Programmieren 1 Übung: Pointer, Arrays und Funktionen

Klaus Kusche

1.) Bubblesort mit Pointern

Nimm deine Lösung aus der vorigen Übung (oder meine Musterlösung dafür)
und bau die *Sortier-Funktion* so um, dass sie nur *Pointer statt Array-Indices* verwendet:
Außer der Hilfsvariable zum Vertauschen zweier Elemente solltest du *keine* **`int`**-*Variablen*
mehr verwenden, nur **`int *`**.

Zur Veranschaulichung des Sortierens kannst du auch die SDL von meiner Webseite
herunterladen und laut Anleitung installieren, und dann deine Sortier-Funktion
in mein Rahmenprogramm einbauen. Damit etwas angezeigt wird, musst du in deinem
`sort` am *Ende jedes Durchlaufes* deiner äußeren Schleife meine Funktion **`paint`** aufrufen!

Zusatzaufgabe (freiwillig!):
Suche dir am Web höhere Sortierverfahren (zum Beispiel Shellsort oder Quicksort)
und implementiere sie genauso.

2.) Permutationen

Wir wollen uns heute einem uralten Problem der Kombinatorik widmen:
Dem *Erzeugen aller möglichen Permutationen* (Vertauschungen, Anordnungen) einer Menge von Objekten. In unserem Fall sind die Objekte einfach die ganzen Zahlen von 1 bis n.
Für n=3 wären alle Permutationen beispielsweise 1-2-3, 1-3-2, 2-1-3, 2-3-1, 3-1-2 und 3-2-1.
Permutationen werden in der Mathematik und Informatik (z.B. zum Testen) verwendet.

Wir wollen dazu eine Funktion **`perm`** schreiben, die in einem *übergebenen und schon befüllten Array* bei jedem Aufruf die *nächste* Permutation (nach aufsteigender Sortierung) der im Array enthaltenen Werte erzeugt.

Die Funktion soll ein Wahr/Falsch-Ergebnis zurückliefern:
"Wahr", wenn sie die nächste Permutation erfolgreich erzeugt hat, und "Falsch", wenn das Array beim Aufruf schon die letzte Permutation (verkehrt sortierte Zahlen) enthält.

Das folgende Verfahren dazu war schon im 14. Jahrhundert bekannt:

- Suche *von hinten* die erste Zahl, die *kleiner als die Zahl danach* ist.

  Wenn du *keine* findest, dann enthält das Array schon die letzte Permutation, und du kannst sofort erfolglos zurückkehren.

  Merk dir die Position dieser Zahl als *"links"*.
- Suche *von hinten* die erste Zahl, die *größer als die Zahl an der Stelle "links"* ist.

  Eine solche Zahl findest du immer.

  Merk dir die Position dieser Zahl als *"rechts"*.
- *Vertausche* die Zahlen an den Positionen "links" und "rechts".
- Drehe die *Reihenfolge* aller Zahlen zwischen der Position "links + 1" und dem Ende des Arrays um.
- Kehre erfolgreich zurück: Das Array enthält die nächste Permutation.

Da wir Funktionen und Pointer üben wollen, teilen wir das Programm gemäß der Programm-Idee in *möglichst viele kleine Funktionen* auf und implementieren jede Funktion mit *Pointern statt mit Array-Indices*:

- Eine Funktion **`swap`**: Sie bekommt *zwei Pointer* übergeben und *vertauscht* die Elemente an diesen beiden Positionen. Die Funktion hat keinen Returnwert.
- Eine Funktion **`reverse`**: Sie bekommt *zwei Pointer* übergeben und bringt die Array-Elemente zwischen diesen beiden Positionen (jeweils einschließlich!) *in umgekehrte Reihenfolge*. Auch **`reverse`** hat keinen Returnwert.

  Gehe dazu wie folgt vor:

  Solange die linke Position kleiner als die rechte Position ist,
  vertausche die Elemente an diesen beiden Positionen mittels **`swap`**
  und gehe dann mit der linken Position eins nach rechts
  und mit der rechten Position eins nach links.

- Eine Funktion **`find_smaller`**, die mit einem *Array*
  und einem *Pointer auf das letzte Element des Arrays* aufgerufen wird
  und das Array *von hinten nach vorne* durchläuft, bis sie eine Zahl findet,
  die *kleiner als die Zahl dahinter* (an der nächsten Position) ist.

  Achtung: Bei welcher Position beginnt deine Schleife,
  wenn du mit der Zahl *dahinter* vergleichen musst?

  Ist die Suche erfolgreich, wird *ein Pointer auf diese Zahl* zurückgeliefert,
  gibt es keine solche Zahl, ist das Ergebnis der **`NULL`**-Pointer.

- Eine Funktion **`find_larger`**, die mit einem *Zahlenwert*, einem *Array*
  und einem *Pointer auf das letzte Element des Arrays* aufgerufen wird
  und das Array *von hinten nach vorne* durchläuft, bis sie eine Zahl findet,
  die *größer als die angegebene Zahl* ist.
  Die Funktion liefert *einen Pointer auf die gefundene Zahl* als Ergebnis zurück.

  Obwohl die Funktion in unserem Permutationsprogramm immer so aufgerufen
  wird, dass eine solche Zahl gefunden wird, sind wir vorsichtig und geben den
  `NULL`-Pointer zurück, wenn sich bis zum Array-Anfang keine größere Zahl findet.

- Unsere *oben beschriebene Funktion* **`perm`** (*Array und Anzahl der Elemente*
  als Parameter, *Erfolg/Misserfolg* als Returnwert) lässt sich unter Verwendung
  der 4 soeben beschriebenen Funktionen ganz einfach programmieren:
  `find_smaller` liefert uns die Position "links" (bei **`NULL`** sind wir erfolglos fertig),
  `find_larger` liefert uns die Position "rechts", **`swap`** vertauscht die beiden,
  und **`reverse`** dreht die Elemente von "links + 1" bis zum letzten um.

- Für das Hauptprogramm schreiben wir noch eine Funktion **`fill`**
  (ohne Returnwert), die ein übergebenes Array von vorne mit den *Zahlen von 1 bis n*
  in aufsteigender Reihenfolge füllt (n wird der Funktion ebenfalls übergeben).
  Achtung: Die Zahlen gehen von 1 bis n, die Positionen von 0 bis n-1 !

- Weiters schreiben wir noch eine Funktion **`print`** (auch ohne Returnwert),
  die mit einem Array und der Anzahl der Elemente aufgerufen wird
  und die Elemente des Arrays in einer Zeile *ausgibt*.

Schreib dazu ein **Hauptprogramm**, das mit einer Zahl n auf der Befehlszeile
aufgerufen wird (bei fehlender Zahl oder n < 1 soll eine Fehlermeldung kommen!)
und *alle Permutationen der Zahlen von 1 bis n ausgibt*.

Mache dazu Folgendes:

- Lege ein Array der Größe n an und fülle es mittels **`fill`**
  mit den Zahlen von 1 bis n in aufsteigender Reihenfolge.
- Gib es mit **`print`** aus.
- Ruf in einer Schleife immer wieder **`perm`** auf dem Array auf:
  - Beende die Schleife und das Programm, wenn **`perm`** "Falsch" liefert.
  - Gib das Array mit **`print`** aus und mach den nächsten Schleifendurchlauf,
    wenn **`perm`** "Wahr" ergibt.

3.) Magisches Quadrat (freiwillig!)

Ein magisches Quadrat der Seitenlänge *n* ist eine *quadratische Anordnung der Zahlen* von 1 bis *n*n*, bei dem die Summe der Zahlen *in allen Zeilen und Spalten* sowie *in den beiden langen Diagonalen* **gleich** ist.

Dein Programm wird mit der Seitenlänge *n* auf der Befehlszeile aufgerufen und soll ein nach folgender Methode berechnetes magisches Quadrat (und dessen Summenwert) ausgeben:

- Man beginnt mit 1 im *rechten oberen Eck*.
- Für jede nächste Zahl geht man ein Feld nach links und 2 nach unten, *nach jeder n-ten Zahl* stattdessen waagrecht 2 nach links.
- Man betrachtet das Quadrat dabei in beiden Richtungen als zyklisch (als ob der untere und der obere bzw. der linke und der rechte Rand zylindrisch zusammengeklebt wären):
  Kommt man über den unteren Rand, macht man oben weiter,
  kommt man über den linken Rand, macht man rechts weiter.

Die Methode funktioniert nur für *ungerade und nicht durch 3 teilbare Werte* von *n*, bei anderen Eingaben soll dein Programm eine Fehlermeldung liefern.

Dein Programm soll auch prüfen, ob das Ergebnis wirklich ein magisches Quadrat ist!
Bei Seitenlänge *n* ist die richtige Zeilen- und Spaltensumme $n * (n^2 + 1) / 2$.

Zur Datendarstellung wirst du ein *zweidimensionales Array* benutzen.
Du brauchst aber in diesem Beispiel *keine Pointer* verwenden,
sondern nur normale Indices.