Stefan Kurtz **Universität Hamburg**
Abteilung für Genominformatik **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 2
Sommersemester 2020
Übungen zur Vorlesung: Ausgabe am 13.05.2020

Bitte beachten Sie die Hinweise zu den für diese Übungsaufgaben relevanten Abschnitten der Vorlesung. Diese Hinweise finden Sie jeweils am Ende der Beschreibung der Übungsaufgaben. Das Zoom-Meeting zur Klärung von Fragen zur Übung oder Vorlesung findet in dieser und nächster Woche bereits am Mittwoch von 12:00–16:00 Uhr statt.

Aufgabe 3.1 (7 Punkte) Im binären Zahlensystem wird eine natürliche Zahl als Summe von Zweierpotenzen dargestellt. Man schreibt diese Summe dann als Bitvektor, d.h. als eine Folge der Ziffern 0 und 1. Nehmen wir an, wir wollen alle Zahlen zwischen 0 und $2^{32} - 1$ darstellen. Dann benötigen wir 32 Bits. Z.B. kann man die Dezimalzahl 42 durch den Bitvektor `00000000 00000000 00000000 00101010` darstellen, denn es gilt:

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 32 + 0 + 8 + 0 + 2 + 0 = 42$$

In der obigen Darstellung des Bitvektors wurden zur besseren Lesbarkeit Leerzeichen eingefügt.

Wir setzen die folgende Definition voraus:

```
#define NUMOFBITS 32    /* number of bits in bitvector */
#define BITVECTOR_MAX_WIDTH (NUMOFBITS + NUMOFBITS/CHAR_BIT - 1)
```

Dabei ist die zweite Präprozessor-Konstante die maximale Länge der String-Repräsentation (inklusive der Leerzeichen) eines Bitvektors. Statt des Begriffs String-Repräsentation eines Bitvektors verwenden wir ab jetzt den Begriff Bitvektor.

Implementieren Sie in einer Datei `bintodec.c` die folgenden C-Funktionen zur Umwandlung von Dezimalzahlen in Bitvektoren und umgekehrt, sowie zur Validierung von Bitvektoren.

```
char *decimal2bitvector(unsigned int n)
int bitvector_validate(const char *bitvector)
unsigned int bitvector2decimal(const char *bitvector)
```

Die Funktion `decimal2bitvector()` soll jede nicht negative Dezimalzahl `n` verarbeiten können und einen Zeiger auf einen String zurückliefern, d.h. ein Array mit dem Basistyp **`char`**, das mit dem Zeichen \0 endet. Solche Strings nennt man auch C-Strings. Das Array besteht nur aus den Zeichen `'0'` oder `'1'` sowie Leerzeichen, und der Speicherbereich für das Array muss dynamisch mit `malloc` allokiert werden. Die maximale Länge des Strings ist `BITVECTOR_MAX_WIDTH`, es muss aber Speicherplatz für `BITVECTOR_MAX_WIDTH+1` Elemente des Basistyps allokiert werden.

`bitvector_validate` erhält einen Zeiger auf einen Bitvektor und verifiziert folgendes:

- Die Länge des Bitvektors, die man mit der Funktion `strlen` (siehe `C_slides.pdf`, Abschnitt 16) berechnen kann, liegt im Wertebereich zwischen `NUMOFBITS` und `BITVECTOR_MAX_WIDTH`.
- Die gesamte Anzahl der Vorkommen der Zeichen `'0'` und `'1'` im Bitvektor ist `NUMOFBITS`.
- Der Bitvektor enthält keine anderen Zeichen außer `'0'`, `'1'` und das Leerzeichen.

Wenn eine dieser Eigenschaften verletzt ist, soll eine entsprechende Fehlermeldung nach `stderr` geschrieben werden, und die Funktion soll mit einer **`return`**-Anweisung den Rückgabewert `-1` liefern. Wenn alle Eigenschaften erfüllt sind, dann soll die Funktion den Rückgabewert 0 liefern.

Die Funktion `bitvector2decimal` soll einen Bitvektor (mit oder ohne Leerzeichen) in einen Dezimalwert vom Typ **`unsigned int`** verwandeln und als **`return`**-Wert liefern. Sie können bei der Benutzung vorraussetzen, dass der Bitvektor bereits mit `bitvector_validate` validiert wurde, so dass keine Fehlerbehandlung erforderlich ist.

In den Materialien finden Sie ein Makefile und ein Hauptprogramm `bintodec_mn.c`. Durch `make` können Sie den Programmcode compilieren. Für die umfangreichen Tests gibt es eine Datei mit korrekt gebildeten Bitvektoren, eine Datei mit falschen Bitvektoren, ein Shell-Skript, sowie ein Ziel `test`. Sobald das Programm vollständig ist, können Sie durch `make test` verifizieren, dass es korrekt funktioniert. Dabei wird insbesondere auch getestet, dass bei falsch gebildeten Bitvektoren das Programm mit einer Fehlermeldung und dem exit-code `EXIT_FAILURE` abbricht. Wenn Ihr Programm auch im Fehlerfall korrekt funktioniert, erscheinen beim Aufruf von `make test` einige Fehlermeldungen vor der Ausgabe von `Congratulations ...`.

Das Hauptprogramm `bintodec_mn.c` ist relativ umfangreich. Schauen Sie sich die einzelnen Funktionen an und beantworten Sie die 6 Fragen innerhalb der Kommentarklammern.

Punktevergabe:

- jeweils 2 Punkte für die beiden Konvertierungsfunktionen,
- einen Punkt für die Validierungsfunktion, sowie
- 2 Punkte für die korrekte Beantwortung der Fragen.

Zur Bearbeitung dieser Aufgabe sollten Sie sich in alle Abschnitte der Vorlesung von „Syntactic Basics of C“bis „Functions“,außer die folgenden Abschnitte eingearbeitet haben:

- „Case Study on Classification“,
- „2Dim Arrays“,
- „Structures“,
- „Synopsis Pointer Notation“ und
- „Valgrind“.

Vom Abschnitt „Functions“sind nur die Seiten 208–221 aus `C_slides.pdf` wichtig.

Aufgabe 3.2 (3 Punkte) Betrachten Sie folgenden C-Code:

```
char c, *string, string2[3] = {0}, **strings;
int num, *nump, a, *b;

string = "Hallo Welt";
c = string[6];

string2[0] = 'A';
string2[1] = 'B';

strings = malloc(sizeof (*strings) * 3);
strings[0] = &c;
strings[1] = string2;
strings[2] = "third string";
```

```
a = 7;
nump = &a;
*nump = 5;
b = &a;
num = (int) (*(strings + 1))[1];
```

Dabei ist eine Wertzuweisung der Form `char *s = "abc";` eine Abkürzung für

```
char *s = {'a','b','c','\0'};
```

- Benennen Sie jeweils den Typ der 8 deklarierten Variablen.
- Geben Sie jeweils den Wert der folgenden Ausdrücke nach Ausührung des Programmcodes an. Falls ein Wert eine Speicheradresse ist, schreiben Sie „Adresse“. Sie müssen natürlich keinen konkreten Wert angeben.
  - `c`
  - `string2`
  - `string2[0]`
  - `*strings`
  - `strings[2][2]`
  - `num`
- Geben Sie für jeden der folgenden Ausdrücke an, ob sie wahr oder falsch sind.
  - `*strings[0] == *(string + 6)`
  - `b == nump`
  - `*b == *nump`
  - `string == strings[1]`

Zur Bearbeitung dieser Aufgabe sollten Sie sich die Folien aus `C_slides.pdf` zum Abschnitt „Pointers“ (Folie 78-81) und zum Abschnitt „Arrays“ (Folie 139-141, 169-174) angesehen haben. Die Zusammenfassung zu Zeigern (Folie 201-203) ist ggf. ebenso hilfreich.

Bitte die Lösungen zu diesen Aufgaben bis zum 20.05.2020 um 18:00 Uhr an pfn2@zbh.uni-hamburg.de schicken.

Checkliste für die Abgabe der Lösungen

Hier finden Sie noch einmal die wichtigsten Dinge, die Sie beachten müssen, bevor Sie Ihre Lösungen abgeben.

1. Wechseln Sie in das Verzeichnis mit den Übungen.
2. Benennen Sie ggf. Ihr Verzeichnis um. Im Material heißt es z.B. `Blatt01`. Ihr Verzeichnisname muss nach `Blatt01.` die Nachnamen der Gruppenmitglieder, jeweils durch einen Punkt getrennt enthalten.
3. Verifizieren Sie, dass die Tests funktionieren. Falls nicht, müssen entsprechende Kommentare eingefügt werden.
4. Rufen Sie in den Verzeichnissen mit einem `Makefile` den Befehl `make clean` auf.
5. Verifizieren Sie, dass Sie für alle Aufgaben die Angaben in der entsprechenden Datei `bearbeitung.txt` gemacht haben.
6. Rufen Sie `tar_loesungen.py N` auf, wobei `N` die Nummer des Übungblattes ist.
7. Wenn alles in Ordnung ist, dann wurde eine `.tar.gz`-Datei mit den Namen erzeugt, die sich aus der Blattnummer und den Namen in der Datei `namen.txt` ergibt.
8. Die `.tar.gz` schicken Sie als Anhang an die E-mail Adresse, die mit der Abkürzung des Modulnamens in Kleinschreibweise beginnt.