Week 02 Laboratory Sample Solutions

Objectives

- to practice using C's bitwise operations
- to understand how integer values are represented
- to practice manipulating dynamic memory
- to explore working with binary-coded decimal values
- to explore arbitrary precision integer arithmetic

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab02`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab02
$ cd lab02
$ 1521 fetch lab02
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE — INDIVIDUAL:

Convert 16 Binary Digits to A Signed Number

Download sixteen_in.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/20T3/activities/sixteen_in/files/sixteen_in.c .
```

Your task is to add code to this function in **sixteen_in.c**:

```
//
// given a string of binary digits ('1' and '0')
// return the corresponding signed 16 bit integer
//
int16_t sixteen_in(char *bits) {

    // PUT YOUR CODE HERE

    return 0;
}
```

Add code to the function `sixteen_in` so that, given a sixteen-character string containing an ASCII positional representation of a binary number, it returns the corresponding signed integer. For example:

```
$ ./sixteen_in 0000000000000000
0
$ ./sixteen_in 1111111111111111
-1
$ ./sixteen_in 0011001100110011
13107
$ ./sixteen_in 1111000011110000
-3856
```

HINT:

Write a loop which, for each character, sets the appropriate bit of a `int16_t`-type result variable, using the bitwise operators `|` and `<<`.

NOTE:

`sixteen_in` can assume it is given a string of exactly sixteen characters, and every character is either `'0'` or `'1'`.

You may define and call your own functions, if you wish.

You are not permitted to call any functions from the C library.

You are not permitted to change the `main` function you have been given, or to change `sixteen_in`'s prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest sixteen_in
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab02_sixteen_in sixteen_in.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `sixteen_in.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Convert string of binary digits to 16-bit signed integer

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>

#define N_BITS 16

int16_t sixteen_in(char *bits);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        printf("%d\n", sixteen_in(argv[arg]));
    }

    return 0;
}

// given a strings of binary digits ('1' and '0')
// return corresponding signed 16 bit integer

int16_t sixteen_in(char *bits) {
    assert(strlen(bits) == N_BITS);
    int16_t result = 0;
    for (int i = 0; i < N_BITS; i++) {
        if (bits[i] == '1') {
            result |= 1 << (N_BITS - i - 1);
        }
    }
    return result;
}
```

EXERCISE — INDIVIDUAL:

Convert a 16-bit Signed Number to Binary Digits

Download sixteen_out.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/20T3/activities/sixteen_out/files/sixteen_out.c .
```

Your task is to add code to this function in **sixteen_out.c**:

```
// given a signed 16 bit integer
// return a null-terminated string of 16 binary digits ('1' and '0')
// storage for string is allocated using malloc
char *sixteen_out(int16_t value) {

    // PUT YOUR CODE HERE

}
```

Add code to the function `sixteen_out` so that, given a 16-bit signed integer it returns a string containing sixteen binary digits (`'0'` or `'1'`). For example:

```
$ ./sixteen_out 0
0000000000000000
$ ./sixteen_out -1
1111111111111111
$ ./sixteen_out 13107
0011001100110011
$ ./sixteen_out -3856
1111000011110000
```

HINT:

Write a loop which, for each bit (determined using the bitwise operators `&` and `<<`) sets the corresponding character in the string to a `'0'` or `'1'`. This should be structurally very similar to `sixteen_in`.

`sixteen_out` returns a string, whose storage space must be allocated using *malloc*. A string is a NUL-terminated character array; remember to allocate enough space for all the characters *and* the terminating NUL byte.

NOTE:

`sixteen_out` can assume its input is a value between 32767 and -32768 inclusive.

You may define and call your own functions, if you wish.

You are not permitted to call any functions from the C library, other than *malloc*.

You are not permitted to change the `main` function you have been given, or to change `sixteen_out`'s prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest sixteen_out
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab02_sixteen_out sixteen_out.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `sixteen_out.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Convert a 16-bit signed integer to a string of binary digits

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

#define N_BITS 16

char *sixteen_out(int16_t value);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        long l = strtol(argv[arg], NULL, 0);
        assert(l >= INT16_MIN && l <= INT16_MAX);
        int16_t value = l;

        char *bits = sixteen_out(value);
        printf("%s\n", bits);

        free(bits);
    }

    return 0;
}

// given a signed 16 bit integer
// return a null-terminated string of 16 binary digits ('1' and '0')
// storage for string is allocated using malloc
char *sixteen_out(int16_t value) {
    char *buffer = malloc((N_BITS + 1) * sizeof(char));
    assert(buffer);

    for (int i = 0; i < N_BITS; i++) {
        int16_t bit_mask = 1 << (N_BITS - i - 1);
        if (value & bit_mask) {
            buffer[i] = '1';
        } else {
            buffer[i] = '0';
        }
    }

    buffer[N_BITS] = '\0';

    return buffer;
}
```

EXERCISE — INDIVIDUAL:

Convert a 2 digit BCD Value to an Integer

Download bcd.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/20T3/activities/bcd/files/bcd.c .
```

Your task is to add code to this function in **bcd.c**:

```c
// given a  BCD encoded value between 0 .. 99
// return corresponding integer
int bcd(int bcd_value) {

    // PUT YOUR CODE HERE

    return 0;
}
```

Add code to the function `bcd` so that, given a 2 digit Binary-Coded Decimal (BCD) value, it returns the corresponding integer.

In binary-coded decimal format, each byte holds 1 decimal value (0 to 9), so each byte contains 1 decimal digit. For example:

```
$ ./bcd 7
7
$ ./bcd 258             # 258 == 0x0102
12
$ ./bcd 1026            # 1026 == 0x0402
42
```

HINT:

Use the bitwise operators `&` and `>>` to extract each BCD digit.

NOTE:

`bcd` should return an integer value between 0 and 99 inclusive.

You may define and call your own functions if you wish.

You are not permitted to call any functions from the C library.

You are not permitted to change the `main` function you have been given, or to change `bcd`'s prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest bcd
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab02_bcd bcd.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `bcd.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Convert a 2 digit Binary Code Decimal value to an integer

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

int bcd(int bcd_value);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        long l = strtol(argv[arg], NULL, 0);
        assert(l >= 0 && l <= 0x0909);
        int bcd_value = l;

        printf("%d\n", bcd(bcd_value));
    }

    return 0;
}

// given a 2 digit BCD encoded value between 0 .. 9999
// return corresponding integer

int bcd(int bcd_value) {
    int bottom_digit = bcd_value & 0xF;
    assert(bottom_digit < 10);
    int top_digit = (bcd_value >> 8) & 0xF;
    assert(top_digit < 10);
    return top_digit * 10 + bottom_digit;
}
```

EXERCISE — INDIVIDUAL:

Convert an 8 digit Packed BCD Value to an Integer

Download packed_bcd.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/20T3/activities/packed_bcd/files/packed_bcd.c .
```

Your task is to add code to this function in **packed_bcd.c**:

```
// given a packed BCD encoded value between 0 .. 99999999
// return the corresponding integer
uint32_t packed_bcd(uint32_t packed_bcd_value) {

    // PUT YOUR CODE HERE

    return 0;
}
```

Add code to the function `packed_bcd` so that, given an eight-digit packed binary-coded decimal value, it returns the corresponding integer.

In packed binary-coded decimal format, each 4 bits holds 1 decimal value (0 to 9), so each byte contains 2 decimal digits. For example:

```
$ ./packed_bcd 66          # ... == 0x42
42
$ ./packed_bcd 39321       # ... == 0x9999
9999
$ ./packed_bcd 1111638594 # ... == 0x42424242
42424242
```

HINT:

Write a loop which extracts each BCD digit using the bitwise operators `&` and `>>`.

NOTE:

`packed_bcd` should return an integer value between 0 and 99999999 inclusive.

You may define and call your own functions if you wish.

You are not permitted to call any functions from the C library.

You are not permitted to change the `main` function you have been given, or to change `packed_bcd`'s prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest packed_bcd
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab02_packed_bcd packed_bcd.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `packed_bcd.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Convert a 8 digit Packed Binary Coded Decimal value to an integer

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

#define N_BCD_DIGITS 8

uint32_t packed_bcd(uint32_t packed_bcd);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        long l = strtol(argv[arg], NULL, 0);
        assert(l >= 0 && l <= UINT32_MAX);
        uint32_t packed_bcd_value = l;

        printf("%lu\n", (unsigned long)packed_bcd(packed_bcd_value));
    }

    return 0;
}

// given a packed BCD encoded value between 0 .. 99999999
// return the corresponding integer

uint32_t packed_bcd(uint32_t packed_bcd_value) {
    int result = 0;

    for (int i = N_BCD_DIGITS - 1; i >= 0; i--) {
        int decimal_digit = (packed_bcd_value >> (4 * i)) & 0xF;
        assert(decimal_digit < 10);
        result = result * 10 + decimal_digit;
    }

    return result;
}
```

CHALLENGE EXERCISE — INDIVIDUAL:

Add 2 Arbitrary Length BCD Values

Download bcd_add.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/20T3/activities/bcd_add/files/bcd_add.c .
```

Your task is to add code to this function in **bcd_add.c**:

```c
big_bcd_t *bcd_add(big_bcd_t *x, big_bcd_t *y) {

    // PUT YOUR CODE HERE

}
```

Add code to the function `bcd_add` so that, given 2 arbitrary length binary-coded decimal numbers, it returns their sum. For example:

```
$ ./bcd_add 123456789123456789 123456789123456789
246913578246913578
$ ./bcd_add 999999999999999999 1
1000000000000000000
$ ./bcd_add 77777777777777777777777777777777 88888888888888888888888888888888
166666666666666666666666666666666665
$ ./bcd_add 98765432198765432198765432 98765987659876598765
9876544207536419818642530 86
```

HINT:

Use *realloc* if you need to grow an array.

You will be working with pointers to `structs` with a field that is a pointer to an array. It would be wise to revise pointers, structs, typedefs, arrays, and dynamic memory allocation. The relevant videos in this playlist may help.

You can use Python (for example) to check what the sum of any two integers is:

```
$ python3 -i
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 123456789123456789 + 123456789123456789
246913578246913578
```

NOTE:

You may define and call your own functions, if you wish.

You are not permitted to call any functions from the C library, other than *malloc* and *realloc*.

You are not permitted to change the `main` function, to change any other functions you have been given, to change the type `big_bcd_t`, or to change the return type or argument types of `bcd_add`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest bcd_add
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab02_bcd_add bcd_add.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `bcd_add.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Add two arbitrary length Binary Coded Decimal Numbers
//

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <ctype.h>
#include <string.h>

//
// Store an arbitray length Binary Coded Decimal number
// bcd points to an array of size n_bcd
// each array element contains 1 decimal digit
//
// Store an arbitray length Binary Coded Decimal number.
// bcd points to an array of size n_bcd
// Each array element contains 1 decimal digit.
// Digits are stored in reverse order.
//
// For example if 42 is stored then
// n_bcd == 2
// bcd[0] == 0x02
// bcd[1] == 0x04
//

typedef struct big_bcd {
    unsigned char *bcd;
    int n_bcd;
} big_bcd_t;

big_bcd_t *bcd_add(big_bcd_t *x, big_bcd_t *y);
void bcd_print(big_bcd_t *number);
void bcd_free(big_bcd_t *number);
big_bcd_t *bcd_from_string(char *string);

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <number> <number>\n", argv[0]);
        exit(1);
    }

    big_bcd_t *left = bcd_from_string(argv[1]);
    big_bcd_t *right = bcd_from_string(argv[2]);

    big_bcd_t *result = bcd_add(left, right);

    bcd_print(result);
    printf("\n");

    bcd_free(left);
    bcd_free(right);
    bcd_free(result);

    return 0;
}

big_bcd_t *bcd_add(big_bcd_t *x, big_bcd_t *y) {
    big_bcd_t *sum = malloc(sizeof *sum);
    assert(sum);

    int n_digits = 0;
    if (x->n_bcd > y->n_bcd) {
        n_digits = x->n_bcd;
    } else {
        n_digits = y->n_bcd;
    }

    sum->n_bcd = n_digits;
    sum->bcd = malloc(n_digits * sizeof sum->bcd[0]);
    assert(sum->bcd);

    int carry = 0;
```

```c
    int carry = 0;
    for (size_t i = 0; i < n_digits; i++) {
        int digit_sum = carry;

        if (i < x->n_bcd) {
            digit_sum += x->bcd[i];
        }

        if (i < y->n_bcd) {
            digit_sum += y->bcd[i];
        }

        sum->bcd[i] = digit_sum % 10;
        carry = digit_sum / 10;
    }

    if (carry) {
        // we need to grow the array by to fit the carry digit
        sum->n_bcd = n_digits + 1;
        sum->bcd = realloc(sum->bcd, (n_digits + 1) * sizeof sum->bcd[0]);
        assert(sum->bcd);
        sum->bcd[n_digits] = carry;
    }

    return sum;
}

// print a big_bcd_t number
void bcd_print(big_bcd_t *number) {
    // if you get an error here your bcd_add is returning an invalid big_bcd_t
    assert(number->n_bcd > 0);
    for (int i = number->n_bcd - 1; i >= 0; i--) {
        putchar(number->bcd[i] + '0');
    }
}

// free storage for big_bcd_t number
void bcd_free(big_bcd_t *number) {
    // if you get an error here your bcd_add is returning an invalid big_bcd_t
    // or it is calling free for the numbers it is given
    free(number->bcd);
    free(number);
}

// convert a string to a big_bcd_t number
big_bcd_t *bcd_from_string(char *string) {
    big_bcd_t *number = malloc(sizeof *number);
    assert(number);

    int n_digits = strlen(string);
    assert(n_digits);
    number->n_bcd = n_digits;

    number->bcd = malloc(n_digits * sizeof number->bcd[0]);
    assert(number->bcd);

    for (int i = 0; i < n_digits; i++) {
        int digit = string[n_digits - i - 1];
        assert(isdigit(digit));
        number->bcd[i] = digit - '0';
    }

    return number;
}
```

Alternative solution for `bcd_add.c`

```c
// COMP1521 19t3 ... lab 3: bcd_add
// Add two arbitrary length Binary Coded Decimal Numbers
//
// 2019-10-06   Jashank Jeremy <jashank.jeremy@unsw.edu.au>

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <ctype.h>
#include <string.h>

//
// Store an arbitray length Binary Coded Decimal number.
// bcd points to an array of size n_bcd
// Each array element contains 1 decimal digit.
// Digits are stored in reverse order.
//
// For example if 42 is stored then
// n_bcd == 2
// bcd[0] == 0x02
// bcd[1] == 0x04
//

typedef struct big_bcd {
    unsigned char *bcd;
    int n_bcd;
} big_bcd_t;


big_bcd_t *bcd_add(big_bcd_t *x, big_bcd_t *y);
void bcd_print(big_bcd_t *number);
void bcd_free(big_bcd_t *number);
big_bcd_t *bcd_from_string(char *string);

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <number> <number>\n", argv[0]);
        exit(1);
    }

    big_bcd_t *left = bcd_from_string(argv[1]);
    big_bcd_t *right = bcd_from_string(argv[2]);

    big_bcd_t *result = bcd_add(left, right);

    bcd_print(result);
    printf("\n");

    bcd_free(left);
    bcd_free(right);
    bcd_free(result);

    return 0;
}


// DO NOT CHANGE THE CODE ABOVE HERE

typedef unsigned char bcd;

big_bcd_t *bcd_add (big_bcd_t *x, big_bcd_t *y)
{
    assert (x != NULL);
    assert (y != NULL);

    struct big_bcd *new = calloc (1, sizeof *new);
    assert (new != NULL);
    *new = (struct big_bcd) { .bcd = NULL, .n_bcd = 0 };

#ifndef max
#define max(a,b) (((a) > (b)) ? (a) : (b))
#endif

#define bcd_push(obj,i,n) \
    ({ \
```

```
    (obj)->bcd = realloc ((obj)->bcd, (size_t) ++((obj)->n_bcd)); \
    (obj)->bcd[(i)] = (n); \
})

bcd carry = 0;
int i = 0;
while (i < max (x->n_bcd, y->n_bcd)) {
    bcd xa = i < x->n_bcd ? x->bcd[i] : 0;
    bcd yb = i < y->n_bcd ? y->bcd[i] : 0;
    assert (
        0 <= xa && xa <= 9 &&
        0 <= yb && yb <= 9
    );

    bcd result = xa + yb + carry;
    assert (0 <= result);

    carry  = result / 10;
    result = result % 10;
    assert (0 <= carry);
    assert (0 <= result && result <= 9);

    bcd_push (new, i, result);
    i++;
}

assert (i == max (x->n_bcd, y->n_bcd));

assert (i == new->n_bcd);
assert (new->n_bcd == 0 || new->bcd != NULL);

if (carry)
    bcd_push (new, i, carry);

assert (i == new->n_bcd || i + 1 == new->n_bcd);

#undef bcd_push

    return new;
}


// DO NOT CHANGE THE CODE BELOW HERE


// print a big_bcd_t number
void bcd_print(big_bcd_t *number) {
    // if you get an error here your bcd_add is returning an invalid big_bcd_t
    assert(number->n_bcd > 0);
    for (int i = number->n_bcd - 1; i >= 0; i--) {
        putchar(number->bcd[i] + '0');
    }
}

// free storage for big_bcd_t number
void bcd_free(big_bcd_t *number) {
    // if you get an error here your bcd_add is returning an invalid big_bcd_t
    // or it is calling free for the numbers it is given
    free(number->bcd);
    free(number);
}

// convert a string to a big_bcd_t number
big_bcd_t *bcd_from_string(char *string) {
    big_bcd_t *number = malloc(sizeof *number);
    assert(number);

    int n_digits = strlen(string);
    assert(n_digits);
    number->n_bcd = n_digits;

    number->bcd = malloc(n_digits * sizeof number->bcd[0]);
    assert(number->bcd);

    for (int i = 0; i < n_digits; i++) {
```

```
        int digit = string[n_digits - i - 1];
        assert(isdigit(digit));
        number->bcd[i] = digit - '0';
    }

    return number;
}
```

CHALLENGE EXERCISE — INDIVIDUAL:

Subtract, Multiply and Divide 2 Arbitrary Length BCD Values

Download bcd_arithmetic.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/20T3/activities/bcd_arithmetic/files/bcd_arithmetic.c .
```

Add code to the functions `bcd_add`, `bcd_subtract`, `bcd_multiply`, and `bcd_divide` so that, given two arbitrary-length binary-coded decimal (BCD) numbers, they return the result of the corresponding arithmetic operation. For example:

```
$ ./bcd_arithmetic 1123456789123456789 - 1123456789123456788
1
$ ./bcd_arithmetic 123456789123456789 '*' 123456789123456789
15241578780673678515622620750190521
$ ./bcd_arithmetic 15241578780673678515622620750190521 / 123456789123456789
123456789123456789
$ ./bcd_arithmetic 123456789 '*' 987654321 +  987654321 / 1234
121932631113435637
$ ./bcd_arithmetic 14 / 5
2
```

HINT:

The code you are given already handles '+', '-', '*', and '/', and calls `bcd_add`, `bcd_subtract`, `bcd_multiply`, and `bcd_divide` respectively. You don't need to understand this code to do the exercises, though you will find it interesting to read. You only need to implement the four arithmetic functions.

Use *realloc* to grow an array.

You could use Python (for example) to check what the value of an expression is, (though you don't need to quote the `*`, and you need to use `//` to get integer division):

```
$ python3 -i
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 123456789 * 987654321 + 987654321 // 1234
121932631113435637
```

NOTE:

You can assume the results of subtraction are non-negative.

You can assume that divisors will be non-zero.

Integer division should yield only the integer part of the result. In other words, truncate towards zero; do not round.

You may define and call your own functions, if you wish.

You are not permitted to call any functions from the C library, other than *malloc* and *realloc*.

You are not permitted to change the `main` function, to change any other functions you have been given, to change the type `big_bcd_t`, or to change the return type or argument types of `bcd_add`, `bcd_subtract`, `bcd_multiply`, or `bcd_divide`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest bcd_arithmetic
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab02_bcd_arithmetic bcd_arithmetic.c
```

You must run `give` before **Monday 28 September 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `bcd_arithmetic.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Add two arbitrary length Binary Coded Decimal Numbers
//

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <ctype.h>
#include <string.h>

//
// Store an arbitray length Binary Coded Decimal number
// bcd points to an array of size n_bcd
// each array element contains 1 decimal digit
//

typedef struct big_bcd {
    unsigned char *bcd;
    int n_bcd;
} big_bcd_t;

void bcd_print(big_bcd_t *number);
void bcd_free(big_bcd_t *number);
big_bcd_t *bcd_from_string(char *string);

big_bcd_t *expression(char ***tokens);
big_bcd_t *term(char ***tokens);

int main(int argc, char *argv[]) {
    char **tokens = argv + 1;

    // tokens points in turn to each of the elements of argv
    // as the expression is evaluated.

    if (*tokens) {
        big_bcd_t *result = expression(&tokens);
        bcd_print(result);
        printf("\n");
        bcd_free(result);
    }

    return 0;
}

big_bcd_t *bcd_add(big_bcd_t *x, big_bcd_t *y) {
    big_bcd_t *sum = malloc(sizeof *sum);
    assert(sum);

    int n_digits = 0;
    if (x->n_bcd > y->n_bcd) {
        n_digits = x->n_bcd;
    } else {
        n_digits = y->n_bcd;
    }

    sum->n_bcd = n_digits;
    sum->bcd = malloc(n_digits * sizeof sum->bcd[0]);
    assert(sum->bcd);

    int carry = 0;
    for (size_t i = 0; i < n_digits; i++) {
        int digit_sum = carry;

        if (i < x->n_bcd) {
            digit_sum += x->bcd[i];
        }

        if (i < y->n_bcd) {
            digit_sum += y->bcd[i];
        }

        sum->bcd[i] = digit_sum % 10;
        carry = digit_sum / 10;
```

```
    carry = digit_sum / 10;
    }

    if (carry) {
        // we need to grow the array by to fit the carry digit
        sum->n_bcd = n_digits + 1;
        sum->bcd = realloc(sum->bcd, (n_digits + 1) * sizeof sum->bcd[0]);
        assert(sum->bcd);
        sum->bcd[n_digits] = carry;
    }

    return sum;
}

big_bcd_t *bcd_subtract(big_bcd_t *x, big_bcd_t *y) {
    // PUT YOUR CODE HERE
    return NULL;
}

big_bcd_t *bcd_multiply(big_bcd_t *x, big_bcd_t *y) {
    // PUT YOUR CODE HERE
    return NULL;
}

big_bcd_t *bcd_divide(big_bcd_t *x, big_bcd_t *y) {
    // PUT YOUR CODE HERE
    return NULL;
}

// print a big_bcd_t number
void bcd_print(big_bcd_t *number) {
    // if you get an error here your bcd_arithmetic is returning an invalid big_bcd_t
    assert(number->n_bcd > 0);
    for (int i = number->n_bcd - 1; i >= 0; i--) {
        putchar(number->bcd[i] + '0');
    }
}

// free storage for big_bcd_t number
void bcd_free(big_bcd_t *number) {
    // if you get an error here your bcd_arithmetic is returning an invalid big_bcd_t
    // or it is calling free for the numbers it is given
    free(number->bcd);
    free(number);
}

// convert a string to a big_bcd_t number
big_bcd_t *bcd_from_string(char *string) {
    big_bcd_t *number = malloc(sizeof *number);
    assert(number);

    int n_digits = strlen(string);
    assert(n_digits);
    number->n_bcd = n_digits;

    number->bcd = malloc(n_digits * sizeof number->bcd[0]);
    assert(number->bcd);

    for (int i = 0; i < n_digits; i++) {
        int digit = string[n_digits - i - 1];
        assert(isdigit(digit));
        number->bcd[i] = digit - '0';
    }

    return number;
}

// simple recursive descent evaluator for  big_bcd_t expressions
big_bcd_t *expression(char ***tokens) {
    big_bcd_t *left = term(tokens);
    assert(left);

    if (!**tokens || (***tokens != '+' && ***tokens != '-')) {
        return left;
    }
```

```
    char *operator= **tokens;
    (*tokens)++;

    big_bcd_t *right = expression(tokens);
    assert(right);

    big_bcd_t *result;
    if (operator[0] == '+') {
        result = bcd_add(left, right);
    } else {
        assert(operator[0] == '-');
        result = bcd_subtract(left, right);
    }
    assert(result);

    bcd_free(left);
    bcd_free(right);
    return result;
}

// evaluate a term of a big_bcd_t expression
big_bcd_t *term(char ***tokens) {
    big_bcd_t *left = bcd_from_string(**tokens);
    assert(left);
    (*tokens)++;

    if (!**tokens || (***tokens != '*' && ***tokens != '/')) {
        return left;
    }

    char *operator= **tokens;
    (*tokens)++;

    big_bcd_t *right = term(tokens);
    assert(right);

    big_bcd_t *result;
    if (operator[0] == '*') {
        result = bcd_multiply(left, right);
    } else {
        result = bcd_divide(left, right);
    }
    assert(result);

    bcd_free(left);
    bcd_free(right);
    return result;
}
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Mon Sep 28 21:00:00 2020** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```
