Programming Concepts and Paradigms (PCP)

PCP-Übung Woche 1: Einstieg - C & Java revisited

Hauptthemen: Imperatives Programmierparadigma; strukturierte, prozedurale und objektorientierte Programmierung; Abstrakte Datentypen und Klassen, Beispielsprachen C und Java
Zeitfenster: ca. 2-3 Lektionen

Ruedi Arnold

Diese Übung repetiert und vertieft den in der ersten Woche vermittelten Stoff. Gehen Sie zur Vorbereitung nochmals durch die Unterrichtsfolien durch und schauen Sie insbesondere, dass Sie alle auf den Folien angegebenen Code-Beispiele verstanden haben. Und falls Sie keinen haben, installieren Sie sich wie in der Vorlesung erwähnt einen C-Compiler Ihrer Wahl. (Bei vielen IDEs wie VisualStudio oder Xcode ist ein C-Compiler standardmässig dabei, bzw. einfach nachrüstbar.)

1. Imperative und strukturierte Programmierung in C: Iteration über Array

Wie in der Vorlesung gesehen, unterscheiden sich strukturierte Programmierung von allgemeiner imperativer Programmierung dadurch, dass strukturierte Programme keine Sprung-Anweisungen (`goto`) verwenden. Und die meisten prozeduralen Sprachen haben Schleifen-Anweisungen (`for` oder `while`), obwohl diese grundsätzlich überflüssig sind, wenn rekursive Funktionen zur Verfügung stehen. In dieser Aufgabe schreiben Sie typischen imperativen C-Code mit einer `goto`-Anweisung, typischen strukturierten C-Code mit einer `for`-Anweisung und typischen prozeduralen Code mit einer rekursiven Funktion. Sie implementieren hier drei Funktionen, welche alle dasselbe tun: die Zahlen 0 bis n auf der Konsole ausgeben.

Hinweis: Wir gehen davon aus, dass das Argument n eine positive Zahl ist, also dass gilt n >= 0.

Die folgende Code-Sequenz:

```
printNumbersGoto(7);
printf("= printNumbersGoto(7)\n");
printNumbersFor(7);
printf("= printNumbersFor(7)\n");
printNumbersRecursiveFunction(7);
printf("= printNumbersRecursiveFunction(7)\n");
```

soll dabei beispielsweise die folgende Konsolenausgabe produzieren:

```
0 1 2 3 4 5 6 7 = printNumbersGoto(7)
0 1 2 3 4 5 6 7 = printNumbersFor(7)
0 1 2 3 4 5 6 7 = printNumbersRecursiveFunction(7)
```

a) Implementieren Sie eine entsprechende C-Funktion mit Hilfe einer goto-Anweisung (und ohne Verwendung einer Schleife oder einer rekursiven Funktion). Die Funktion soll folgendem Funktionskopf haben:

```
void printNumbersGoto(int n)
```

b) Implementieren Sie eine entsprechende C-Funktion mit Hilfe einer for-Anweisung (und ohne Verwendung einer goto-Anweisung oder einer rekursiven Funktion). Die Funktion soll folgendem Funktionskopf haben:

```
void printNumbersFor(int n)
```

c) Implementieren Sie eine entsprechende C-Funktion mit Hilfe einer rekursiven Funktion (und ohne Verwendung von Sprung- oder Schleifen-Anweisungen). Diese rekursive Funktion soll folgendem Funktionskopf haben:

```
void printNumbersRecursiveFunction(int n)
```

d) Implementieren Sie analog zu c) eine rekursiven C-Funktion Funktion (ohne Verwendung von Sprung- oder Schleifen-Anweisungen), welche anstelle der Zahlen von 0..n die Zahlen von n..0 in der Konsole ausgibt. Die Funktion soll die folgende Signatur haben:

```
void printReverseNumbersRecursiveFunction(int n)
```

Der Aufruf dieser Funktion soll beispielsweise die folgende Konsolenausgabe produzieren:

```
7 6 5 4 3 2 1 0 = printReverseNumbersRecursiveFunction(7)
```

2. Aufgabe: ADT Stack in C (Array-Implementierung)

In C werden Abstrakte Datentypen (ADT) typischerweise mithilfe von Strukturen (`struct`) und eigenen Typen (`typedef`) implementiert. Wir schauen hier im Detail den ADT Stack an. In der Vorlesung haben wir die folgenden Stack-Operationen angeschaut:

- **init:** liefert leeren Stack zurück.
- **push:** fügt ein Element zum Stack hinzu.
- **top:** liefert das zuletzt hinzugefügte Element zurück.
- **pop:** entfernt das zuletzt hinzugefügte Element.
- **print:** Gibt den aktuellen Stack aus, ohne den Stack zu verändern (Operation zu Testzwecken hinzugefügt).

a) Die in der Vorlesung verwendete C-Implementierung, basierend auf einem Array ist im GitLab (https://gitlab.enterpriselab.ch/PCP/PCP-public-Code/) abgelegt. Laden Sie sich den entsprechenden C-Code herunter und führen Sie diesen aus. Analysieren Sie die Implementierung, so dass Sie diesen Code komplett verstehen.

b) Ergänzen Sie den Code um die folgenden zwei weiteren Operationen:

- **isEmpty:** liefert 1 zurück, falls der Stack leer ist, und 0 sonst
- **size:** liefert die aktuelle Anzahl Elemente im Stack zurück

Die entsprechenden C-Funktionen sollen dabei die folgende Signatur haben:

```
int isEmpty(stack s)
int size(stack s)
```

Implementieren Sie diese beiden Funktionen. Testen Sie anschliessend ihren Code mit folgender Code-Sequenz:

```
printf("size(myStack) = %i\n", size(myStack));
printf("isEmpty(myStack) = %i\n", isEmpty(myStack));
print(myStack);
top(myStack);
myStack = push(42, myStack);
myStack = push(77, myStack);
myStack = push(1, myStack);
printf("size(myStack) = %i\n", size(myStack));
printf("isEmpty(myStack) = %i\n", isEmpty(myStack));
print(myStack);
```

Obige Code-Sequenz müsste dann in der Konsole die folgende Ausgabe produzieren:

```
size(myStack) = 0
isEmpty(myStack) = 1
print - Stack is empty
ERROR - top: stack empty!
size(myStack) = 3
isEmpty(myStack) = 0
print - Stack contains: 42, 77, 1, top element = 1
```

c) Man könnte für diesen Stack auch eine Operation **clear** definieren, welche den Stack leert. Macht es Sinn, für diese C-Implementierung von einem Stack eine **clear**-Operation zu implementieren? Begründen Sie ihre Antwort.

3. Aufgabe: ADT Stack in Java (Implementierung als verkettete Liste)

Implementieren Sie nun einen Stack in Java. Diese Implementierung soll nicht (wie die C-Implementierung oben) auf einem Array basieren, sondern auf einer verketteten Liste. Dieser Java-Stack soll dabei dieselben Operationen mit derselben Semantik anbieten, wie die C-Implementierung oben. Konkret soll ihre Klasse `Stack` also die folgende Schnittstelle haben:

```
public void push(Element e);
public Element top();
public boolean pop();
public void print();
public boolean isEmpty();
public int size();
```

a) Implementieren Sie den ADT Stack in Java unter Verwendung einer eigenen Implementierung einer verketteten Liste von Elementen (d.h. Instanzen der Klasse `Element`). Jedes Element auf dem Stack soll also eine Referenz auf seinen Nachfolger haben. Verwenden Sie dazu eine Klasse `Element` mit folgenden Methoden:

```
public Element(int value);
public int getValue();
public Element getNext();
public void setNext(Element next);
```

Testen Sie anschliessend ihre Implementierung mit folgendem Java-Code:

```
Stack myStack = new Stack();
System.out.println("myStack.size() = " + myStack.size());
System.out.println("myStack.isEmpty() = " + myStack.isEmpty());
myStack.print();
myStack.top();
myStack.push(new Element(42));
myStack.push(new Element(77));
myStack.push(new Element(1));
System.out.println("myStack.size() = " + myStack.size());
System.out.println("myStack.isEmpty() = " + myStack.isEmpty());
myStack.print();
myStack.push(new Element(33));
myStack.pop();
myStack.push(new Element(33));
myStack.print();
Element e = myStack.top();
System.out.println("top Element is " + e.getValue());
```

Obige Code-Sequenz müsste dann in der Konsole die folgende Ausgabe produzieren:

```
myStack.size() = 0
myStack.isEmpty() = true
print - Stack is empty
myStack.size() = 3
myStack.isEmpty() = false
print - Stack contains: 1, 77, 42, top Element = 1
print - Stack contains: 33, 1, 77, 42, top Element = 33
top Element is 33
```

b) Man könnte für diesen Stack auch eine Operation **clear** definieren, welche den Stack leert. Macht es Sinn, für diese Java-Implementierung von einem Stack eine **clear**-Operation zu implementieren? Begründen Sie ihre Antwort.