Dokumentation für den PE in Programmieren 2 (C++)

Anmerkungen

- Es gibt eine interaktive Dokumentation, die im Browser angeschaut werden kann. Dazu einfach die Datei *„index.html“* öffnen. Nun sollte sich ein Browser-Fenster öffnen. Sie wurde mit dem Tool Doxygen aus den Kommentaren im Quellcode generiert und etwas angepasst.
- Parallel zu dieser Dokumentation liegt die Datei *„UML.pdf“*. Sie enthält alle vollständigen Klassendiagramme und stellt zusätzlich die Beziehung der Klassen dar.
- Das eigentliche Netbeans (8.2) Projekt ist der Ordner *„src“*.

Die Projekt-Struktur

- In den Header Files sind die Header von Klassen und Namespaces zu finden (bzw. die Implementierung in .hpp Files).
- Die Resource Files enthalten die nötigen Dateien für Tests und die Datei (*„persons.txt“*), die die serialisierten, zufällig generierten Daten enthält.
- In den Source Files ist das main-File zu finden und die zugehörigen Implementierungen zu den Data-Headern.
- In den Test Files befinden sich die Tests, die jeden testbaren Aspekt des Projekts testen.

Der Algorithmus

Ich habe mich dafür entschieden, den Sortier-Algorithmus „Quick-Sort“ selbst zu implementieren. Die Implementierung ist so generisch wie es nur geht, da der Algorithmus mit Iteratoren und Funktoren / Komparatoren arbeitet. Die einzigen geeigneten Iteratoren für den Algorithmus sind die Random-Access-Iteratoren, die einen Zugriff über den Index mit einer Zeit-Komplexität von O(1) garantieren. Dies ist auch der Grund, warum ein erster Versuch von mir, den Algorithmus auf eine Linked-List mit eigenen Iteratoren anzuwenden, fehlgeschlagen ist. Random-Access-Iteratoren für solche verketteten Listen sind nicht möglich bzw. sehr ineffizient.

Damit auch komplexe Datentypen verarbeitet werden können, muss dem Algorithmus zusätzlich ein Funktor bzw. Komparator übergeben werden. Diese verhalten sich im Prinzip wie eine Funktion, die zwei Objekte des betreffenden Datentyps als Parameter übergeben bekommt und dann einen Boolean-Wert zurückgibt.

Der Algorithmus kann also mit jedem beliebigen Datentypen der zu sortierenden Objekte und mit jedem beliebigen Container umgehen, solange dieser über **`begin()`** und **`end()`** Random-Access-Iteratoren anbietet. Damit werden die Vorgaben der STL vollständig eingehalten. Im Endeffekt kann diese eine Implementierung des Sortier-Algorithmus mit einer Vielzahl von Container-Daten-Kombinationen umgehen, z.B. **`std::vector<int>`**, **`std::vector<double>`**, **`std::deque<long>`**, oder eigene Implementierungen von Containern und Datentypen, wie **`Array<Person>`**.

Die Container

- Aus der STL
  `std::vector<T>`, **`std::deque<T>`**, **`std::array<T>`**

  Diese Container repräsentieren im Kern Sequenzen und bieten Random-Access-Iteratoren an. Das macht sie kompatibel zum Algorithmus. Für **`std::array<T>`** gilt allerdings eine Einschränkung, was meine Motivation zur eigenen Implementierung eines ähnlichen Arrays ausmacht. Das Array aus der STL kann nämlich nur mit literalen Größen initialisiert werden. Es ist also nicht möglich, zuerst die Eingangsdaten einzulesen und dann auf Basis der Größe / Menge der Daten das entsprechende Array aus der STL zu initialisieren. Deshalb muss eine literale Größe definiert werden (in *„Configuration.h“*). Ist die Größe der Eingangsdatenmenge größer als der literale Wert, dann werden nicht alle Daten im Array abgelegt. Ist die Eingangsdatenmenge kleiner als dieser literale Wert, dann werden u.U. leere Objekte im Array abgelegt und im schlimmsten Fall kann es sogar zum Absturz kommen, wenn Klassen verwendet werden würden, die keinen Default-Konstruktor haben, was hier natürlich nicht der Fall ist.

- Selbst entwickeltes **`Array<T>`**
  Das Array wurde aus dem oben genannten Grund noch einmal selbst entwickelt. Die non-literale Initialisierung der Array-Größe wird dadurch erreicht, dass der benötigte Speicher dynamisch auf dem Heap alloziert wird. Es unterstützt auch die Initialisierung durch eine „initializer-list“.

  Für das Iterieren über das Array sollten Random-Access-Iteratoren über **`begin()`** und **`end()`** angeboten werden. Dafür wurde ein eigener Iterator entwickelt, der im übernächsten Abschnitt beschrieben wird.

  Um auch die letzten Ansprüche an einen „bibliotheks-artigen“ Container zu erfüllen, werden im Fehlerfall aussagekräftige Exceptions geworfen, z.B. die **`IndexOutOfBoundsException`** wenn auf einen Index zugegriffen wird, der nicht zwischen Null und der Array-Größe liegt, oder die **`InitializeException`** wenn versucht wird, ein Array der Größe Null oder kleiner zu initialisieren.

  Das Array kann mit der Funktion **`fillWith(Container c)`** mit dem Inhalt eines beliebigen Containers gefüllt werden, natürlich solange der Container STL-konform mit einer Range-Based-For-Loop iterierbar ist und der Datentyp seiner Werte mit dem des Arrays übereinstimmt. Der subscript-Operator [] wird natürlich auch unterstützt.

- Selbst entwickelter **`CircularBuffer<T>`**
  Der CircularBuffer ist im Prinzip ein Buffer, bei dem es unmöglich ist, einen Buffer-Overflow zu erzeugen. Denn, ist der Buffer voll und ein neues Element soll hinzugefügt werden, dann wird dieses entweder abgelehnt, also nicht hinzugefügt, oder das älteste Element im Buffer wird mit dem neuen überschrieben. Dies stellt eine gute Balance zwischen Sicherheit und Speicherbedarf dar.

  In bestimmten Anwendungsfällen kann es von Nutzen sein, den Buffer sortieren zu können, z.B. wenn Daten über ein Netzwerk verschickt werden. Dort kann es passieren, dass die ursprüngliche Reihenfolge der Daten verloren geht. Wenn der Buffer nach erhalt aller Daten auf Basis von z.B. Zeitstempeln des Absendens sortiert wird, kann die ursprüngliche Reihenfolge auch auf der Client-Seite wiederhergestellt werden. Doch um diese Funktion umsetzen zu können, muss der Buffer auch Iteratoren anbieten, mit denen ein Sortier-Algorithmus arbeiten kann. Dieser selbst entwickelte Iterator wird im nächsten Abschnitt erläutert.

Die Behandlung von Ausnahmefällen ist beim CircularBuffer etwas einfacher als beim Array. Da es hier beim Einfügen keine Fehler, wie die **`IndexOutOfBoundsException`** im Array, geben kann, muss nur eine **`InitializeException`** im Konstruktor des Buffers geworfen werden, falls die geforderte Kapazität kleiner als 1 ist. Weitere Exceptions sind nicht nötig, da der CircularBuffer per Design die Randfälle behandelt, wie z.B. einen potenziellen Buffer-Overflow.

Der CircularBuffer kann, wie das Array, mit der Funktion **`fillWith(Container c)`** mit dem Inhalt eines beliebigen Containers gefüllt werden, solange der Container STL-konform mit einer Range-Based-For-Loop iterierbar ist und der Datentyp seiner Werte mit dem des Buffers übereinstimmt.

- Selbst entwickelter **`SequenceIterator<T>`**
  Um die selbst entwickelten Container STL-konform zu machen, wurde ein eigener Iterator-Typ entwickelt. Dieser ist so generisch programmiert, dass er sowohl für das selbst entwickelte Array als auch für den selbst entwickelten CircularBuffer als Iterator eingesetzt werden kann. Der **`SequenceIterator<T>`** implementiert dafür alle nötigen Funktionen, die für ein Random-Access-Iterator verpflichtend sind.

Die Eingangsdaten

- **Verwendung**
  Die Eingangsdaten sollen die Eigenschaften einer Person darstellen. Dazu gehören der Name, das Alter und das Gehalt. Der Name ist noch einmal in den Vor- und Nachnamen unterteilt. Die Attribute der Person sind *public*, da es sich hierbei nur um eine reine Datenklasse handelt, bei der die Kapselung der Attribute keinen Mehrwert bietet. Es handelt sich also um komposite Eingangsdaten mit unterschiedlichen Datentypen. Damit sieht die Klassenhierarchie für die Daten wie folgt aus:
    - Person (Klasse)
      - Name (Klasse)
        - Vorname (String)
        - Nachname (String)
      - Alter (Integer)
      - Gehalt (Double)

  Die Klasse **`Person`** implementiert die Methoden der abstrakten Klasse **`Serializable`**, um das Serialisieren einer Person zu einem String und das belegen der Eigenschaften einer Person über einen String zu ermöglichen. Die Klasse **`Name`** macht dies analog für ihre Felder. Deshalb muss **`Person`** den Namen auch nicht mehr explizit de-/serialisieren, da der Name diese Funktionalität selbst anbietet.

- **Serialisierung in ein File**
  Um letztendlich auch eine Person in ein File zu serialisieren und deserialisieren zu können, gibt es die Klasse **`Serializer<T>`**. Diese ist ebenfalls generisch und kann Objekte jeden Typs in ein File serialisieren, solange sie (**T**) die Methoden von **`Serializable`** implementieren, also zu und von einem String de-/serialisiert werden können. Der Serialisierer erhält einen **`std::vector`** von Pointern auf diese serialisierbaren Objekte und schreibt sie auf Basis ihrer **`serializeToString()`**-Methode Zeile für Zeile in eine Textdatei. Umgekehrt funktioniert das natürlich ähnlich. Es wird Zeile für Zeile der Textdatei durchgegangen. Dabei wird versucht jede Zeile in ein Objekt vom Typ **T** umzuwandeln. Dies geschieht auf Basis von **T**s **`deserializeFromString(…)`**-Methode.

Der Prozess des Serialisierens und Deserialisierens von Daten ist sehr fehleranfällig. Deshalb sind aussagekräftige Fehlermeldungen besonders wichtig. Noch vor der Ausführung des Programms könnte der Fall eintreten, dass der Programmierer einen Typen de-/serialisieren will, der gar nicht serialisierbar ist. Dann wird eine **`IncorrectTypeParameterException`** geworfen. Diese sollte allerdings zur Laufzeit gar nicht mehr auftreten können, da der Programmierer generell sicherstellen muss, ob der zu de-/serialisierende Typ auf wirklich de-/serialisierbar ist.

Beim Serialisieren kann es sein, dass z.B. die Schreib-Rechte nicht gegeben sind. Dann wird eine **`SerializationException`** geworfen. Diese Exception wird ebenfalls geworfen, wenn die geforderte Datei kaputt sein sollte. Falls die geforderte Datei nicht existiert kann sie erstellt werden. Falls die Datei zum Serialisieren also gar nicht existiert, wird deshalb kein Fehler ausgelöst, natürlich vorausgesetzt, dass die Schreib-Rechte gegeben sind.

- **Deserialisierung aus einem File**
  Beim Deserialisieren kann es sein, dass die Lese-Reche nicht gegeben sind. In diesem Fall wird eine **`DeserializationException`**. Anders als beim Serialisieren, kann hier natürlich nicht einfach eine nicht existierende Datei erzeugt werden, das wäre sinnlos. Auch hier wird eine solche Exception geworfen. Sind all diese Hürden genommen, muss beim Umwandeln der Textdaten in den internen Datentypen einiges beachtet werden. Der (serialisierbare) Typ, in den umgewandelt werden soll, muss die Funktion **`deserializeFromString(…)`** für seine Instanzen anbieten. Dann wird für jede Zeile aus der Textdatei ein Objekt vom entsprechenden Typen erstellt und auf diesem die genannte Methode mit der Zeile als Parameter ausgeführt. Dadurch werden die Attribute dieses Objekts mit den richtigen Werten, die den richtigen Datentypen haben, aus der Zeile belegt. Die erfolgreich erstellten Objekte werden in einem vector gespeichert und vom Serialisierer zurückgegeben.

  Die **`deserializeFromString(…)`**-Methode von **`Person`** und der anderen **`Serialisables`** funktionieren ähnlich. Der Eingangs-String wird an einem definierten Zeichen, z.B. ein Komma, gesplittet. Dadurch erhält man einen vector, der pro Element ein Attribut des Objekts enthält. Die Reihenfolge muss natürlich der **`serializeToString()`**-Methode des Objekts entsprechen. Falls es zu viele oder zu wenige Elemente gibt, wurden offensichtlich zu viele oder zu wenige Daten für das Deserialisieren angegeben. In beiden Fällen wird eine **`DeserializationException`** mit der Anzahl der angegebenen Daten, der Anzahl der erwarteten Daten und der Daten an sich als Beschreibung geworfen. Dann wird, im Falle von **`Person`** als zu deserialisierender Typ, versucht das dritte Element (Alter) in einen Integer umzuwandeln. Geht das schief, wird die Exception der Funktion aus der Standardbibliothek gefangen und eine sprechende **`DeserializationException`** geworfen. Für das vierte Element (Gehalt) verhält es sich analog, nur dass hier versucht wird in einen Double zu konvertieren. Ist bis hierhin kein Fehler aufgetreten, dann kann davon ausgegangen werden, dass alles korrekt ist. Dann werden die Attribute der Person auf Basis der eingelesen Werte gesetzt. Für den Vor- und Nachnamen müssen keine Umwandlungen gemacht werden, weil das sowieso Strings im **`Name`**-Objekt einer Person sind. Der Serialisier ist so generisch, dass er ohne Anpassungen auch Objekte des Typs **`Name`** de-/serialisieren, da er die Methoden von **`Serialisable`** implementiert. Dies ist auch in den Tests getestet.

  Im Serialisierer werden die **`DeserializationException`**s gefangen und die Beschreibung des Fehlers sowie der Zeile in der Textdatei ausgegeben. Fehlerhafte Zeilen werden dann einfach ignoriert, damit die korrekten Zeilen weiterhin eingelesen werden. Damit weiß der Nutzer im Fehlerfall exakt was der Fehler beim Einlesen war und an welcher Stelle er aufgetreten ist.

- **Erzeugung**
  Für die randomisierte Erzeugung der Eingangsdaten, als eine Menge von **`Person`**-Objekten, gibt es die Klasse **`DatasetGenerator`**. Sie kennt die 20 beliebtesten Vornamen in Deutschland aus 2020 und die 20 häufigsten Nachnamen in Deutschland. In der Datei „*Configuration.h*“ sind noch das jeweilige minimale und maximale Alter und Gehalt definiert. Der Generator wählt aus den Vor- und Nachnamen jeweils ein zufälliges Element aus und für das Alter und das Gehalt jeweils eine zufällige Zahl in der gegebenen Spanne. Das wird so oft gemacht, wie die Größe des Datensatzes am Ende sein soll. Die zufällig erstellten Objekte werden in einem **`Dataset`** gespeichert. Das ist im Prinzip nur ein Alias für einen vector aus Personen (**`std::vector<Person>`**).

  Der Generator kann diesen Datensatz mit Hilfe des oben beschriebenen Serialisierers in ein File schreiben und aus einem File lesen. Dabei auftretende Exceptions werden direkt gefangen und eine sprechende Ausgabe mit der Beschreibung ausgegeben.

  Dieses System macht es einfach, den Nutzer mit nur einer Entscheidung bestimmen zu lassen, ob ein neuer Datensatz generiert werden soll, und wenn ja wie groß er sein soll.

- **Limitationen**
  Der Datensatz im Eingangs-File muss einem strikten Schema folgen, um 100% valide zu sein. Jede Zeile muss eine Person widerspiegeln. Diese Zeile muss genau das richtige Format haben, also: „*Vorname,Nachname,Alter,Gehalt*“, andernfalls wird dem Nutzer gesagt, dass z.B. Zeile 5 invalide ist. Was im Vor- und Nachnamen steht, ist im Prinzip egal, aber für das Alter und das Gehalt gelten natürlich gewisse Einschränkungen: Sie müssen in einen Integer bzw. Double konvertiert werden können. Valide Einträge für das Alter sind z.B. „1“, „55“, aber auch „55P“ (wird einfach zu 55) und „45.8“ (wird einfach zu 45). Invalide sind dahingegen z.B. nur Buchstaben, wie „AB“; dann wird dem Nutzer aber genau gesagt, dass das Alter in z.B. Zeile 7 invalide ist. Für das Gehalt funktioniert das analog, nur dass natürlich „45.8“ auch zu 45.8 konvertiert wird. Weitere Limitationen für die Daten gibt es nicht. Für die randomisierte Erzeugung werden allerdings, wie oben beschrieben, Einschränkungen vorgenommen, damit der generierte Datensatz auch ansatzweise Sinn ergibt und eine Person z.B. nicht -20 Jahre alt sein kann.

Die Protokollierung der Ausgangsdaten

Bevor die Ausgangsdaten überhaupt erstellt werden, findet ein Dialog mit dem Nutzer statt. Dieser kann entscheiden, ob er einen neuen Datensatz erzeugen will oder den vorhandenen behalten will. Falls ein neuer generiert wird, wählt er die Größe zwischen 1 und 100. Anschließend wird der Nutzer gefragt, wie die Daten sortiert werden sollen, dafür gibt es 12 Möglichkeiten. Dann wird noch eine Zusammenfassung der gewünschten Einstellungen ausgegeben. Erst danach werden die verschiedenen Container mit den Daten gefüllt und der Algorithmus mit der angegebenen Sortier-Reihenfolge auf sie ausgeführt.

Die Ausgangsdaten sollten immer gleich sein (u.U. beim **`std::array`** abweichend). Denn die Rohdaten werden in jedem oben beschriebenen Container gleichzeitig verwaltet und es wird jeweils derselbe Algorithmus darauf angewendet. Die Ausgangsdaten sind dann eine strukturierte Ausgabe (auf die Konsole) der Container nachdem der Algorithmus auf sie angewendet wurde. Zur Referenz werden zuvor noch die Rohdaten strukturiert ausgegeben. Die Ausgabe erfolgt formatiert und tabellarisch, um ein schnelles Sichten und Vergleichen der Ergebnisse zu ermöglichen.