# Programmieren C++: Templates, STL, File-I/O, Exceptions, ...: File mit rekursivem Merge-Sort sortieren

*Klaus Kusche*

Wir wollen ein Programm schreiben, das eine angegebene *Datei zeilenweise* komplett in den Speicher *liest*, dann mittels rekursivem Merge-Sort sortiert, und schließlich die sortierten Zeilen in eine andere angegebene *Datei speichert*.

Zum internen *Speichern der eingelesenen Zeilen* verwenden wir ein Objekt der vordefiniereten *STL-Container-Klasse* **`vector`** mit Elementtyp **`string`**.
Vektoren verhalten sich in vieler Hinsicht *wie Arrays* (man kann z.B. mit **`[ ]`** auf die einzelnen Elemente zugreifen), aber sie *wachsen automatisch* nach Bedarf, d.h. man muss *beim Anlegen keine Größe* angeben.
Das ist in diesem Beispiel besonders praktisch, da wir ja vorab nicht wissen, wie viele Zeilen die einzulesende Datei enthält.

Für das *Ergebnis des Sortierens* verwenden wir die vordefinierte *STL-Container-Klasse* **`forward_list`**, wieder mit Elementtyp **`string`** für die einzelnen Zeilen.
**`forward_list`** (gibt es erst seit C++11) ist intern als *einfach verkettete Liste* realisiert.
**`forward_list`** bietet bereits eine *Methode* **`merge`** (dass **`forward_list`** auch eine Methode **`sort`** anbietet, wollen wir zu Übungszwecken einmal *ignorieren*) die uns den *wesentlichen Teil der Arbeit* abnimmt:

- **`merge`** wird für eine **`forward_list`** aufgerufen und bekommt eine *zweite* **`forward_list`** *als Parameter*.

  *Beide* Listen müssen *schon vor dem Aufruf in sich sortiert* sein (was beim Merge-Sort automatisch der Fall ist).
- **`merge`** *entnimmt alle Elemente* aus der *Parameter-Liste* und *fügt* sie an den richtigen Stellen *in die eigene Liste* ein, sodass die eigene Liste danach die *Elemente beider Listen in sortierter Reihenfolge* (gemäß **<**-Operator) enthält.
- **`merge`** hängt dabei *nur* Listen-Elemente mittels *Pointer-Operationen* um: Es werden weder Listen-Elemente gelöscht oder neu angelegt, noch werden die Nutzdaten in den Elementen kopiert oder zugewiesen.

Mit einem **`vector`**-Ergebnis könnte man den Merge-Sort zwar *performanter realisieren*, aber der *Code* für eine schnelle Lösung wäre *deutlich aufwändiger*.

## a) Funktion `merge_sort`

Die Funktion **`merge_sort`** ist eine *Template-Funktion*, der *Typ der zu sortierenden Werte* ist Template-Parameter. Sie ist *rekursiv*, jeder Aufruf kümmert sich um *einen Teilbereich* der ursprünglichen Daten.

Sie hat *keinen Returnwert* und *vier Parameter*:

- Eine **`forward_list`** mit dem angegebenen Elementtyp für das *Ergebnis*
  (wie musst du den Parameter deklarieren,
  wenn die Funktion etwas in diesem Parameter *zurückliefern* soll?).
  **`merge_sort`** darf sich darauf verlassen, dass die Liste *beim Aufruf immer leer* ist.

- Einen **`vektor`** (ebenfalls mit diesem Elementtyp), der *die zu sortierenden Daten* enthält (wie deklarierst du ihn, damit er beim Aufruf *nicht kopiert* wird?).
- Zwei **`int`**'s: Die *Position des ersten und des letzten Elementes* jenes Bereiches im Vektor, der von diesem rekursiven Aufruf zu bearbeiten ist.

Die Funktion hat *zwei einfache* und *einen rekursiven* Fall:

- Der zu sortierende Bereich ist *leer*, d.h. das letzte liegt *vor* dem ersten Element. In diesem Fall kehrt die Funktion zurück, *ohne etwas zu tun* (*leeres* Ergebnis).
- Der zu sortierende Bereich enthält *genau einen Wert*, d.h. *erste und letzte* Position sind *gleich*: In diesem Fall hängt die Funktion *diesen Wert in die Ergebnis-Liste* (mit **`push_front`**) und kehrt zurück.
- Der zu sortierende Bereich enthält *mehrere Werte*.
  In diesem Fall brauchen wir eine *leere Hilfs-Liste* und müssen die *Position in der Mitte des zu sortierenden Bereiches* berechnen.

  Dann folgen *zwei rekursive Aufrufe*:
  - Einer sortiert die *linke Hälfte* des zu sortierenden Bereiches in die *Ergebnis-Liste*.
  - Der zweite sortiert die *rechte Hälfte* des Bereiches in unsere *Hilfs-Liste*.

  Zuletzt kommt der oben schon erwähnte **`merge`**-Aufruf, um die Elemente der Hilfs-Liste *sortiert* zur Ergebnis-Liste *hinzuzufügen*.

## b) Funktion `copy_sorted`

Das *Einlesen und Ausgeben* soll in einer Funktion **`copy_sorted`** geschehen.
Diese Funktion ist sehr ähnlich zur Übung "Zeilen einer Datei zufällig umordnen".
Sie hat *zwei* **`string`**-*Parameter* (die Dateinamen der zu lesenden und der zu schreibenden Datei) und *keinen* Returnwert.

- *Öffne* zuerst die beiden Dateien.
  Wirf einen **`string`** mit einem *Fehlertext als Exception*, wenn das Öffnen *fehlschlägt*.
- Deklariere einen **`vector`** mit **`string`**-Elementen als lokale Variable.
- Lies in einer *Schleife* immer wieder eine Zeile in eine **`string`**-Hilfsvariable (mit der Funktion **`getline`**, nicht mit der Methode **`getline`**) und hänge die gelesene Zeile mit der Methode **`push_back`** *hinten* an den Vektor an, bis das Lesen einen Fehler liefert.
- Wenn der Input *nicht leer* ist:
  - Leg eine leere **`forward_list`** an und ruf unsere Funktion **`merge_sort`** auf. Übergib ihr *diese Liste* für das Ergebnis, *unseren Vektor* als Input, und die *Position des ersten und des letzten Elementes* im Vektor.
  - *Schreib* mit einer *weiteren Schleife* alle Elemente der Liste in die Ausgabedatei. Da man auf die Elemente einer **`forward_list`** *nicht* mit **`[ ]`** zugreifen kann, brauchst du *entweder* eine **`for ( … : … )`** -Schleife *oder* eine *Iterator-Schleife*.

## c) Hauptprogramm

Das Hauptprogramm ist auch fast ident zur Übung *"Zeilen einer Datei zufällig umordnen"* (nur das **`srand`** fällt weg): Es wird mit *zwei Argumenten auf der Befehlszeile* aufgerufen: Dem *Dateinamen* der *Eingabedatei* und dem Dateinamen der *Ausgabedatei*.

- Prüfe zuerst, ob *wirklich zwei Namen* auf der Befehlszeile angegeben wurden. Wenn nicht: Fehlermeldung, Programmende.
- Ruf **`copy_sorted`** auf. Falls es eine *Fehlermeldung* wirft:
  *Fange* sie, gib sie aus, und beende das Programm.

## d) Zusatzaufgabe: Eigenes `merge`

Zuletzt wollen wir **`merge`** *selbst implementieren* (auch wenn unser **`merge`** bei weitem nicht so effizient wie das eingebaute wird, weil es Listen-Elemente anlegt und freigibt und die Werte aus der Liste frisch zuweist).

Eigentlich müssten wir dazu eine *eigene Template-Klasse* von **`forward_list`** *ableiten*, in dieser Klasse **`merge`** überschreiben, und überall unsere neue Klasse verwenden. Das erfordert aber Dinge (z.B. Ableiten und Überschreiben in Template-Klassen), mit denen wir uns nicht befasst haben. Daher gehen wir einen *einfacheren Weg*:

Wir schreiben eine *globale Template-Funktion* mit *zwei* **`forward_list`**-*Parametern* (wie übergeben?) und *ohne* Returnwert, die alle Elemente *aus der zweiten Liste entnimmt* und deren Werte *an den richtigen Stellen in die erste Liste einfügt*, und rufen diese Funktion in **`merge_sort`** statt **`merge`** auf (das sollte die *einzige Änderung* in unserem Code sein!).

Zum Verständnis der folgenden Logik dieser Funktion empfehle ich, ein paar Skizzen zu machen (Kästchen und Pfeilchen):

Die Funktion enthält *zwei Iteratoren*, die immer auf *zwei aufeinanderfolgende Elemente der ersten Liste* zeigen: Auf das Element, *hinter* dem wir *einfügen* wollen (den *"Vorgänger"*), und auf dessen *"Nachfolger"* (d.h. auf das Element *vor* dem wir *einfügen* wollen).

*Am Anfang* lassen wir den *"Vorgänger"* in das *"nichts"* ***vor*** *dem ersten Element* zeigen (suche in der Online-Doku, wie man einen solchen Iterator bekommt und wozu er dient), weil wir ja eventuell schon vor dem ersten Element einfügen müssen, und den "Nachfolger" ***auf*** *das erste Element*.

Dann kommt eine *Schleife*, die Folgendes wiederholt, *bis die zweite Liste leer ist*:

- Der *Wert des vordersten Elementes* der *zweiten* Liste wird in einer *Hilfsvariable* gespeichert, und das Element wird *aus der Liste entfernt*.
- Als nächstes müssen wir die *richtige Stelle zum Einfügen* dieses Wertes suchen: Solange ein *"Nachfolger" existiert*, und solange der *Wert* in diesem *"Nachfolger" kleiner als der einzufügende Wert* ist (oder gleich), werden *beide Iteratoren* in einer zweiten Schleife immer wieder *um eins nach hinten* verschoben.
- Dann wird der Wert der *Hilfsvariable* hinter dem "Vorgänger" in der ersten Liste *eingefügt*. Die vordefinierte Methode **`insert_after`**, die das leistet, liefert *als Returnwert* einen *Iterator* auf das neu eingefügte Element. Dieser wird unser *neuer "Vorgänger"-Iterator* für den nächsen Schleifendurchlauf (der "Nachfolger"-Iterator wird durch das Einfügen nicht verändert).