Einführung in die Programmierung

Zeiger

1. Zeiger auf Variablen

Zeiger (Pointer) ist ein Typ der Variablen. Bekanntlich beinhaltet ein Datentype eine Beschreibung von dazu gehörenden Werten und erlaubten Operationen.

Werte von Zeigern sind die Adressen der Bytes im Hauptspeicher. Eine Großzahl der Operationen sind in der Sprache C/C++ definiert. In Pascal und Go dagegen gibt es relativ wenig Operationen.

Sinn der Zeiger ist u.a. die Organisation eines dynamischen Speichers, d.h. während ein Programm läuft, kann es beim Betriebssystem einen zusätzliche Speicher für Daten abfordern und bekommen. Die Daten können dann unterschiedlich aufgebaut werden: als verkettete Listen, Warteschlangen, Stacks, Bäume, um eine effektive Verarbeitung zu gewährleisten.

Wichtig sind:

- ✔ Adress- oder Referenz-Operator holt die Adresse einer "normalen" Variable;
- ✔ Inhalts- oder Dereferenz-Operator liefert den Wert von einer Adresse.

```
VAR
  i, j: Integer;
  p: ^Integer;  //  p kann nur Adresse einer ganzen Variablen annehmen
BEGIN
  p := nil;     //  p enthält eine Null (nichts, keine Adresse)
  i := 42;
  p := @i;      //  @ ist Adress- oder Referenz-Operator, in p liegt Adresse von i
  j := p^;      //  ^ ist Inhalts- oder Dereferenz-Operator, in j liegt Wert von i
  WRITELN(i, '  ',  j, '  ', p^);
END.
```

```
func main() {
  var  i, j int
  var p *int     //  p kann nur Adresse einer ganzen Variablen annehmen
  p = nil        //  p enthält eine Null (nichts, keine Adresse)
  i = 42
  p = &i         //  & ist Adress- oder Referenz-Operator, in p liegt Adresse von i
  j = *p         //  * ist Inhalts- oder Dereferenz-Operator, in j liegt Wert von i
  fmt.Print(i, "  ",  j, "  ", *p, p)
}
```

<u>Aufgabe 1.</u> Experimentieren Sie wie in oberen Beispielen mit anderen Datentypen, die in Pascal und Go definiert sind.

<u>Aufgabe 2.</u> Recherchieren Sie, welche Operationen mit Zeigern und in welchen Situationen gefährlich sein können. In welche Programmiersprachen sind solche Operationen verboten und welchen erlaubt?

2. Zeiger auf Strukturen

Praxisrelevant sind die Strukturen und Möglichkeit, dynamische Speicherstrukturen zu realisieren. Operator `new` reserviert einen Speicherbereich und liefert die Adresse dieses Bereiches.

```
TYPE Auto = RECORD
  KFZ: String[12]; { 12 ist die max. Länge der Zeichenkette, optional }
  Gewicht: Real;
  AnzPassagiere: Integer;
END;

VAR a : ^Auto;  // a kann Adresse von Auto-Struktur beinhalten

BEGIN
  New(a);  // Ein Bereich im Hauptspeicher wird angefordert, a ist dessen Adresse
  a^.KFZ := 'B XY 123456';
  a^.Gewicht := 555.77;
  a^.AnzPassagiere := 5;
  WRITELN (a^.KFZ, ' -- ', a^.Gewicht:8:2, ' -- ', a^.AnzPassagiere);
  Dispose(a);  // Bereich muss freigegeben werden
END.           // Meistens gibt es keinen Garbage Collector
```

```
type Auto struct {
  KFZ string
  Gewicht float64
  AnzPassagiere int
}

func main() {
  var a *Auto
  a = new(Auto) // Ein Bereich im Hauptspeicher wird angefordert, a ist dessen Adresse
  a.KFZ = "B XY 123456"
  a.Gewicht = 555.77
  a.AnzPassagiere = 5
  fmt.Println (a.KFZ, " -- ", a.Gewicht, " -- ", a.AnzPassagiere)
}  /* Garbage Collector gibt den Bereich frei */
```

Aufgabe 3. Schreiben Sie ein Programm in Pascal und in Go, in dem Sie linear verkette Liste erstellen. Hinweis zu dieser Liste: in der Struktur neben den Nutzfeldern muss noch ein Adressfeld hinzugefügt werden, in dem man die Adresse des nächsten Elements der Liste speichert, oder `nil`, wenn noch kein nächstes Element existiert.

Aufgabe 4. Recherchieren Sie, wie ein Stack oder eine Warteschlange aufgebaut sind und schreiben Sie entsprechende Programme in Pascal und in Go, in denen Listenelemente hinzugefügt und/oder gelöscht werden.

Aufgabe 5. Schreiben Sie Programme in Pascal und in Go, in denen ein Stack oder Warteschlange in einer Datei gespeichert und wieder daraus ausgelesen wird. Es reicht, 5 bis 6 Elemente zu haben.