Beispiel `sorting`

Dr. Günter Kolousek

7. August 2016

1 Allgemeines

- Drucke dieses Dokument **nicht** aus!
- Halte **unbedingt** die Coding Conventions ein! Zu finden am edvossh!
- Lege für dieses Beispiel wieder ein entsprechendes Unterverzeichnis mit Beispielnummer und Beispielname an!
- Verwende den angegebenen Texteditor!

2 Aufgabenstellung

In einer Textdatei `numbers.txt` befinden sich zeilenweise ganze Zahlen, die sortiert werden sollen und in eine Datei `numbers_sorted.txt` geschrieben werden sollen.
Schreibe ein `C++` Programm `sorting`, das auf der Kommandozeile den Namen des Sortieralgorithmus – "bubble", "selection", "insertion" – bekommt und die – gemäß dem angegebenen Algorithmus – sortierten Zahlen der Textdatei `numbers.txt` in die Textdatei `numbers_sorted.txt` schreibt.
Containerdatentypen aus der Standardbibliothek wie `vector` dürfen in diesem Beispiel *nicht* verwendet werden.

3 Anleitung

1. Zuerst erstelle wie gewohnt ein CMake Projekt mit `src`, `include` und `build` Verzeichnis. Es soll eine Datei `main.cpp` geben, die bis jetzt aus einer leeren Funktion `main` besteht.
2. Ändere die Signatur der Funktion `main` folgendermaßen um:

```
int main(int argc, char* argv[])
```

Du hast schon gelernt was das bedeutet. Hier nochmals eine Kurzzusammenfassung:

- `argc` enthält die Anzahl der Kommandozeilenargumente. An erster Stelle steht jedoch immer der Programmname mit dem das Programm gestartet wurde.
- `char*` bedeutet Pointer auf ein `char`. Wie gelernt ist dies auch die Darstellung wie ein Array von `char`-Elementen verwendet wird. So ein `char` Array, das durch eine 0 (also die Zahl 0) abgeschlossen ist, nennt man einen C-String. Diese 0 als "Endezeichen" nennt man auch einen Wächter (engl. sentinel).
- `char*[]` ist demzufolge ein Array von C-Strings. Das wiederum ist nichts anderes als ein Array von Arrays.
- Du erinnerst dich sicher, dass in Python `sys.argv` dafür verwendet wurde. Aber `sys.argv` ist ja eine Liste und keine Array…

Ergänze jetzt die Funktion `main`, dass diese alle übergebenen Kommandozeilenargumente (inkl. dem Programmnamen) in einer Zeile durch je ein Leerzeichen ausgegeben wird.

3. Ändere die Funktion `main` so um, dass kein abschließendes Leerzeichen ausgegeben wird. Ok, das macht jetzt so keinen Sinn, aber du übst das letzte Element eines Arrays speziell zu behandeln.

4. Jetzt weißt du wie man auf die Kommandozeilenargumente zugreift. Ändere jetzt `main` so um, dass genau ein Kommandozeilenargument erwartet wird und dieses eines der angegegebenen Wörter darstellt. Jeder mögliche Fehler soll bewirken, dass sich das Programm mit einer sinnvollen Fehlermeldung und dem Exit-Wert 1 beendet!

   Der Exit-Wert kann in `C++` leicht mittels `return 1` in `main` an das aufrufende Programm weitergegeben werden. Das ist auch der Grund warum `main` als Rückgabetyp `int` aufweist.

   Falls es aus irgendeinem "komischen" Grund nicht funktionieren sollte, dann schaue zum nächsten Punkt.

5. Hast du vielleicht die Abfrage folgender Art von Ausdrücken programmiert:

   ```
   argv[1] != "bubble"
   ```

   Tja, so etwas kann nicht funktionieren, da `argv[1]` ein Pointer ist und `"bubble"` ebenfalls ein Pointer ist und diese beiden Pointer auf jeden Fall verschieden sind!

   Abhilfe schafft folgender Trick:

   ```
   string method{argv[1]};
   if (method != "bubble"...)
   ```

Was ist hier anders? `method` ist eine Instanz der Klasse `string` (die wir schon kennen) und wird mit dem Inhalt des C-Strings `argv[1]` initialisiert. Bei Ausdrücken auf "gleich" oder "ungleich" eines `string` mit einem C-String (ein Pointer) wird kein Pointervergleich mehr vorgenommen, sondern die Inhalte verglichen.

6. Du weißt, dass jeder Prozess standardmäßig 3 Kanäle zugeordnet hat, nämlich `stdin`, `stdout` und `stderr`. Fehlermeldungen sollen aber besser auf den Fehlerausgabekanal ausgegeben werden.

   Für `stdin` steht in `C++` `cin`, für `stdout` steht `cout` und für `stderr` steht `cerr` zur Verfügung.

   Ändere dein Programm entsprechend um.

7. Jetzt geht es an das Lesen der Eingabedatei. Dafür haben wir in einem vorherigen Beispiel schon die Funktion `read_textfile` zum Lesen einer Textdatei geschrieben. Diese Funktion hat wunderbar funktioniert.

   Lediglich zwei kleine Probleme haben wir im Moment mit der Funktion:

   - Containerdatentypen der Standardbibliothek dürfen in diesem Beispiel ja nicht verwendet werden. Damit kann auch `vector` nicht verwendet werden.

     Aus diesem Grund muss man auf ein Array zurückgreifen, um die Textzeilen der Datei jeweils als String in dem Array abzulegen.

   - Wie wir schon gelernt haben, kann ein Array allerdings nicht zur Laufzeit vergrößert werden, daher ist es notwendig die benötigte Größe im vorhinein zu kennen.

     Das wiederum bedeutet, dass wir die Datei zweimal lesen müssen: Einmal um die Anzahl der Zeilen zu eruieren und einmal um den eigentlichen Inhalt im Array abzuspeichern.

   Folgender Vorschlag: Ausgehend von der schon entwickelten Funktion `read_textfile` entwickelst du eine neue Funktion `string* read_textfile(const string filename)` (wieder in einem Modul `file_utility`), die ein Array von `string` Objekten zurückliefert. Da wir mittlerweile wissen, dass ein Array immer als ein Pointer auf das erste Element übergeben wird, kann man den Rückgabewert auch als `string*` anschreiben.

   Wie ist hier vorzugehen bzw. was ist zu beachten?

   - Zuerst einmal zeilenweise lesen und die Anzahl der Zeilen zählen.
   - Dann ein Array von lauter `string` Instanzen mit der entsprechenden Anzahl am Heap anlegen.
   - Wichtig ist dann folgendes: Am Ende des Lesens einer Datei hat das entsprechende Streamobjekt das EOF-Bit gesetzt und der Dateizeiger zum Lesen steht am Ende. Das kannst du mit folgendem Codestück ausprobieren:

```
cout << infile.eof() << endl;
cout << infile.tellg() << endl;
```

- Setze also zuerst das EOF-Bit wieder zurück. Das kann am einfachsten mittels `infile.clear();` erreicht werden. BTW, damit werden auch alle Fehlerflags zurückgesetzt.
- Dann muss der Dateizeiger zum Lesen wieder an den Anfang gesetzt werden: `infile.seekg(0);`
- Zum Lesen gehst du am Besten folgendermaßen vor:

```
string* curr{contents};  // contents is the pointer to the array
for (int i{}; i < lines; ++i) {
    string line;
    getline(infile, line);
    *curr = line;
    ++curr;
}
```

- Ein Problem gibt es allerdings noch… Wie weiß der Aufrufer, wieviele Elemente im Array enthalten sind. Der Aufrufer muss dies da er über das Array iterieren will und das Ende kennen muss.

  Es gibt zwei Möglichkeiten wie dies prinzipiell erreicht werden kann. Entweder es wird dem Aufrufer der Funktion dies auf irgendeine Art und Weise mitgeteilt (z.B. mittels eines sogenannten Referenzparameters) oder wir kennzeichnen das Ende des Arrays mittels eines besonderen Wertes.

  Wir entscheiden uns hier für die zweite Möglichkeit und kennzeichnen das Ende mit einem Leerstring. Dazu muss natürlich auch die Größe des Array um eins größer sein als die Anzahl der Zeilen. Das funktioniert natürlich nur, wenn das Endezeichen nicht in den eigentlichen Daten enthalten ist. Bei uns ist das der Fall, da in unserer Datei nur Zahlen und keine leeren Zeilen enthalten sind.

  Ich empfehle derzeit noch in `main` die gelesenen Zeilen auf `stdout` auszugeben, damit du siehst, ob alles richtig funktioniert.

- Ach ja, vergiss nicht den Speicherplatz wieder freizugen (`delete[]`).

8. Wir stehen kurz vor dem Sortieren der ganzen Zahlen. Allerdings haben wir im Moment noch gar keine Zahlen, sondern Strings in einem Array am Heap gespeichert. Was wir benötigen ist ein Array von ganzen Zahlen.

   Lege jetzt ein Array von ganzen Zahlen der entsprechenden Größe wieder am Heap an und befülle es mit den entsprechend umgewandelten Zahlen aus den Strings. Umwandeln kannst du einen String in eine Zahl mittels der Funktion `int stoi(const string&)`. Was dieses `&` bedeutet werden wir noch lernen. Aber du kannst diese Funktion auch mit einem C-String aufrufen und `C++` wird auto-

matisch aus dem C-String einen `string` erzeugen und diesen an die Funktion `stoi` übergeben.

9. Jetzt wäre ein geeigneter Zeitpunkt den Speicher für das Array von C-Strings freizugeben, da es jetzt nicht mehr benötigt wird. Und wenn Speicher, den man manuell angefordert hat, nicht mehr benötigt wird, dann sollte man diesen auch schnellstmöglich wieder freigeben.

10. Jetzt entwickelst du noch ein Modul `sorting`, das jeweils eine Funktion zum Sortieren nach dem Bubble-Sort-Algorithmus (einfachste Variante), nach dem Selection-Sort-Algorithmus und nach dem Insertion-Sort-Algorithmus sortieren kann.

    Der Prototyp jeder Sortierfunktion soll von der Struktur so aussehen:

    ```
    void sort(unsigned int n, int numbers[]);  // n is count of numbers
    ```

    Hier noch ein zwei Fragen zum Beantworten:

    - Warum wird jetzt kein Pointer übergeben, sondern ein Array?
    - Wie wird auf das sortierte Ergebnis zugegriffen?
    - Warum wird jetzt die Anzahl der Zahlen als erstes Argument übergeben?

11. Zum Schluss ist noch an der entsprechenden Stelle in `main` der dem Kommandozeilenargument entsprechende Aufruf der jeweiligen Sortierfunktion einzubauen und die Zeilen (ohne der Endemarkierung) in die Datei `numbers_sorted.txt` zu schreiben. Dafür wirst du wieder eine Funktion in `file_utitlity` benötigen.

12. Ganz zum Schluss darf nochmals höflich daran erinnern, dass noch ein wenig "Müll" händisch zu entfernen ist (bzw. dem Betriebssystem zurückgegeben werden sollte).

Fertig!

4 Übungszweck dieses Beispiels

- einfache Kommandozeilenargumente
- C-Strings
- Endemarkierung (Sentinel)
- Exit-Code von Prozessen
- `stderr` und `cerr`
- einfache Pointer und Arrays, Stack vs. Heap
- Arrays aus Textdateien lesen, in Textdateien schreiben
- Wiederholung von Bubble-Sort, Selection-Sort und Insertion-Sort
- Zurücksetzen eines Eingabestroms