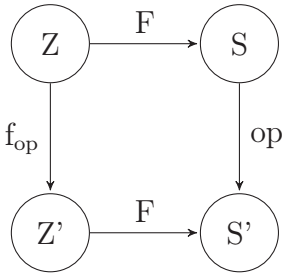


Funktion F (Semantik)

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
}
```

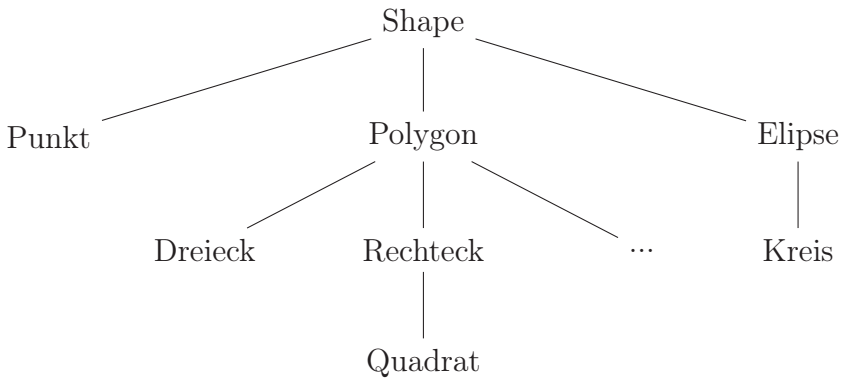
```
class Polygon{
   private // Daten

   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