

Anmerkungen zu Aufgabenblatt 4

Der zu verwendende Programmrahmen steht als Zip-Archiv `uebung4.zip` im Moodle unter der Adresse <https://moodle.hpi3d.de/course/view.php?id=96> zum Download zur Verfügung.

Aufgabe 4.1: Game of Life (9 Punkte ³⁺⁵⁺¹)

In dieser Aufgabe soll eine einfache Variante des Game of Life¹ implementiert werden. Zugrunde liegt dieser Simulation ein zweidimensionales Array, in dem spalten- und zeilenweise Zellen verzeichnet werden. Eine Zelle des Arrays ist entweder lebendig oder tot. Beim Übergang von einem Zeitpunkt zum nächsten verändert sich die Population (d. h. der Zustand der einzelnen Zellen), wobei die Nachbarschaft der jeweils betrachteten Zelle einen wesentlichen Einfluss hat. Die Regeln sind wie folgt definiert:

- Leben entsteht in einer toten Zelle, wenn diese genau drei lebendige Nachbarzellen hat.
- Eine Zelle mit keiner oder nur einer lebendigen Nachbarzelle stirbt aufgrund von Einsamkeit.
- Eine Zelle mit vier oder mehr lebendigen Nachbarzellen stirbt aufgrund einer zu hohen Population in ihrem Umfeld.
- Eine Zelle mit zwei oder drei lebendigen Nachbarzellen überlebt, weil ihr Umfeld ausgewogen ist.

Im Programmrahmen übernimmt das Modul `game_of_life.cpp` die Auswertung der übergebenen Kommandozeilenargumente und initiiert die Simulation.

- a) Import und Export: Erweitern Sie die gekennzeichneten Konstruktoren der Klasse `Raster`, um das zweidimensionale Array zu initialisieren. Der eine Konstruktor soll das Array auf Grundlage einer angegebenen Beispielgrafik (siehe hierzu die Bitmaps im Data-Ordner des Programmrahmens) initialisieren. Die Beispielgrafik kann über eine Kommandozeilenoption spezifiziert werden (z. B. `game_of_life -p ./example.bmp`). Der andere Konstruktor soll das Array auf Grundlage einer Zufallsfunktion befüllen. Die Größe des Arrays sowie die Wahrscheinlichkeit, mit der eine Zelle als lebendig initialisiert wird, können über die Kommandozeile spezifiziert werden (z. B. `game_of_life -w 600 -h 400 -s 0.1`). Ein Wert von 0.5 führt beispielsweise dazu, dass die Hälfte der Zellen befüllt wird. Initialisieren Sie den Zufallszahlengenerator in einer geeigneten Art und Weise.

Implementieren Sie außerdem die Methode `save()` der Klasse `Raster`, um den Export der einzelnen Simulationsschritte als Bitmaps zu ermöglichen. Der finale Dateiname wird dieser Methode bereits übergeben. Das Ausgabeverzeichnis kann über die Kommandozeile spezifiziert werden (z. B. `game_of_life -p ./example.bmp -o output_directory/`).

- b) Simulation: Implementieren Sie die Funktion `simulateNextState()`, um die Population zum nächsten Zeitpunkt der Simulation zu bestimmen. Ermitteln Sie hierzu die Anzahl an lebendigen Zellen in einer 3x3-Nachbarschaft um die jeweilige Zelle. Wenden Sie anschließend die oben genannten Regeln an, um den Folgezustand der Zelle zu bestimmen. Implementieren Sie die Funktion `neighborValue()`, um den Wert einer angegebenen Zelle zu bestimmen. Fangen Sie ungültige Anfragen (z. B. Position außerhalb des Rasters) ab, indem Sie
- i) die Zelle als tot angeben oder
 - ii) das Array als Torus-förmig ansehen (d. h. den Zustand der am gegenüberliegenden Rand liegenden Zelle zurückgeben).

Für die Simulation wird standardmäßig die erste Variante gewählt. Die zweite Variante kann jedoch über eine entsprechende Kommandozeilenoption (`-t 1`) gewählt werden. Die Anzahl

¹https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

an berechneten Simulationsschritten kann ebenfalls über die Kommandozeile definiert werden (z. B. `-i 20`).

- c) Invasionssimulation: Implementieren Sie als Erweiterung die Funktion `simulateInvasion()`, die zu Beginn jedes Simulationsschritts den Zustand jeder Zelle mit einer angegebenen Wahrscheinlichkeit invertiert. Die Wahrscheinlichkeit, mit der eine Zelle invertiert wird, kann als Kommandozeilenargument spezifiziert werden (z. B. `-iv 0.005` für eine Wahrscheinlichkeit von 0.05%).

Aufgabe 4.2: Iteratoren (6 Punkte ²⁺²⁺²)

Iteratoren ermöglichen den Zugriff auf Inhalte von Datencontainern mit einer abstrakten Traversierungsfunktionalität. Dadurch können Zugriffs-, Such- und Auswertungsfunktionen implementiert werden, die vom konkret verwendeten Containertyp unabhängig sind. Implementieren Sie die folgenden Funktionen ohne den Subscript-Operator `[]`.

a) Umordnung von Containerelementen

Gegeben sei eine Folge $Z = \{c_0, c_1, \dots, c_{N-1}\}$ von Elementen beliebigen Typs. Erzeugen Sie eine neue Folge Z' , bei der die Elemente abwechselnd vom Anfang bzw. vom Ende von Z stammen, d. h. $Z' = \{c_0, c_{N-1}, c_1, c_{N-2}, \dots\}$. Entwickeln Sie die Funktion `front_back_pairing` generisch, so dass diese mindestens für die Containertypen `std::vector` und `std::list` funktioniert, sowie hinsichtlich des Datentyps unabhängig ist. Testen Sie ihre Implementierung zumindest für eine Zahlenfolge im Format `std::vector<int>` und eine Liste von Zeichenketten `std::list<std::string>`.

b) Entfernung von Duplikaten

Gegeben sei eine Folge $Z = \{c_0, c_1, \dots, c_{N-1}\}$ von Elementen beliebigen Typs, die in einem `std::vector` enthalten sind. Zu entfernen sind alle Duplikate, d. h. von mehreren Elementen mit gleichem Wert soll nur das erste Element erhalten bleiben. Beispiel: Falls $c_5 = c_7 = c_{39}$, dann werden c_7 und c_{39} aus dem Container entfernt. Entwickeln Sie eine generische Funktion `remove_duplicates`, die Duplikate entfernt. STL-Standardalgorithmen (z. B. `std::sort` und `std::unique`) dürfen verwendet werden.

c) Erweitern einer Zahlenfolge

Gegeben sei eine Zahlenfolge $Z = \{c_0, c_1, \dots, c_{N-1}\}$. Erzeugen Sie eine neue Folge Z' , bei der zu jedem Ursprungselement jeweils links und rechts ein neues Element hinzukommt; das jeweils neue linke bzw. rechte Element enthält die numerische Differenz zum Ausgangselement, d. h. das Element c_i in der Ursprungsfolge wird ersetzt durch die Teilfolge $c_{i-1} - c_i, c_i, c_{i+1} - c_i$. Die resultierende Folge Z' ist dreifach so groß wie Z . Die Folge wird zirkulär betrachtet, d. h. $c_{-1} = c_{N-1}$ und $c_N = c_0$. Implementieren Sie eine entsprechende generische Funktion `insert_differences`, die den Inhalt des übergebenen Containers C (enthält die Folge Z) durch die neue erweiterte Folge Z' ersetzt. STL-Standardalgorithmen dürfen verwendet werden.

Aufgabe 4.3: Suchen in Datensätzen (6 Punkte ²⁺⁴)

Gegeben sind Flugverbindungsdaten, die nach dem Einlesen einer Datei (`routes.csv`) im Speicher durch eine Struktur `Route` mit den Attributen `SourceID`, `DestinationID` und `AirlineID` dargestellt werden. Eine Flugverbindung wird als Eintrag in einem Vektor vom Typ `std::vector<Route>` abgelegt. Die gegebene Einleseroutine fügt die Daten direkt in den Vektor ein und gruppiert sie nach `AirlineID`.

a) Lineare Suche der Verbindungen zu einem Zielflughafen

Implementieren Sie eine Funktion, die zu einer gegebenen `DestinationID`, d. h. einem Zielflughafen, ermittelt, wie viele Verbindungen zu diesem Zielflughafen insgesamt existieren. Verwenden Sie dazu in dieser Teilaufgabe zunächst das Prinzip der linearen Suche.

Zum Test führen Sie Abfragen für alle möglichen `DestinationsIDs` (1 bis 9541) aus und messen Sie die Gesamtlaufzeit dieser Suchanfragen. Ermitteln Sie als statistische Information

die Anzahl der Zugriffe auf den Datenvektor.

b) Binäre Suche der Verbindungen zu einem Zielflughafen

Implementieren Sie die Suchfunktion zur Ermittlung aller Verbindungen zu einem Zielflughafen (analog zu a) mit Hilfe eines sortierten Vektors. Überlegen Sie, wie Sie die Elemente so sortieren können, um die Sortierreihenfolge für die Suchanfrage ausnutzen zu können. Für das Sortieren steht die Standardfunktion `std::sort` bereit.

Führen Sie wie bei Teilaufgabe a zum Test Abfragen für alle möglichen `DestinationsIDs` (1 bis 9 541) aus und messen Sie die Gesamtlaufzeit dieser Suchanfragen (ohne die Sortierung). Ermitteln Sie als statistische Information die Anzahl der Zugriffe auf den Datenvektor. Wie verändern sich Laufzeit und Containerelementzugriffszahlen?

Hinweis: Nutzen Sie zur Laufzeitmessung geeignete Funktionen aus der `std::chrono`-Bibliothek.

Aufgabe 4.4: Mergesort für Vektoren (3 Punkte)

Implementieren Sie zu einem Vektor mit beliebigem Datentyp `T` (`std::vector<T>`) den Mergesort-Algorithmus. Die entsprechende Funktion erhält den Vektor per Referenz; das sortierte Ergebnis wird darin wieder gespeichert. Für den Datentyp `T` sei die Kleiner-Relation definiert, d. h. es existiert stets `bool operator<(const T& a, const T& b)`. Die `std::inplace_merge`-Funktion darf nicht verwendet werden.

Zusatzaufgabe: Management von Restaurant-Bestellungen (3 Bonuspunkte)

In einem Fastfood-Restaurant gibt es Speisen und Getränke. Das Bedienpersonal kommt zu den Tischen und nimmt die Aufträge entgegen. Da die geselligen Gäste gerne verweilen, gibt es meist mehrfache Bestellungen pro Tisch, zum Beispiel erst Getränke, dann Speisen und später nochmals Nachbestellungen. Diese Bestellungen werden jeweils für einen Tisch mit einer eindeutigen Tischnummer gebündelt.

Erstellen Sie eine Implementierung, die Aufträge (*Orders*) entgegennimmt, in eine globale Auftragsammlung einfügt und dort Einzelaufträge pro Tisch zusammenfasst. Beispielsweise müssen, wenn die Gäste eines Tisches zahlen möchten, die Einzelaufträge zusammengefasst werden, um die Rechnungssumme zu ermitteln. Es gilt hier die Sonderaktion: „Jeder dritte Burger ist gratis! Jedes vierte Getränk (egal welches) auch!“

Ein Auftrag ist in unserem Beispielrestaurant übersichtlich definiert durch die folgende Struktur:

```
struct Order {
    int table;
    int coffee;
    int coke;
    int burger;
    int salad;
    int id;
};
```

Die bestellbaren Waren umfassen Getränke (`coffee`, `coke`) und Speisen (`burger`, `salad`). Jede Bestellung bekommt eine fortlaufende globale Bestellnummer (`id`). Jede Bestellung ist immer genau einem Tisch (`table`) zugeordnet.

Bei der Zusammenfassung der momentanen globalen Auftragsliste, sind alle Bestellungen eines Tisches zusammenzuführen. Das geschieht, indem die Anzahlen der Speisen und Getränke addiert werden. Die erste Bestellung gilt dabei als Hauptbestellung.

Beim Bezahlen muss zu einer Tischnummer ermittelt werden, was die zusammengefassten Bestellungen kosten. Dazu wird ein einfaches Kostenschema genutzt (Getränke: 2 Euro, Burger: 5 Euro, Salat: 4 Euro). Beachten Sie die Sonderaktion.

Im Programmrahmen ist eine Auftragsmenge fest kodiert, die verarbeitet werden muss. Implementieren Sie folgende Funktionen:

- a) **processOrders**
Diese Funktion überführt `incoming orders` in die globale Auftragsliste (`current orders`). Überlegen Sie sich, wie die Einzelaufträge sinnvoll in die globale Auftragsliste eingeführt werden sollten.
- b) **mergeOrders**
Diese Funktion „komprimiert“ die vorliegenden Bestellungen in der globalen Auftragsliste (`current orders`). Alle Aufträge für einen Tisch werden zu einem einzigen Auftrag zusammengefasst.
- c) **pay**
Diese Funktion berechnet den Preis der gesamten Bestellungen pro Tisch auf Basis der oben genannten einfachen Preistabelle. Der Auftrag wird hierbei aus der globalen Auftragsstabelle gelöscht. Hat ein Tisch keine Bestellung, dann wird ein Betrag von 0 Euro zurückgegeben.

Für die Implementierung sind allgemein Iteratoren und – unter anderem – auch die Funktionen `std::sort`, `std::remove_if`, `std::find_if`, `std::vector<Order>::erase` hilfreich. Dem Programmrahmen sind Testdaten für Aufträge beigelegt (`orders.txt`), gegen die automatisiert getestet wird.

Allgemeine Hinweise zur Bearbeitung und Abgabe

- Die Aufgaben sollen maximal zu zweit bearbeitet werden; Ausnahmen sind nicht vorgesehen. Je Gruppe ist nur eine Abgabe notwendig.
- Bonuspunkte können innerhalb des Übungsblattes verlorene Punkte ausgleichen. Sie sind nicht auf andere Übungsblätter übertragbar und werden nicht über die reguläre Punktzahl hinaus angerechnet.
- Bitte reichen Sie Ihre Lösungen bis spätestens **Montag, den 19. Juni um 17:00 Uhr** ein.
- Die Implementierung kann auf einer üblichen Plattform (Windows, Linux, OS X) erfolgen, darf aber keine plattformspezifischen Elemente enthalten, d. h. die Implementierung soll plattformunabhängig entwickelt werden. Lesen Sie bei Fragen zum Kompilieren und Ausführen bitte genau die dem Programmrahmen beiliegende `README.TXT`.

Bestehen weitere Fragen und Probleme, kontaktieren Sie den Übungsleiter oder nutzen Sie das Forum im Moodle.

- Archivieren Sie zur Abgabe Ihren bearbeiteten Programmrahmen als Zip-Archiv und ergänzen Sie Ihre Namen im Bezeichner des Zip-Archivs entsprechend im folgenden Format: **uebung4_vorname1_nachname1_vorname2_nachname2.zip**. Beachten Sie, dass dabei nur die vollständigen Quelltexte, die CMake-Konfiguration sowie eventuelle Zusatzdaten gepackt werden (alle Dateien, die im gegebenen Programmrahmen vorhanden waren). Laden Sie keine Kompilate und temporären Dateien (`*.obj`, `*.pdb`, `*.ilk`, `*.ncb` etc.) hoch. Testen Sie vor dem Hochladen, ob die Abgabe fehlerfrei kompiliert und ausgeführt werden kann.
- Reichen Sie Ihr Zip-Archiv im Moodle ein:
<https://moodle.hpi3d.de/course/view.php?id=96>.