OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

ELEKTRO- UND
INFORMATIONSTECHNIK

Praktikum Informatik 1

Big Brother

Autor(en): M. Niemetz, L. Osinski, T. Langer, M. Schumm

Oktober 2024

Version: 1.31

letzte Aktualisierung: 25. Oktober 2024

Ostbayerische Technische Hochschule Regensburg
Fakultät für Elektro- und Informationstechnik

1 Einleitung und Vorbereitung

In dieser Aufgabe verwenden Sie die dynamische Speicherverwaltung der Sprache C und gewinnen einen ersten Einblick in die Effizienz von unterschiedlichen Programmieransätzen.

Das zu erstellende Programm soll Ihnen mitteilen, wie häufig Ihr Vorname im aktuellen Semester vertreten ist. Um dies zu bewerkstelligen, muss das Programm die Vornamen des aktuellen Semesters aus einer Textdatei einlesen, diese der Häufigkeit nach sortieren und abschließend die Häufigkeit eines eingelesenen Vornamens ausgeben.

> **Wichtig:** Bevor Sie mit dem Programmieren beginnen, sollten Sie sich Zeit nehmen und mindestens die **gesamte Teilaufgabe komplett lesen** und die **Struktur Ihres Programms auf Papier/Zeichentablett planen**!
>
> Dabei sollten Sie Dinge überlegen wie:
>
> - Welche Variablen benötigen Sie?
> - Welche Schritte müssen durchgeführt werden?
> - Welche Funktionen benötigen Sie und welche Parameter müssen diese besitzen?
>
> Sie sparen sich dabei **enorm viel Zeit**, die Sie sonst mit Herumbasteln und **äußerst aufwendigen** Fehlersuchen verschwenden.

Folgende Übersicht soll Ihnen bei der Vor- bzw. Nachbereitung der in dieser Aufgabe benötigten inhalte helfen:

Vorbereitung

2 Die schrittweise Umsetzung

Um möglichst effizient voranzukommen, empfiehlt sich ein strukturiertes und schrittweises Vorgehen.

2.1 Definition einer Datenstruktur

☕
Definieren Sie einen Struktur-Datentyp zur Speicherung der Häufigkeitswerte einzelner Namen.

Recherche ☕
Definition und Verwendung von Strukturen

Deklarieren Sie zunächst eine Struktur `struct NameCounter` zur Speicherung

- der Vornamen der Studierenden als Array von Buchstaben als erstem Element der Struktur und
- deren absoluten Auftrittshäufigkeiten als Integer-Wert als zweitem Element der Struktur.

Wählen Sie dabei für die beiden Strukturelemente geeignete und gut verständliche Bezeichnungen! Sie können bei der Dimensionierung des Arrays in der Struktur davon ausgehen, dass **kein Name länger als 1500 Zeichen** sein wird[1].

Gehen Sie weiterhin davon aus, dass **maximal 500 verschiedene Namen** eingelesen werden sollen. Legen Sie nun ein entsprechendes Array an, welches Ihren neu definierten Strukturdatentyp als Basistyp verwendet.

> **Wichtig:** Sollten Sie bei Ihrem Compiler bzw. Betriebssystem mit 1500 Zeichen pro Name an Grenzen stoßen, verwenden Sie bitte 500 als maximale Länge der Namen.

[1] Diese hohe Zahl an Buchstaben verwenden wir **nicht** für den unrealistischen Fall, dass die Namen so lang sind und erst recht nicht um zu vermeiden dass es zu einem Speicherüberlauf beim Einlesen eines Namens kommt. Dieses Problem müsste man „richtig“ lösen und nicht durch Speicherverschwendung nur „höchstwahrscheinlich“. Uns geht es hier darum, bei der Optimierung am Ende auch einen deutlichen Effekt zu sehen und das Beispiel dennoch einfach zu halten. Daher wird der Datensatz bewusst groß gewählt.

Frage: Verdeutlichen Sie sich die von Ihnen erzeugte Datenstruktur durch eine Skizze. Überlegen Sie, wie Sie diese anderen Personen erklären könnten.

2.2 Einlesen der Namen aus der Datei

Schreiben Sie eine Funktion zum Einlesen der Namen aus der Datei.

Schreiben Sie nun unter Verwendung dieses neu definierten Strukturdatentyps `struct NameCounter` eine Funktion `readNameDB`, welche die Folge von Vornamen aus einer Datei einliest, deren Dateiname ihr als Parameter übergeben wird.

Wichtig: Falls Sie `fgets` statt `fscanf` zum Einlesen verwenden wollen, müssen Sie unbedingt beachten, dass `fgets` das Zeilenende-Zeichen am Ende mit in die Zeichenkette einliest. Dieses müssen Sie (z.B. durch Überschreiben durch ein Nullbyte) entfernen, da es bei der späteren Suche nach Namen stört, indem es zwar bei der Ausgabe "fast unsichtbarïst, aber dennoch bei Vergleichen als Zeichen beachtet wird. Unerwartete Zeilenende-Zeichen sind eine beliebte Ursache für stundenlange Debugging-Sitzungen!

Die eingelesenen Daten sollen dabei in dem in Abschnitt 2.1 definierten Array des Basistyps `struct NameCounter` abgelegt werden. Das Array selbst wie auch die Maximalzahl der darin vorhandenen Einträge werden dabei ebenfalls als Parameter an die Funktion übergeben.

Die Funktion hat damit also drei Parameter:

- Name der Datei
- Array für die eingelesenen Daten und
- (Maximal-)Länge des Arrays.

Beim Einlesevorgang sollen mehrfach auftretende identische Namen nur einmal abgespeichert werden. Allerdings sollen die Zähler in den `struct NameCounter`-Elementen so aktualisiert werden, dass die Häufigkeit des Auftretens der Namen korrekt darin gespeichert ist.

Als Rückgabewert soll die Funktion die Anzahl der belegten Einträge im Array zurückliefern.

Testen Sie Ihre Funktion mit der vorgegebenen Datei.

Lesen Sie die Daten aus der vorgegebenen Datei `vornamen.txt` ein und prüfen Sie das Ergebnis stichprobenweise, indem Sie die eingelesenen Daten im Array mit dem Debugger untersuchen.

Wichtig: Analysieren und beheben Sie eventuell auftretende Fehler in Ihrem Programm jetzt!

2.3 Einträge suchen

Erstellen Sie eine Funktion zum Suchen eines Namenseintrags.

Erstellen Sie nun eine Funktion zur Suche eines konkreten Vornamens in einem Array von Namenseinträgen, wie es durch die Einlesefunktion der vorangehenden Teilaufgabe erzeugt wird. Die Funktion soll folgende Signatur besitzen

```
int find_Name(char* Name,
              struct NameCounter Database[],
              int DBSize);
```

und den Namen `Name` in dem durch den Pointer `Database` spezifizierten Array suchen. `DBSize` gibt dabei an, wie viele gültige Namenseinträge im Array enthalten sind.

Wird der Name gefunden, soll die Funktion den **Index der Fundstelle** im Array als Rückgabewert zurückgeben. Scheitert die Suche, soll ein **negativer Wert als Fehlermeldung** zurückgegeben werden.

Testen Sie Ihre Suchfunktion.

> **Wichtig:** Testen Sie die Funktion durch ein kleines Beispielprogramm, bevor Sie weiterarbeiten und beheben Sie eventuell auftretende Fehler.

2.4 Häufigkeit konkreter Namen ermitteln

Schreiben Sie ein Programm, das die Häufigkeit konkreter Namen ausgibt.

Nun sind alle Bausteine vorhanden, um das Programm zu testen und die Häufigkeit eines bestimmten Namens zu ermitteln:

Ändern Sie Ihr Hauptprogramm so ab, dass es eine Datei mit dem Namen `vornamen.txt` einliest, dann einen Namen vom Benutzer abfragt, diesen im Namensarray sucht und die entsprechende Häufigkeit auf dem Bildschirm ausgibt.

Recherche 

Funktionsweise von Zeichenkettenvergleichen mittels `strcmp`

> **Hinweis:** Für den Vergleich der Namen benötigen Sie die Funktion `strcmp`.

Recherche 

Einlesen von Zeichenketten mittels `scanf`

> **Hinweis:** Schlagen Sie ggf. nach, wie Sie Zeichenketten mit `scanf` interaktiv von der Tastatur einlesen können.

2.5 Sortieren nach Häufigkeit

Erstellen Sie eine Funktion zum Sortieren der Namensdatenbank nach der Häufigkeit.

Nun soll das Programm so erweitert werden, dass zusätzlich noch ein Ranking der Namenshäufigkeiten erstellt wird, also die Namen im Namensarray nach **absteigender Häufigkeit** sortiert werden.

Zum Sortieren verwenden Sie den sogenannten **Bubblesort**-Algorithmus. Dieser hat zwar ein ineffizientes Laufzeitverhalten für

große Datenmengen, ist aber einfach zu implementieren und zu verstehen:

Kurz gesagt besteht die Idee bei Bubblesort darin, alle Elemente zu durchlaufen und dabei jedes Element mit seinem Nachfolgeelement zu vertauschen, wenn die beiden in der falschen Reihenfolge stehen. Da nach einem Durchlauf typischerweise die Sortierung nicht vollständig ist, muss dieser Vorgang so oft wiederholt werden, bis nichts mehr zu vertauschen war, also alle Elemente in der korrekten Reihenfolge lagen.

Als Pseudocode sieht der Bubblesort-Algorithmus zum Beispiel wie in Abbildung 1 dargestellt aus. Dabei wird ein „Flag“ F verwendet um zu ermitteln, ob im letzten Durchgang noch mindestens eine Vertauschungsoperation durchgeführt wurde – also ob noch ein weiterer Durchgang erforderlich ist.

input : Unsortiertes Array A der Länge n
output: Sortiertes Array A der Länge n
$F \leftarrow$ true ;
while F **do**
  $F \leftarrow$ false ;
  for $i = 0$ **to** $n-2$ **do**
    if $A[i] > A[i+1]$ **then**
      tausche $A[i]$ mit $A[i+1]$;
      $F \leftarrow$ true ;
    end
  end
end

Abbildung 1: Bubblesort-Algorithmus, siehe auch [1]

Implementieren Sie den Bubblesort-Algorithmus in der Funktion

```
void sortNameDB(struct NameCounter NameDB[],
                int DBSize);
```

welche die belegten Namenseinträge im übergebenen Array sortiert.

Recherche
Ringtausch zum Vertauschen der Werte zweier Variablen

Beachten Sie, dass Sie zum Vertauschen der Inhalte zweier Variablen zwingend eine Variable zum temporären Zwischenspeichern eines Werts benötigen.

Hinweis: Werte eines Strukturtyps können ebenso wie andere Variablen direkt per Zuweisungsoperator verarbeitet werden. Sie müssen entgegen anderslautenden Internet-Mythen **nicht** die Einzelelemente getrennt kopieren. Das ist schneller, einfacher und besser lesbar!

Erweitern Sie Ihr Hauptprogramm um die Ausgabe der Rankingposition.

Testen Sie Ihre Implementierung, indem Sie das oben erstellte Programm so erweitern, dass neben der absoluten Häufigkeit auch die Position im Ranking mit ausgegeben wird.

Die Ausgabe Ihres Programms soll wie folgt auf dem Bildschirm dargestellt werden.

```
Geben Sie Ihren Namen ein: Max
-------------------------------------------
Max steht an 3. Stelle und ist 4x vorhanden.
```

2.6 Vorbereitung und Durchführung der Laufzeitmessung

Ihr aktuelles Programm ist in zweierlei Hinsicht ineffizient:

- Jeder Name belegt 1500 Byte Speicherplatz, obwohl Namen typischerweise nicht diese Länge erreichen. Dies gilt auch für ungenutzte Einträge in Ihrem Array.
- Aktuell muss für jede Tauschoperation beim Sortieren der **komplette Inhalt** der Arrayelemente kopiert werden:
  Insgesamt müssen für das Zuweisen eines einzelnen Eintrags also die 1500 Bytes des Namens sowie die (typischerweise) vier Bytes der Häufigkeit kopiert werden. Bei jeder Tauschoperation finden drei Zuweisungen statt. Das heißt für den Tausch zweier Elemente müssen jeweils *mindestens* $(1500\,\mathrm{Byte} + 4\,\mathrm{Byte}) \cdot 3 = 4512\,\mathrm{Byte}$ kopiert werden.
  Das mag für wenige Namenseinträge noch tolerierbar sein, mit einer steigenden Zahl an Einträgen oder bei häufigem Sortieren wird diese Ineffizienz schnell zum Problem.

Um den Erfolg unserer geplanten Optimierung des Programms später auch beurteilen zu können, führen wir nun zunächst eine Laufzeitmessung für die bisherige Implementierung durch.

Binden Sie die Bibliothek zur Laufzeitmessung in Ihr Projekt ein.

Hierfür laden Sie die im Online-Kurs verfügbare kleine Bibliothek zur Laufzeitmessung herunter, entpacken die Dateien aus dem Archiv und binden sie in Ihr Projekt ein.

Mit dieser Bibliothek erhalten Sie unter anderem folgende Datentypen und Funktionen zur Durchführung von Laufzeitmessungen:

t_stopwatch Jede Variable dieses **Typs** stellt eine unabhängige Stoppuhr dar. Die Bedienung der Stoppuhren geschieht mit den jeweiligen Funktionen.

Sie können also zum Beispiel mit üblichen Variablendefinitionen wie `t_stopwatch MyFancyStopwatch`; Stoppuhr-„Variablen“ erzeugen, die unabhängig voneinander die Zeit messen.

t_timevalue Dieser **Typ** wird verwendet, um das Ergebnis der Zeitmessung in Nanosekunden anzugeben.

startStopwatch Diese **Funktion** initialisiert die als Parameter übergebene Stoppuhr und startet sie. Diese Funktion muss für jede Stoppuhr als erste Funktion aufgerufen werden, bevor mit der Stoppuhr gearbeitet werden kann, sonst funktioniert die Stoppuhr nicht korrekt.

stopStopwatch Diese **Funktion** stoppt die als Parameter übergebene laufende Stoppuhr und liefert den aktuellen Zeitwert der Stoppuhr als Rückgabewert des Typs `t_timevalue`.

Messen Sie die Laufzeit Ihrer Sortierfunktion

Definieren Sie in Ihrem Programm eine Stoppuhr als Variable des Typs `t_stopwatch` und fügen Sie direkt vor dem Aufruf Ihrer Sortierfunktion den Start der Stoppuhr mit `startStopwatch`, und direkt nach dem Aufruf einen Stopp mit `stopStopwatch` ein.

Wichtig: Achten Sie darauf, zwischen diesen drei Aufrufen keine weiteren Anweisungen auszuführen.

Wichtig: Achten Sie darauf, in Ihrer Sortierfunktion keine Bildschirmausgaben durchzuführen. Diese kosten sehr viel Laufzeit und würden das Ergebnis verfälschen!

Geben Sie das Ergebnis der Messung auf dem Bildschirm aus.

Hinweis: Der Typ `t_timevalue` ist intern durch den Typ `long long unsigned` realisiert. Der standardkonforme Formatbezeichner für `printf` für diesen Typ ist `%llu`. Auf manchen Windows-Systemen wird dieser Bezeichner aber nicht korrekt unterstützt. Verwenden Sie in diesem Fall ersatzweise `%I64d`.

Starten Sie Ihr Programm **mehrmals** (z.B. fünf Mal) und notieren Sie sich alle Ergebnisse der Zeitmessung.

Wichtig: Stellen Sie sicher, dass Sie diese Messwerte im Abgabegespräch zur Verfügung haben!

Frage: Was fällt Ihnen bei der wiederholten Durchführung der Messungen auf? Wie erklären Sie sich das Ergebnis?

2.7 Optimierung der Datenstruktur

Im Folgenden soll das Programm so optimiert werden, dass die Zahl der Daten, die für einen Tausch kopiert werden müssen, möglichst gering ist. Außerdem soll der Speicherbedarf des Programms an die tatsächlich Bedürfnisse angepasst werden.

> **Frage:** Ermitteln Sie mit `sizeof` die Größe einer Variable des Typs `struct NameCounter` im Speicher und lassen Sie den Wert auf dem Bildschirm ausgeben. Welchen Wert erhalten Sie?

Dazu existieren verschiedene Möglichkeiten, welche sich aufgrund ihres Programmieraufwands unterscheiden.

Wir wählen einen „minimal-invasiven“ Ansatz, bei dem die Zeichenketten aus der Struktur ausgelagert und lediglich Pointer auf die Zeichenketten in die Strukturen eingetragen werden.

Recherche
Speicherverwaltung mit `malloc` und `free`

> **Hinweis:** Lesen Sie ggf. die Funktionsweise von `malloc` und `free` nach, falls sie deren Aufgaben und Verwendungszweck nicht (mehr) kennen.

Ändern Sie Ihre `struct NameCounter` Definition.

Ändern Sie nun zunächst Ihre in Abschnitt 2.1 selbstdefinierte Struktur `struct NameCounter` so ab, dass das Element `Name` kein Array mehr ist, sondern ein **Pointer** auf ein `char`-Objekt.

Damit legt der Compiler nun beim Erzeugen einer Variable des Typs `struct NameCounter` keinen Platz mehr für die Buchstaben in der Zeichenkette an, sondern reserviert nur noch den Platz für den sehr viel kleineren Pointer auf die Zeichenkette. Die Größe der Strukturvariable im Speicher reduziert sich dadurch um ein Vielfaches.

> **Frage:** Ermitteln Sie mit `sizeof` die nun veränderte Größe einer Variable des Typs `struct NameCounter` im Speicher und lassen Sie den Wert auf dem Bildschirm ausgeben. Welchen Wert erhalten Sie?

> **Frage:** Verdeutlichen Sie sich den Unterschied zwischen der bisherigen Implementierung und der neuen Implementierung anhand einer kleinen Skizze, damit Sie verstehen, was die Änderung bewirkt.

> **Frage:** Versuchen Sie sich zu überlegen, wie Sie den Unterschied jemand anders erklären würden. Bringen Sie Ihre Skizze zur Abgabe mit, um sie für die Erklärung einzusetzen.

Ändern Sie Ihre `readNameDB`-Funktion.

Damit Ihr Programm mit dieser geänderten Definition wieder funktionsfähig wird, muss beim Einlesen eines neuen Vornamens natürlich der für den individuellen Namen jeweils erforderliche Speicherplatz reserviert werden.

Dafür müssen Sie in Ihrer Funktion `readNameDB` vor dem bereits vorhandenen Kopiervorgang des Vornamens aus dem Lesepuffer noch dem Pointer in der Struktur einen Pointer auf einen zuvor reservierten Speicherbereich zuweisen. Die Größe des zu reservierenden Speicherbereichs können Sie dabei mit `strlen` ermitteln.

> **Wichtig:** Die Funktion `strlen` zählt nur die Anzahl der sichtbaren Buchstaben in der Zeichenkette. Sie müssen bedenken, dass Sie stets ein zusätzliches Byte für das terminierende Nullbyte benötigen, das von `strlen` nicht mitgezählt wird.

Weitere Änderungen sind aufgrund unseres minimalistischen Ansatzes nicht nötig.

Hierbei machen wir uns die bereits aus früheren Aufgaben bekannte Äquivalenz von Pointern und Arrays zunutze. Alle Zugriffe auf Zeichenketten benötigen nur einen Pointer auf das erste Element und haben das bisher in der Struktur eingebettete Array wie einen Pointer auf sein erstes Zeichen interpretiert. Jetzt finden diese Funktion dort einen „echten“ Pointer und weitere Codeänderungen sind daher nicht nötig.

> **Hinweis:** Normalerweise müssen wir den mit `malloc` belegten Speicher wieder frei geben, sobald er nicht mehr benötigt wird. Da aber der Speicher in unserem Programm durchgehend bis zum Ende genutzt wird, und der Speicher am Programmende vom Betriebssystem automatisch wieder frei gegeben wird, lassen wir dies *ausnahmsweise* weg.

Wiederholen Sie die Zeitmessungen mit der optimierten Version Ihres Programms.

Wiederholen Sie nun die Zeitmessungen analog derer für die ursprüngliche Variante des Programms mehrmals und notieren Sie sich alle Einzelergebnisse.

> **Wichtig:** Stellen Sie sicher, dass Sie auch diese Messwerte im Abgabegespräch zur Verfügung haben!
>
> **Frage:** Was fällt ihnen bei den Messungen auf? Bewerten sie die Messungen bei beiden Varianten des Programms!
>
> Stellen Sie sicher, dass Sie die Messungen mit Ihren Dozenten diskutieren können.

Literatur

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson: *Introduction to Algorithms*, The MIT Press, 3rd edition, 2013.