Exam number: 41

1. a) The output will be: 2 2.250000
   Since x = 10/4 + 1/4 which is equal to 2.75 but since x has the datatype of an integer the value is rounded to 2. y = 10 / 4 + 1.0 / 4 will give the output 2.25, the value is not rounded since it has the datatype of a double.

   b) The output will be: -3 -2 -1 0 1 2 3
   This is because the variable i has an initial value of 4, the loop is a never-ending loop except that we have one condition, if i is greater than 2 we break out of the loop. Which means that the print statement will print all numbers from -3 to 3 since the loop stops when i is greater than 2.

   c) The output will be: 28
   This is because we shift 7 by two positions with a bitwise operator.

   d) The output will be: 5 20
   The following code for this question was actually wrong in the exam paper:

```c
#include <stdio.h>
void f1(int x){
    x = 10;
}

void f2(int *x) {
    *x = 10 * x;
}

int main() {
    int x = 5, y = 2;
    f1(x);
    f2(&y);
    printf("%d %d", x, y);
    return 0;
}
```

   This is because in the function f2 we need to take the pointer to x and set it equal to 10 times the pointer of x. The code should look like following:

```c
#include <stdio.h>
void f1(int x){
    x = 10;
}

void f2(int *x) {
    *x = 10 * *x;
}

int main() {
    int x = 5, y = 2;
    f1(x);
    f2(&y);
    printf("%d %d", x, y);
    return 0;
}
```

2.

```c
#include <stdio.h>
#include <math.h>

int main() {
    double x, y;
    printf("Enter x and y: ");
    scanf("%lf%lf", &x, &y);

    double a = (1 + y) / (2 -x);
    double b = sqrt(x) + sqrt(y) + pow(x, 2) + pow(y, 2);
    double c = exp(x -y);
    double d = fabs(pow(x, 2) - pow(y, 2));

    printf("1 + y / 2 - x: %lf\n", a);
    printf("sqrt(x) + sqrt(y) + x^2 + y^2: %lf\n", b);
    printf("e^x-y: %lf\n", c);
    printf("|x^2 - y^2|: %lf\n", d);
}
```

3.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double calculateSum(double *numbers, int len) {
    double sum = 0.0f;
    for(int i = 0; i < len; ++i) {
        sum += numbers[i];
    }
    return sum;
}

double calculateMean(double *numbers, int len) {
    return calculateSum(numbers, len) / len;
}

int main() {
    int amountOfNumbers = 0;
    printf("Enter the amount of numbers you want to calculate the standard
deviation for: ");
    scanf("%d", &amountOfNumbers);

    double* numbers = malloc(amountOfNumbers * sizeof(int));
    if(numbers == NULL) {
        printf("Failed to allocate memory!\n");
        return -1;
    }

    // Get all numbers from the user.
    for(int i = 0; i < amountOfNumbers; ++i) {
        printf("Enter a number: ");
        scanf("%lf", &numbers[i]);
    }

    double temp = 0.0f;
    double mean = calculateMean(numbers, amountOfNumbers);
    for (int i = 0; i < amountOfNumbers; ++i) {
        // Formula used to calculate the standard deviation.
        temp += pow((numbers[i] - mean), 2);
    }
    double sdResult = sqrt(temp / amountOfNumbers);
    printf("Standard deviation: %lf", sdResult);

    // Free all the allocated memory to prevent memory leaks.
    free(numbers);
    return 0;
}
```

4.

```
int main() {
    int *x;
    scanf("%d", x);
    *x = *x * 10;
    printf("%d", *x);
    return 0;
}
```

The problem in this code is that “int *x” is uninitialized. It has no memory which means that when we read a value into x there’s nowhere to place it. So to fix it we need to dynamically allocate the memory using malloc which is shown in the following code.

```
int main() {
    int *x = malloc(sizeof(int));
    scanf("%d", x);
    *x = *x * 10;
    printf("%d", *x);

    free(x);
    return 0;
}
```

This also means that we have to free the memory afterwards cause otherwise it may cause memory-leaks which is never pleasant.

5. The best way to solve this problem is to write a linked list built with a struct.

```
struct Book {
    char title[120];
    char category[70];
    int edition;
    int publish_year;
    double price;
    struct Book *next;
};
```

the struct could be changed to something like this. The point of doing it this way is so we can keep a pointer to the next item in the list which means that every book will have a link to the next book which then creates a long list. The benefit is that the list can be big, the only limit is basically our computer memory (RAM).

A function to add a new book would look something like the following:

```c
struct Book* addBooks(struct Book* prev) {
    printf("Enter all the book information: ");
    char keyboardInput[256];
    fgets(keyboardInput, sizeof(keyboardInput), stdin);

    struct Book* addedBook = malloc(sizeof(struct Book));
    sscanf(keyboardInput, "%s %s %d %d %lf",
           addedBook->title,
           addedBook->category,
           &addedBook->edition,
           &addedBook->publish_year,
           &addedBook->price);

    if(prev) {
        prev->next = addedBook;
    }
    return addedBook;
}
```

The add function above could then be used when loading the books into memory from the file “books.dat”. Assume that we have a function that reads the file by line in binary format and then parses the data from every line and then adds that into a new list using the “addBooks” function above. When the data is loaded into memory sorting the books can be done with some basic if & else statements when the list is being looped over. The sorting and writing would look something like this assuming that we have function that writes a list to a given filepath:

```c
if(!strcmp(bookList->category, "Mathematics") && bookList->publish_year >
2015) {
    writeBooksToFile(bookList, "books1.dat");
} else if(!strcmp(bookList->category, "Mathematics") && bookList-
>publish_year < 2015) {
    writeBooksToFile(bookList, "books2.dat");
} else {
    writeBooksToFile(bookList, "books3.dat");
}
```