Human-Computer Interaction Group

Leibniz Universität Hannover

Übungsblatt 6

Programmieren 1 - WiSe 22/23

Prof. Dr. Michael Rohs, Jan Feuchter, M.Sc., Tim Dünte, M.Sc

Alle Übungen (bis auf die erste) müssen in Zweiergruppen bearbeitet werden. Beide Gruppenmitglieder müssen die Lösung der Zweiergruppe einzeln abgeben. Die Namen beider Gruppenmitglieder müssen sowohl in der PDF Abgabe, als auch als Kommentar in jeglichen Quelltextabgaben genannt werden. Plagiate führen zum Ausschluss von der Veranstaltung.

Abgabe bis Donnerstag den 24.11. um 23:59 Uhr über https://assignments.hci.uni-hannover.de/WiSe2022/Prog1. Die Abgabe muss aus einer einzelnen Zip-Datei bestehen, die den Quellcode, ein PDF für Freitextaufgaben und alle weiteren nötigen Dateien (z.B. Eingabedaten oder Makefiles) enthält. Lösen Sie Umlaute in Dateinamen auf.

Aufgabe 1: Quersummen

Um die Quersumme einer Zahl zu bilden werden die einzelnen Ziffern der Zahl aufaddiert. Die Quersumme von 123 ist also $6 = 1 + 2 + 3$. Bei der alternierenden Quersumme werden von Links nach Rechts die Ziffern an ungeraden Stellen addiert und die Ziffern an geraden Stellen subtrahiert. Die alternierende Quersumme von 537591 ist also $12 = 5 - 3 + 7 - 5 + 9 - 1$.

Die Template-Datei für diese Aufgabe ist **digit_sum.c**. Bearbeiten Sie die mit **todo** markierten Stellen. Die **digit_sum** Funktion soll die Quersumme der positiven Ganzzahl **number** berechnen. Über den Parameter **alternating** wird bestimmt ob die Quersumme alternierend ist (**true**) oder nicht (**false**).

a) Schreiben Sie ein Purpose Statement für die Funktion **digit_sum**, in dem Sie Parameter und Rückgabewert erläutern.

b) Implementieren Sie die Funktion **digit_sum** und definieren Sie mindestens 6 sinnvolle Testfälle.

c) **(OPTIONAL)** Eine Zahl ≥ 11 ist genau dann durch 11 teilbar, wenn ihre alternierende Quersumme 0 ist oder selbst durch 11 teilbar ist. Implementieren Sie die Funktion **divisible_by_eleven**. Nutzen Sie hier **nicht** den Modulo Operator. Negative Quersummen werden normalisiert, indem man sie wiederholt mit 11 addiert bis sie positiv sind. Definieren Sie sinnvolle Testfälle.

Aufgabe 2: Place-Value Notation

In dieser Aufgabe geht es um die Darstellung von Zahlen mit verschiedenen Basen über die Stellenwertsnotation. Das Template für diese Aufgabe ist die Datei **base_converter.c**.

a) Erstellen Sie ein Purpose Statement für die Funktion **int length_for_base(int number, int base)** inklusive Parameterbeschreibung und erklären Sie die Funktionalität. Schauen Sie sich dazu auch die Verwendung dieser Funktion in **String get_string_for_number_and_base(int number, int base)** an.

b) Implementieren Sie Funktion **String convert_to_base(int number, int base)**. Die Funktion bekommt eine Zahl **number** übergeben sowie eine Basis mit der diese dargestellt werden soll. Bspw. sollte **number = 5** und **base = 2** zu folgender Ausgabe führen: **101**. Basen sollen nur aus dem Intervall $[2, 36]$ gewählt werden können. Für die Darstellung von Stellen mit einer Wertigkeit von mehr als 9 sollen wie im Hexadezimalsystem Buchstaben verwendet werden. Bpsw. Wäre „Z" eine gültige Zahl in einem 36er-Zahlensystem und würde die Dezimalzahl 35 repräsentieren. Die Funktion **String get_string_for_number_and_base(int number, int base)** gibt Ihnen eine Zeichenkette passender Länge zurück, in der Sie mit der **s_set** Funktion aus der prog1lib die Ziffern einsetzen können. Beispiele für die Verwendung der benötigten Hilfsfunktionen finden Sie im Template.

Hinweise:

- Nutzen Sie die Funktionen **s_get** und **s_set** zum zeichenweisen Zugriff auf Strings.
- Nutzen Sie den String **characters**.
- Nutzen Sie den Modulo Operator **%** und die Integer Division **/**.

Aufgabe 3: Operationen auf einzelnen Bits

Diese Aufgabe baut auf der vorherigen Aufgabe auf und nutzt auch das Template **base_converter.c**. Die vorherige Aufgabe kann Ihnen helfen, da Sie mit der Funktion **convert_to_base** eine einfache Funktion haben, um die Zahlen binär darzustellen. Lösen Sie daher zuerst Aufgabe 2.

a) Schauen Sie sich die Funktion **void bit_operations()** an und beschreiben Sie die Funktionsweise der folgenden Operatoren **&, |, ^, << und >>** als Kommentar im Quelltext.

b) Implementieren Sie die Funktion **bool get_bit(int value, int index)** ohne die Nutzung von externen Funktionen oder Macros, mit der Sie ein einzelnes Bit aus einem Integer extrahieren können. Ist das Bit an Stelle index gleich 1 soll true zurückgegeben werden ansonsten false. Zum Beispiel. gibt der Aufruf von **get_bit(5, 0)** gibt den Wert des niederwertigsten Bits und der Aufruf **get_bit(-1, 31)** das höchstwertige Bit als bool zurück

c) Implementieren Sie die Funktion **int set_bit(int value, int index, bool bit)** ohne die Nutzung von externen Funktionen oder Macros, mit der Sie ein einzelnes Bit in einem Integer setzen können. Beispielsweise soll der Aufruf **set_bit(0, 31, true)** das höchstwertige Bit auf 1 setzen.

d) **(OPTIONAL)** Implementieren Sie die Funktion **int extract_bits(int value, int start, int end)**. Die Funktion soll einen beliebigen Bereich innerhalb eines Integers extrahieren. Dabei soll der Bereich **[start, end)]** extrahiert werden und zurückgegeben werden. Schauen Sie sich auch die Testfälle an. Beispielsweise sollte der Aufruf **extract_bits(0xf0, 4, 8)** (0xf0 entspricht 11110000) **0x0f** bzw. **1111** zurückgeben. Die Bits werden extrahiert und verschoben. Die Funktion kann die Funktionen aus b) und c) verwenden, muss dies aber nicht. Ansonsten ist die Nutzung von externen Funktionen oder Macros nicht erlaubt. Nutzen Sie **&, |, ^, << und >>**

Aufgabe 4: Parkhaus

Um einem Autofahrer die Parkplatzsuche zu vereinfachen, soll der Zustand eines Parkhauses, d.h. ob es noch freie Parkplätze gibt, auf einen Blick erkennbar sein. Die Template-Datei ist **park_house.c** Benutzen Sie die in Recipe for Enumerations beschriebene Vorgehensweise, um die Parkhauszustände zu definieren.

a) Implementieren Sie die Funktion **ParkHouseState det_park_house_state(int free_spots)**, welche die freien Parkhausplätze übergeben bekommt und basierend auf der Anzahl der freien Plätze den Zustand des Parkhauses zurückgibt.

   Dabei sollen folgende Grenzwerte gelten:

   - 0 freie Plätze → Volles Parkhaus
   - Weniger als 10 freie Plätze → Fast volles Parkhaus
   - 10 oder mehr freie Plätze → Freies Parkhaus

   Fügen Sie (sinnvolle) Tests hinzu. Dokumentieren Sie ihren Code mithilfe von Kommentaren.

b) Implementieren Sie die Funktion **String print_park_house_state(ParkHouseState state)**, welche einen String zurückgibt, der nähere Informationen zum Zustand des Parkhauses gibt.

c) **(OPTIONAL)** Verändern Sie das Programm so, dass man die Anzahl der freien Parkplätze in der Kommandozeile eingeben kann und das Ergebnis auch auf der Kommandozeile ausgegeben wird.