Inf Programmieren 1, Übung 3

Klaus Kusche

1.) Buchstaben-Rechnen: Nochmals die Quersumme…

Berechne wie in der vorigen Übung die Quersumme einer eingegebenen Zahl
(nur die einfache, nicht die wiederholte), allerdings diesmal, *ohne* die Eingabe
in einen **`int`** zu verwandeln: *Greife direkt auf die einzelnen Buchstaben von* **`argv[1]`** *zu!*

Tipps:

- Ein String (Text), zum Beispiel ein Element von **`argv`**, ist ein Array von einzelnen **`char`**-Werten. Man kann daher mit **`[…]`** auf die einzelnen Zeichen eines Textes zugreifen, die Nummerierung beginnt wie üblich bei **`0`** statt bei **`1`**.

  Keine Scheu vor zwei **`[]`** unmittelbar nacheinander, es funktioniert wie erwartet:
  Zuerst das x-te Wort von **`argv`**, und dann das i-te Zeichen aus diesem Wort…

- Jeder (korrekt gespeicherte) Text enthält nach dem letzten sichtbaren Zeichen einen **`char`** mit dem ASCII-Wert 0 (als **`char`**-Konstante geschrieben **`'\0'`**) als Endemarkierung.

  Deine Schleife für das zeichenweise Durchgehen von **`argv[1]`** muss also *enden*, sobald sie bei einem Zeichen angekommen ist, das *gleich* **`'\0'`** ist.

Erinnere dich:

- Wie kommst du vom *Wert eines einzelnen Zeichens*, das eine Ziffer darstellt, zum *Zahlenwert*, den du für diese Ziffer zur Quersumme dazuzählen musst?

Zusatzaufgaben:

- Kannst du eine Fehlermeldung ausgeben und abbrechen, wenn die Eingabe ein Zeichen enthält, das *keine Ziffer* ist?

  Du hast zwei Möglichkeiten, auf eine Ziffer zu prüfen
  (wer es schafft, soll beide probieren!):

  1. Du kannst den ASCII-Wert des Zeichens selbst prüfen (wie?).
  2. Es gibt in **`ctype.h`** eine vordefinierte Funktion **`isdigit(c)`**.

  Tipp: **`%c`** ist der **`printf`**-Platzhalter für einen einzelnen **`char`**,
  wenn du das schuldige Zeichen in deiner Fehlermeldung ausgeben willst.

- Kannst du ein **`'-'`** ganz am Anfang der Eingabe (und nur dort!) ohne Fehlermeldung *überspringen*?

Scharf nachdenken:

- In welchem Punkt ist diese Version der vorigen überlegen?

2.) Bit-Rechnen: Umwandlung "Big Endian" / "Little Endian"

Manche Rechner (z.B. IBM Großrechner oder Apple Mac's, bevor sie auf x86-Prozessoren umgestellt wurden) legen aus mehreren Bytes bestehende Daten (z.B. einen **`int`**) so im Speicher ab, wie wir sie normalerweise anschreiben:
Das höchstwertige Byte (Bit 24 - 31) zuerst, das Byte mit den Bits 0 – 7 als letztes.
Auch am Internet werden alle Daten in dieser Reihenfolge übertragen,
man nennt das "*Big Endian*".

Andere Rechner (z.B. alle Rechner mit Intel- oder ARM-Prozessoren) speichern alle Daten "*Little Endian*", d.h. das hinterste Byte steht zuerst im Speicher. Die Zahl steht im Speicher also "auf dem Kopf": Byte 1 ist mit Byte 4 vertauscht und Byte 2 mit Byte 3.

- Verwandle eine einzelne auf der Commandline angegebene Zahl in einen **`int`**.
- Berechne daraus einen zweiten **`int`**,
  in dem die *Bytes wie oben angegeben vertauscht* sind.
- Gib beide *als 8-stellige Hexzahl (mit führenden Nullen)* aus.

Zur Big-Endian / Little-Endian-Umwandlung sollst du *nur Bit-Operationen auf* **`int`***'s*
verwenden, keine Multiplikation, Division oder Restbildung, keinen Zugriff
auf einzelne Bytes im Speicher, und auch nicht die dafür auf den meisten Systemen vorhandene vordefinierte Funktion.

Zur Ausgabe darfst du das *Hex-Format* **`%08X`** von **`printf`** verwenden
(achtstellig, mit führenden Nullen und großgeschriebenen Hex-Buchstaben).

Tipp: Mit *Hex-Konstanten* wird der Code leichter lesbar und klarer verständlich…

Achtung:

- Du wirst vermutlich *Bit-Schiebe-Operationen* brauchen.

  Wie deklarierst du deine **`int`**'s, damit es dabei
  keine unerwünschten "Überraschungen" gibt?

Scharf nachdenken:

- Wie siehst du den beiden Hexzahlen im Output an, ob du richtig gerechnet hast?

Für die Tüftler:

- Die in der Vorlesung gezeigte Variante funktioniert nur für **`int`**'s mit 4 Bytes Länge. Für **`int`**-Typen, die 2 oder 8 Bytes lang sind, müsste man den Code ändern.

  Versuche, statt einer großen Formel für alle 4 Bytes eine *Schleife* zu verwenden, die *pro Umlauf ein Byte* vom ursprünglichen **`int`** in das Ergebnis bringt
  und für *alle* **`int`**-Längen funktioniert.

  Du kannst mit **`short`** oder **`long long`** testen. Für **`long long`** brauchst du **`atoll(…)`** zum Umwandeln der Eingabe und **`%016llX`** statt **`%08X`** für die Ausgabe. Bei **`short`** funktioniert das normale **`atoi`** und **`%04hX`** in der Ausgabe.

  Die Größe (in Bytes) einer Variable **`x`** im Speicher bekommst du mit **`sizeof(x)`**, das solltest du statt einer fixen Anzahl von Bytes in deiner Schleifenbedingung verwenden.

3.) Bit-Rechnen: Ausgabe binär und hex

Verwandle alle auf der Commandline angegebenen Zahlen der Reihe nach in einen **`int`** (einlesen darfst du die Zahlen wie bisher mit **`atoi`**), und gib diesen *als 8-stellige Hexzahl und als 32-stellige Binärzahl aus (jeweils mit führenden Nullen)*.

Verwende zur Umwandlung *weder* **`printf`***-Formate*
(du sollst nur einzelne Zeichen mit **`printf %c`** oder effizienter **`putchar`** ausgeben), *noch Multiplikation, Division oder Restbildung*.

Auch für '0' oder '1' bei binär solltest du *weder ein* **`if`** *noch einen Conditional Operator* brauchen, nur für die Unterscheidung zwischen Hex 0-9 und Hex A-F ist ein **`if`** oder Conditional Operator erlaubt.

Zusatzaufgabe:

- Kannst du die Binärzahl schön in mit Zwischenräumen getrennten *Vierergruppen* ausgeben?