202018021
Fu, Ziyu

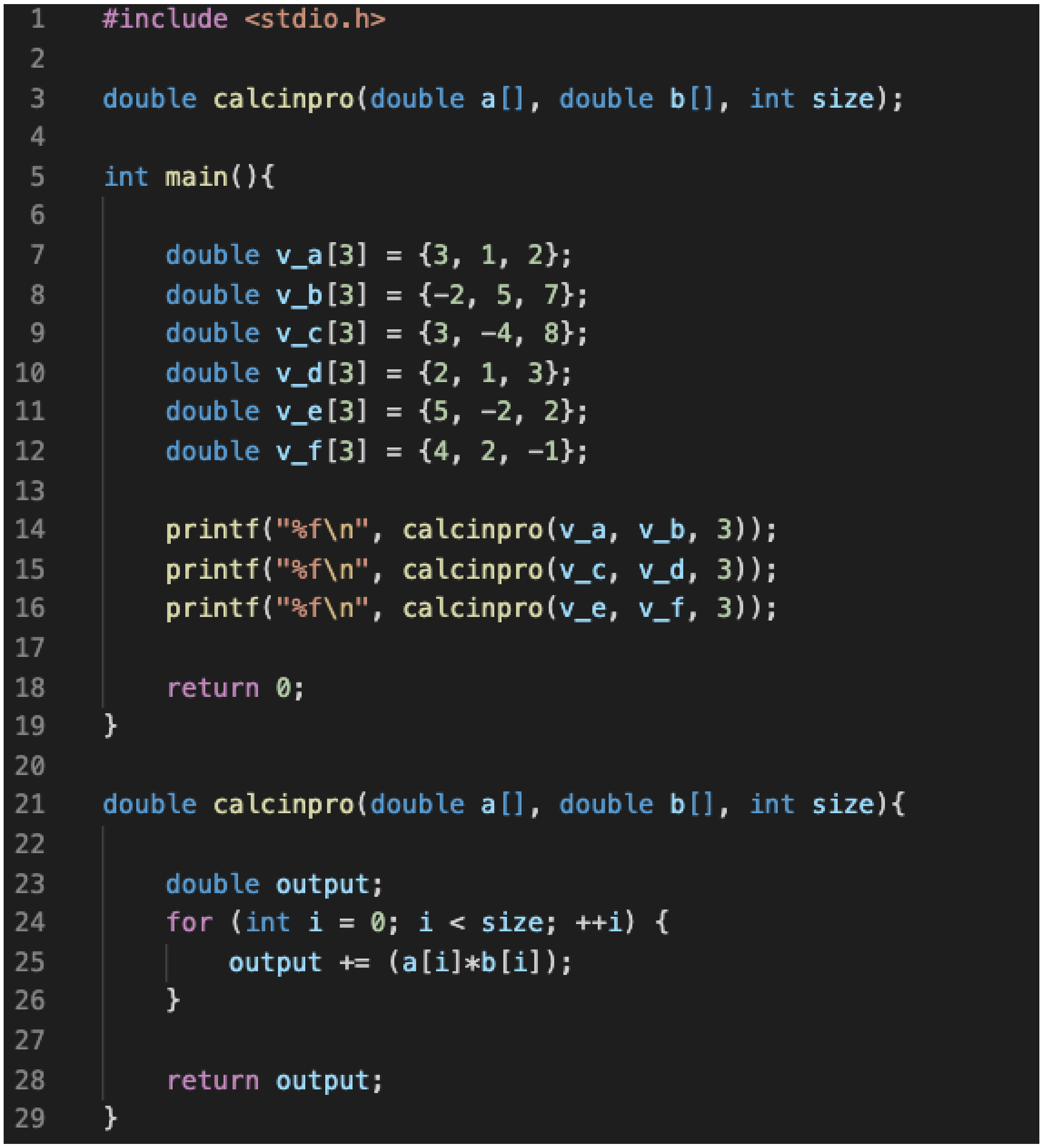
Programming I Chapter 5 Task 1, 2, 3
2021/06/03

Task 1

1. Objective
   To create a function that finds the inner product of two vectors.

2. Strategy of solving
   - Understand the calculation for scalar product
   - Design the code
   - Observe the outputs

3. Program code



```c
#include <stdio.h>

double calcinpro(double a[], double b[], int size);

int main(){

    double v_a[3] = {3, 1, 2};
    double v_b[3] = {-2, 5, 7};
    double v_c[3] = {3, -4, 8};
    double v_d[3] = {2, 1, 3};
    double v_e[3] = {5, -2, 2};
    double v_f[3] = {4, 2, -1};

    printf("%f\n", calcinpro(v_a, v_b, 3));
    printf("%f\n", calcinpro(v_c, v_d, 3));
    printf("%f\n", calcinpro(v_e, v_f, 3));

    return 0;
}

double calcinpro(double a[], double b[], int size){

    double output;
    for (int i = 0; i < size; ++i) {
        output += (a[i]*b[i]);
    }

    return output;
}
```

4. Results and discussions

<u>Console output:</u>

```
13.000000
26.000000
14.000000
```

The program outputs the correct results as expected.

(Comments and possible improvements)

The possible inputs to this function are in fact heavily constrained by the task statement. I did not actually have to account for dealing with certain other possibilities like having different vector lengths or none integer elements in the vector, as the task only asks for the scalar product of three sets of specific vectors. But I did it anyway for good measure.

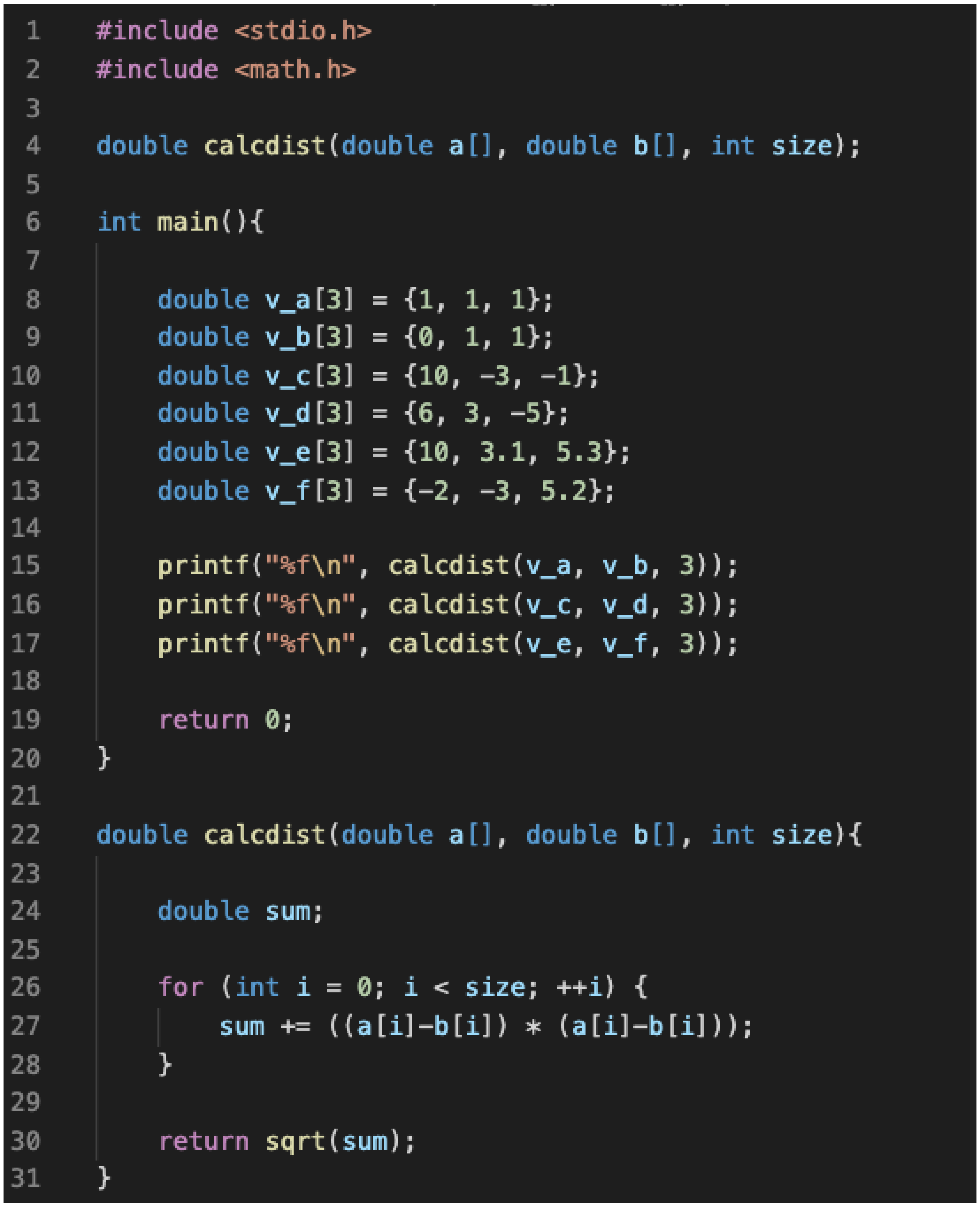
It would also have been a good idea to check if the vectors have the same length before doing the actual calculation, and return an error code if the calculation is not legal at all (dot product of two vectors of different lengths). It might have also been possible to use `sizeof` to find the length of the array so that the user won't have to explicitly pass the size as an argument into the function. However, I believe the `sizeof` function returns the size of the pointer instead of the size of the array itself when the array is being passed into a function as an argument. I got lazy and passing it explicitly is the simplest solution that came to my mind.

Task 2

1. Objective
   To create a function that calculates the distance between two points in space.

2. Strategy of solving
   - Understand the calculation for point distance in space
   - Design the code
   - Observe the outputs

3. Program code



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

double calcdist(double a[], double b[], int size);

int main(){

    double v_a[3] = {1, 1, 1};
    double v_b[3] = {0, 1, 1};
    double v_c[3] = {10, -3, -1};
    double v_d[3] = {6, 3, -5};
    double v_e[3] = {10, 3.1, 5.3};
    double v_f[3] = {-2, -3, 5.2};

    printf("%f\n", calcdist(v_a, v_b, 3));
    printf("%f\n", calcdist(v_c, v_d, 3));
    printf("%f\n", calcdist(v_e, v_f, 3));

    return 0;
}

double calcdist(double a[], double b[], int size){

    double sum;

    for (int i = 0; i < size; ++i) {
        sum += ((a[i]-b[i]) * (a[i]-b[i]));
    }

    return sqrt(sum);
}
```

4. Results and discussions

   Console output:

```
1.000000
8.246211
13.461798
```

   The program outputs the correct results as expected.

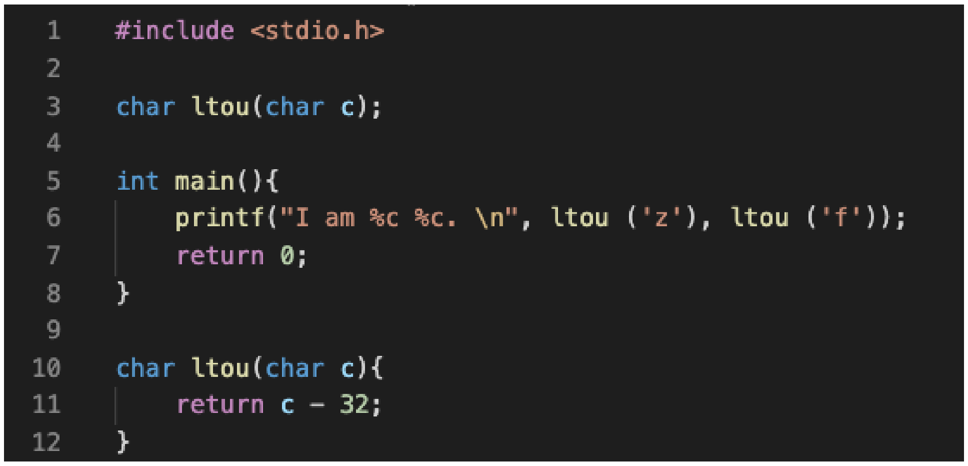
   Again, the inputs are supposedly heavily constrained so the points are strictly in 3-dimensional space. However, this program can calculate point distances in higher dimensions too.

Task 3

1. Objective
   To write your own **`toupper()`**.

2. Strategy of solving
   - Understand the relationship (codepoint value difference) between lower and upper case letters in standard US-ASCII encoding
   - Design the code
   - Observe the outputs

3. Program code



```c
#include <stdio.h>

char ltou(char c);

int main(){
    printf("I am %c %c. \n", ltou ('z'), ltou ('f'));
    return 0;
}

char ltou(char c){
    return c - 32;
}
```

4. Results and discussions
   Console output:

```
I am Z F.
```

The program outputs as expected.

The codepoint values of lower and upper letters of the same letter are **`0x20`** apart from each other in US-ASCII, which is 32 in decimal value. This code obviously only works for converting lowercase to uppercase, and hence will break if characters other than **`0x61`** to **`0x7A`** are inputted into the function.