



Programmierblatt 2

Ausgabe: 11.11.2021 12:00

Abgabe: 23.11.2021 08:00

Thema: Countsort, Pseudocode, Debug-Techniken

Abgabemodalitäten

1. Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf Deinem Rechner mit dem Befehl `clang -std=c11 -Wall -g` kompilieren.
2. Die Abgabe für den Quellcode erfolgt ausschließlich über unser Git im entsprechenden Branch. Nur wenn ein Ergebnis im [ISIS-Kurs](#) angezeigt wird, ist sichergestellt, dass die Abgabe erfolgt ist. Die Abgabe ist bestanden, wenn Du an Deinem Test einen grünen Haken siehst.
3. Du kannst bis zur Abgabefrist beliebig oft neue Versionen abgeben. Lies Dir die Hinweise der Tests genau durch, denn diese helfen Dir Deine Abgabe zu korrigieren.
Bitte beachte, dass ausschließlich die letzte Abgabe gewertet wird.
4. Die Abgabe erfolgt, sofern nicht anders angegeben, in folgendem Branch: `iprg-b<xx>-a<yy>`, wobei `<xx>` durch die zweistellige Nummer des Aufgabenblattes und `<yy>` durch die entsprechende Nummer der Aufgabe zu ersetzen sind.
5. Gib für jede Aufgabe die Quellcodedatei(en) gemäß der Vorgabe ab. Im [ISIS-Kurs](#) werden zum Teil Vorgabedateien bereitgestellt. Nutze diese zur Lösung der Aufgaben.
6. Die Abgabefristen werden vom Server überwacht. Versuche Deine Abgabe so früh wie möglich zu bearbeiten. Du minimierst so auch das Risiko, die Abgabefrist auf Grund von „technischen Schwierigkeiten“ zu versäumen. Eine Programmieraufgabe gilt als bestanden, wenn alle bewerteten Teilaufgaben bestanden sind.

valgrind zum Finden von Speicherfehlern: Student vs. Gauß (unbewertet)

Diese Aufgabe baut auf Vorlesung 8 des C-Kurses zum Thema `gdb` und Debugging auf. Sie thematisiert Speicherfehler und stellt `valgrind` als nützliches Tool vor, mit dem Speicherfehler in C gefunden werden können. Ziel der Aufgabe ist es, vorzustellen, wie in C mit Hilfe von `printf`, `gdb` und

`valgrind` die Funktionsweise von Programmen nachvollzogen und Fehler effektiv gefunden werden können. Der Programmcode zu dieser Aufgaben ist in unserem [ISIS-Kurs](#) hinterlegt. Eine Lösung muss nicht abgegeben werden.

Ein Student traut der Gaußschen Summenformel trotz aller induktiven Beweise nicht und bezweifelt, dass die Summe $1 + 2 + \dots + N$ wirklich gleich $(N + 1) \cdot N / 2$ ist. Um die Formel zu überprüfen, hat besagter Student ein recht komplexes Programm geschrieben, welches die Summe der ersten N natürlichen Zahlen aufaddiert und das Ergebnis anschließend mit dem Wert der Gaußschen Summenformel vergleicht.

Der Student hat sich Folgendes ausgedacht:

- Er alloziert dynamisch ein Array `array` der Länge N und schreibt in das Array die Werte 1, 2, ..., N . Die Zahl i steht somit im Array am Index $i - 1$.
- Die Zahlen im Array werden anschließend aufsummiert, indem über das Array `array` – von hinten nach vorne – iteriert wird. Befindet sich der Algorithmus am Index i , wird der Wert `array[i]` durch die Summe von `array[i]` und `array[i + 1]` ersetzt.
- Somit steht das Ergebnis¹ am Ende in `array[0]`.

Der Student ist sich sicher, dass sein Code korrekt arbeitet und meint somit Gauß widerlegen zu können: Bei manchen Werten unterscheidet sich nämlich sein Ergebnis von dem von Gauß ‘vorhergesagten’.

1. Führe das Programm für verschiedene Eingaben aus. Bei welchen Eingaben erhält der Student ein anderes Ergebnis als Gauß?
2. Untersuche den Code des Studenten. Versuche die Funktionsweise des Programms nachzuvollziehen, und decke etwaige Fehler auf.
3. Mache Vorschläge, wie die Fehler beseitigt werden können, und verbessert, falls möglich, den Code, so dass dieser besser lesbar / besser strukturiert / einfacher wird.

Bei der Suche nach Fehlern kann das Programm `valgrind` sehr hilfreich sein. Es deckt folgende Fehler im Zugriff auf Speicher auf, welche teilweise das Programm zum Abstürzen bringen können:

- Wird auf Speicherbereiche geschrieben, welche nicht vom Benutzer mittels `malloc` reserviert wurden, gibt `valgrind` „INVALID WRITE“ Fehlermeldungen aus und verweist auch auf die entsprechende Code-Zeile, in welcher der Zugriff erfolgte.
- Wird aus Speicherbereichen gelesen, welche nicht vom Benutzer mittels `malloc` reserviert wurden, gibt `valgrind` „INVALID READ“ Fehlermeldungen aus und verweist auch auf die entsprechende Code-Zeile, in welcher der Zugriff erfolgte.
- Wird mittels `malloc` allozierter Speicher nicht wieder freigegeben, so kann auch dies durch `valgrind` kenntlich gemacht werden. In der Ausgabe von `valgrind` finden sich dementsprechend Informationen zu „MEMORY LEAKS“ (Deutsch: undichtem, bzw. leckgeschlagenem Speicher).

¹Der Student war an dieser Stelle stolz darauf, dass keine weitere Variable benötigt wird.

Um valgrind zu nutzen, gehe wie folgt vor:

- Kompiliere das C-Programm zunächst mit der Debug-Option: `clang -std=c11 -Wall -g introprog_valgrind_debugging.c input_valgrind_debug.c -o introprog_valgrind_debugging`
- Rufe nun valgrind wie folgt auf: `valgrind ./introprog_valgrind_debugging`
- Betrachte die Ausgabe von Valgrind (für die ursprüngliche Version) und versuche die verschiedenen Fehlerarten sowie deren Ursprungsort zu lokalisieren.
- Die Optionen `--leak-check` bzw. `--verbose` können benutzt werden, um die Menge an ausgegebenen Informationen zu kontrollieren. Im Allgemeinen genügt jedoch der Aufruf ohne diese Parameter.

Aufgabe 1 Implementierung Count Sort (bewertet)

Implementiere anhand des Pseudocodes in Listing 1 und der Vorlesungsfolien die Funktion `count_sort()` in C. Beachte dabei, dass per Konvention Array-Indizes in Pseudocode bei **1** beginnen, in C jedoch bei **0**. Passe die Indizes in Deiner Implementierung also entsprechend an!

Listing 1: Pseudocode Count Sort

```
1 CountSort(Array A_in, Array A_out)
2   // C ist Hilfsarray mit 0 initialisiert
3   for j ← 1 to length(A_in) do
4     C[A_in[j]] ← C[A_in[j]] + 1
5
6   k ← 1
7   for j ← 1 to length(C) do
8     for i ← 1 to C[j] do
9       A_out[k] ← j
10      k ← k + 1
```

Die Funktion bekommt als Eingabeparameter zwei Integer-Arrays. Im Ersten werden die zu sortierenden Werte gespeichert und im Zweiten die nach Durchlauf des Algorithmus sortierte Folge von Werten.

Hinweis: Wir gehen zur Vereinfachung hier davon aus, dass nur Werte im Bereich $\{0, \dots, \text{MAX_VALUE}\}$ sortiert werden sollen. Der Wert `MAX_VALUE` wird hierbei in der Vorgabe als globale Variable definiert².

Die zu sortierenden Zahlen werden aus einer Datei eingelesen. Verwende dazu die in der Datei `arrayio.c` vorgegebene Funktion `read_array_from_file`. Gib am Ende Deines Programms das sortierte Array mit der Funktion `print_array` aus.

²Teste Deinen Code auch mit verschiedenen `MAX_VALUE` Werten und benutze dabei im Code immer die Variable `MAX_VALUE` anstatt einer festen Zahl wie 100.

Mach Dich zunächst mit der Signatur der vorgegebenen Funktionen vertraut, um diese korrekt aufzurufen (siehe Erklärung Aufgabe 1).

Auf unserem [ISIS-Kurs](#) findest Du eine Beispielliste mit zu sortierenden Zahlen in der Datei `zahlen_count_sort.txt`. Die Werte in dieser Datei dienen nur als Beispiele. Teste Dein Programm also mit unterschiedlichen Eingaben. Halte Dich dabei an das Format der Datei `zahlen_insertion_sort.txt`.

Das folgende Listing zeigt Dir einen beispielhaften Programmaufruf:

Listing 2: Programmbeispiel

```
1 > clang -std=c11 -Wall introprog_countsort.c arrayio.c \
2   -o introprog_countsort
3 > ./introprog_countsort zahlen_countsort.txt
4 Unsortiertes Array: 90 38 42 34 8 0 77 1 84 5 25 72 44 42 90 63 23
5 Sortiertes Array: 0 1 5 8 23 25 34 38 42 42 44 63 72 77 84 90 90
```

Nutze zur Lösung der Aufgabe die Vorgaben aus unserem [ISIS-Kurs](#). Füge Deine Lösung als Datei `introprog_countsort.c` im entsprechenden Abgabebereich in Dein persönliches Repository ein und übertrage die Lösung an die Abgabepattform.