Programmieren C:
Pointer-Arithmetik in Arrays:
Alle Permutationen eines Arrays erzeugen, *Pointer-Version*

Klaus Kusche

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.

Wir wollen dazu eine Funktion **`perm`** schreiben, die in einem *übergebenen und schon befüllten Array* bei jedem Aufruf die *nächste* Permutation der im Array enthaltenen Werte erzeugt, und zwar *in sortierter Reihenfolge*.

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 dahinter* ist.

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

  Merk dir die Position (Index) 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 üben wollen, teilen wir das Programm gemäß der Programm-Idee in möglichst viele kleine Funktionen auf:

- Eine Funktion **`swap`**: Sie bekommt *zwei Pointer* auf Array-Elemente übergeben und *vertauscht* die Array-Elemente, auf die diese beiden Pointer zeigen.
  Die Funktion hat keinen Returnwert.
- Eine Funktion **`reverse`**: Sie bekommt *zwei Pointer* auf Array-Elemente übergeben und bringt alle Array-Elemente zwischen diesen beiden Pointern (*einschließlich* der beiden Elemente, auf die die Pointer zeigen!) *in umgekehrte Reihenfolge*.
  Auch **`reverse`** hat keinen Returnwert.

  Gehe dazu wie folgt vor:
  Solange der erste Pointer im Array *vor* den zweiten Pointer zeigt, vertausche die Elemente an diesen beiden Positionen mittels **`swap`** und schiebe dann den ersten Pointer eins nach rechts und den zweiten Pointer eins nach links (die Schleife hat also *zwei* Pointer-Schleifenvariablen, die *aufeinander zulaufen*).

- Eine Funktion **`find_smaller`**, die mit *einem Array und einem Pointer auf dessen letztes Element* aufgerufen wird und das Array *von hinten nach vorne* durchläuft (mit einer Schleife, die einen *Pointer als Schleifen-Variable* hat!), 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 die gefundene Zahl* zurückgeliefert, gibt es keine solche Zahl, soll das Ergebnis **`NULL`** sein.

- Eine Funktion **`find_larger`**, die mit *einem Array, einem Pointer auf dessen letztes Element und einem Zahlenwert* aufgerufen wird und das Array *von hinten nach vorne* durchläuft (mit einer Schleife, die einen *Pointer als Schleifen-Variable* hat!), bis sie eine Zahl findet, die *größer als die als Parameter angegebene Zahl* ist.
  Die Funktion liefert einen *Pointer auf die gefundene Zahl* als Ergebnis zurück.

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

- Unsere *oben beschriebene Funktion* **`perm`** (*Array und Anzahl der Elemente* als Parameter, *Erfolg/Misserfolg* als Returnwert) lässt sich unter Verwendung der vier soeben beschriebenen Funktionen ganz einfach programmieren:
  - Zuerst einmal berechnen wir für die folgenden Aufrufe einen *Pointer* auf das *letzte* Array-Element.
  - **`find_smaller`** liefert uns den *Pointer "links"* (bei **`NULL`** sind wir erfolglos fertig)
  - **`find_larger`** (mit dem Array-Wert an der Stelle "links" als Suchwert) liefert uns den *Pointer "rechts"*
  - **`swap`** vertauscht die beiden Elemente "links" und "rechts"
  - **`reverse`** dreht die Elemente von "links + 1" bis incl. letztem Element 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 !

  Für **`fill`** ist eine Pointer-Schleife *nicht* sinnvoll, da wir ja nicht nur die Position im Array, sondern auch den Index als Wert brauchen.

- 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* (mittels *Pointer-Schleife*).

Schreib dazu ein **Hauptprogramm**, das mit einer *Zahl n auf der Befehlszeile* aufgerufen wird (bei fehlender Zahl, n < 1 oder n > 30 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 dann mit **`print`** aus.
- Ruf in einer Schleife immer wieder **`perm`** mit dem Array auf:
  - Wenn **`perm`** "Falsch" geliefert hat: Beende die Schleife und das Programm.
  - Sonst:
    Gib das Array mit **`print`** aus und mach den nächsten Schleifendurchlauf.