```c
#ifndef CALCULATION
#define CALCULATION

#include <stdbool.h>

#define _USE_MATH_DEFINES
#include <math.h>

#include "../datatypes/enum.h"
#include "../datatypes/struct.h"

#include "print.h"
#include "selection.h"

#define ONES 1
#define TENS 10
#define HUNDREDS 100
#define THOUSANDS 1000

double GetParameterInput(void (*paramInstructions)(char *parameter), char *parameter)
{
    char *endptr, buffer[100];
    double number;

    (*paramInstructions)(parameter);

    while (fgets(buffer, sizeof(buffer), stdin))
    {
        number = strtod(buffer, &endptr);
        if (endptr == buffer || *endptr != '\n')
        {
            NumericInputAlert(false);
        }
        else if (number <= 0)
        {
            NumericInputAlert(true);
        }
        else
        {
            return number;
        }
    }
}

void AssignRectangleParameter(struct History *history, int base)
{
    history->rectangles[history->count[0]].width = GetParameterInput(ParamaterSelectionInstructions, "width") / base;
    history->rectangles[history->count[0]].length = GetParameterInput(ParamaterSelectionInstructions, "length") / base;
}

void GetRectangleParameter(struct History *history, enum unit *unit)
{
    switch (*unit)
    {
    case m:
        AssignRectangleParameter(&(*history), ONES);
        break;

    case dm:
        AssignRectangleParameter(&(*history), TENS);
        break;

    case cm:
        AssignRectangleParameter(&(*history), HUNDREDS);
        break;

    case mm:
        AssignRectangleParameter(&(*history), THOUSANDS);
        break;
    }
}

void AssignSquareParameter(struct History *history, int base)
{
    history->squares[history->count[1]].length = GetParameterInput(ParamaterSelectionInstructions, "length") / base;
}

void GetSquareParameter(struct History *history, enum unit *unit)
{
    switch (*unit)
    {
    case m:
        AssignSquareParameter(&(*history), ONES);
        break;

    case dm:
        AssignSquareParameter(&(*history), TENS);
        break;

    case cm:
        AssignSquareParameter(&(*history), HUNDREDS);
        break;

    case mm:
```

```
        AssignSquareParameter(&(*history), THOUSANDS);
        break;
    }
}

void AssignCircleParameter(struct History *history, int base)
{
    history->circles[history->count[2]].radius = GetParameterInput(ParamaterSelectionInstructions, "radius") / base;
}

void GetCircleParameter(struct History *history, enum unit *unit)
{
    switch (*unit)
    {
    case m:
        AssignCircleParameter(&(*history), ONES);
        break;

    case dm:
        AssignCircleParameter(&(*history), TENS);
        break;

    case cm:
        AssignCircleParameter(&(*history), HUNDREDS);
        break;

    case mm:
        AssignCircleParameter(&(*history), THOUSANDS);
        break;
    }
}

void AssignCuboidParameter(struct History *history, int base)
{
    history->cuboids[history->count[3]].width = GetParameterInput(ParamaterSelectionInstructions, "width") / base;
    history->cuboids[history->count[3]].length = GetParameterInput(ParamaterSelectionInstructions, "length") / base;
    history->cuboids[history->count[3]].height = GetParameterInput(ParamaterSelectionInstructions, "height") / base;
}

void GetCuboidParameter(struct History *history, enum unit *unit)
{
    switch (*unit)
    {
    case m:
        AssignCuboidParameter(&(*history), ONES);
        break;

    case dm:
        AssignCuboidParameter(&(*history), TENS);
        break;

    case cm:
        AssignCuboidParameter(&(*history), HUNDREDS);
        break;

    case mm:
        AssignCuboidParameter(&(*history), THOUSANDS);
        break;
    }
}

void AssignCubeParameter(struct History *history, int base)
{
    history->cubes[history->count[4]].length = GetParameterInput(ParamaterSelectionInstructions, "length") / base;
}

void GetCubeParameter(struct History *history, enum unit *unit)
{
    switch (*unit)
    {
    case m:
        AssignCubeParameter(&(*history), ONES);
        break;

    case dm:
        AssignCubeParameter(&(*history), TENS);
        break;

    case cm:
        AssignCubeParameter(&(*history), HUNDREDS);
        break;

    case mm:
        AssignCubeParameter(&(*history), THOUSANDS);
        break;
    }
}

void AssignSphereParameter(struct History *history, int base)
{
    history->spheres[history->count[5]].radius = GetParameterInput(ParamaterSelectionInstructions, "radius") / base;
}

void GetSphereParameter(struct History *history, enum unit *unit)
{
```

```
    switch (*unit)
    {
    case m:
        AssignSphereParameter(&(*history), ONES);
        break;

    case dm:
        AssignSphereParameter(&(*history), TENS);
        break;

    case cm:
        AssignSphereParameter(&(*history), HUNDREDS);
        break;

    case mm:
        AssignSphereParameter(&(*history), THOUSANDS);
        break;
    }
}

void AssignConeParameter(struct History *history, int base)
{
    history->cones[history->count[6]].radius = GetParameterInput(ParamaterSelectionInstructions, "radius") / base;
    history->cones[history->count[6]].height = GetParameterInput(ParamaterSelectionInstructions, "height") / base;
}

void GetConeParameter(struct History *history, enum unit *unit)
{
    switch (*unit)
    {
    case m:
        AssignConeParameter(&(*history), ONES);
        break;

    case dm:
        AssignConeParameter(&(*history), TENS);
        break;

    case cm:
        AssignConeParameter(&(*history), HUNDREDS);
        break;

    case mm:
        AssignConeParameter(&(*history), THOUSANDS);
        break;
    }
}

void CalculateProperties(enum shape shape, struct History *history)
{
    enum unit unit;

    UnitSelection(&unit);

    switch (shape)
    {
    case Rectangle:
        GetRectangleParameter(&(*history), &unit);

        history->rectangles[history->count[0]].perimeter = 2 * (history->rectangles[history->count[0]].width + history->rectangles[history-
>count[0]].length);
        history->rectangles[history->count[0]].area = history->rectangles[history->count[0]].width * history->rectangles[history->count[0]].length;

        DisplayResults(shape, history->rectangles[history->count[0]].perimeter, history->rectangles[history->count[0]].area);
        history->count[0]++;

        break;

    case Square:
        GetSquareParameter(&(*history), &unit);

        history->squares[history->count[1]].perimeter = 4 * history->squares[history->count[1]].length;
        history->squares[history->count[1]].area = history->squares[history->count[1]].length * history->squares[history->count[1]].length;

        DisplayResults(shape, history->squares[history->count[1]].perimeter, history->squares[history->count[1]].area);
        history->count[1]++;

        break;

    case Circle:
        GetCircleParameter(&(*history), &unit);

        history->circles[history->count[2]].circumference = 2 * M_PI * history->circles[history->count[2]].radius;
        history->circles[history->count[2]].area = M_PI * history->circles[history->count[2]].radius * history->circles[history->count[2]].radius;

        DisplayResults(shape, history->circles[history->count[2]].circumference, history->circles[history->count[2]].area);
        history->count[2]++;

        break;

    case Cuboid:
        GetCuboidParameter(&(*history), &unit);

        history->cuboids[history->count[3]].area = 2 * (history->cuboids[history->count[3]].width * history->cuboids[history->count[3]].length + history-
>cuboids[history->count[3]].width * history->cuboids[history->count[3]].height + history->cuboids[history->count[3]].length * history->cuboids[history-
```

```
>count[3]].height);
283         history->cuboids[history->count[3]].volume = history->cuboids[history->count[3]].width * history->cuboids[history->count[3]].length * history->cuboids[history->count[3]].height;
284
285         DisplayResults(shape, history->cuboids[history->count[3]].area, history->cuboids[history->count[3]].volume);
286         history->count[3]++;
287
288         break;
289
290     case Cube:
291         GetCubeParameter(&(*history), &unit);
292
293         history->cubes[history->count[4]].area = 6 * history->cubes[history->count[4]].length * history->cubes[history->count[4]].length;
294         history->cubes[history->count[4]].volume = history->cubes[history->count[4]].length * history->cubes[history->count[4]].length * history->cubes[history->count[4]].length;
295
296         DisplayResults(shape, history->cubes[history->count[4]].area, history->cubes[history->count[4]].volume);
297         history->count[4]++;
298
299         break;
300
301     case Sphere:
302         GetSphereParameter(&(*history), &unit);
303
304         history->spheres[history->count[5]].area = 4 * M_PI * history->spheres[history->count[5]].radius * history->spheres[history->count[5]].radius;
305         history->spheres[history->count[5]].volume = 4 / 3 * M_PI * history->spheres[history->count[5]].radius * history->spheres[history->count[5]].radius * history->spheres[history->count[5]].radius;
306
307         DisplayResults(shape, history->spheres[history->count[5]].area, history->spheres[history->count[5]].volume);
308         history->count[5]++;
309
310         break;
311
312     case Cone:
313         GetConeParameter(&(*history), &unit);
314
315         history->cones[history->count[6]].area = M_PI * history->cones[history->count[6]].radius * (history->cones[history->count[6]].radius + sqrt(history->cones[history->count[6]].radius * history->cones[history->count[6]].radius + history->cones[history->count[6]].height * history->cones[history->count[6]].height));
316         history->cones[history->count[6]].volume = M_PI * history->cones[history->count[6]].radius * history->cones[history->count[6]].radius * history->cones[history->count[6]].height / 3;
317
318         DisplayResults(shape, history->cones[history->count[6]].area, history->cones[history->count[6]].volume);
319         history->count[6]++;
320
321         break;
322     }
323 }
324
325 void CalculateHistoricalProperties(struct History *history)
326 {
327     enum shape shape;
328     int i, parameters;
329     double *means, *stds;
330
331     ShapeAndObjectSelection(&shape);
332
333     switch (shape)
334     {
335     case Rectangle:
336         parameters = 4;
337         if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
338         {
339             NoMemoryAlert();
340             exit(1);
341         }
342
343         if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
344         {
345             NoMemoryAlert();
346             exit(1);
347         }
348
349         for (i = 0; i < parameters; i++)
350         {
351             means[i] = 0;
352             stds[i] = 0;
353         }
354
355         for (i = 0; i < history->count[0]; i++)
356         {
357             means[0] += history->rectangles[i].width;
358             means[1] += history->rectangles[i].length;
359             means[2] += history->rectangles[i].perimeter;
360             means[3] += history->rectangles[i].area;
361         }
362
363         means[0] /= history->count[0];
364         means[1] /= history->count[0];
365         means[2] /= history->count[0];
366         means[3] /= history->count[0];
367
368         for (i = 0; i < history->count[0]; i++)
369         {
370             stds[0] += pow(history->rectangles[i].width - means[0], 2);
371             stds[1] += pow(history->rectangles[i].length - means[1], 2);
```

```
            stds[2] += pow(history->rectangles[i].perimeter - means[2], 2);
            stds[3] += pow(history->rectangles[i].area - means[3], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[0]);
        stds[1] = sqrt(stds[1] / history->count[0]);
        stds[2] = sqrt(stds[2] / history->count[0]);
        stds[3] = sqrt(stds[3] / history->count[0]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;

    case Square:
        parameters = 3;

        if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        for (i = 0; i < parameters; i++)
        {
            means[i] = 0;
            stds[i] = 0;
        }

        for (i = 0; i < history->count[1]; i++)
        {
            means[0] += history->squares[i].length;
            means[1] += history->squares[i].perimeter;
            means[2] += history->squares[i].area;
        }

        means[0] /= history->count[1];
        means[1] /= history->count[1];
        means[2] /= history->count[1];

        for (i = 0; i < history->count[1]; i++)
        {
            stds[0] += pow(history->squares[i].length - means[0], 2);
            stds[1] += pow(history->squares[i].perimeter - means[1], 2);
            stds[2] += pow(history->squares[i].area - means[2], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[1]);
        stds[1] = sqrt(stds[1] / history->count[1]);
        stds[2] = sqrt(stds[2] / history->count[1]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;

    case Circle:
        parameters = 3;

        if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        for (i = 0; i < parameters; i++)
        {
            means[i] = 0;
            stds[i] = 0;
        }

        for (i = 0; i < history->count[2]; i++)
        {
            means[0] += history->circles[i].radius;
            means[1] += history->circles[i].circumference;
            means[2] += history->circles[i].area;
        }

        means[0] /= history->count[2];
        means[1] /= history->count[2];
        means[2] /= history->count[2];

        for (i = 0; i < history->count[2]; i++)
```

```
        {
            stds[0] += pow(history->circles[i].radius - means[0], 2);
            stds[1] += pow(history->circles[i].circumference - means[1], 2);
            stds[2] += pow(history->circles[i].area - means[2], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[2]);
        stds[1] = sqrt(stds[1] / history->count[2]);
        stds[2] = sqrt(stds[2] / history->count[2]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;

    case Cuboid:
        parameters = 5;

        if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        for (i = 0; i < parameters; i++)
        {
            means[i] = 0;
            stds[i] = 0;
        }

        for (i = 0; i < history->count[3]; i++)
        {
            means[0] += history->cuboids[i].length;
            means[1] += history->cuboids[i].width;
            means[2] += history->cuboids[i].height;
            means[3] += history->cuboids[i].area;
            means[4] += history->cuboids[i].volume;
        }

        means[0] /= history->count[3];
        means[1] /= history->count[3];
        means[2] /= history->count[3];
        means[3] /= history->count[3];
        means[4] /= history->count[3];

        for (i = 0; i < history->count[3]; i++)
        {
            stds[0] += pow(history->cuboids[i].length - means[0], 2);
            stds[1] += pow(history->cuboids[i].width - means[1], 2);
            stds[2] += pow(history->cuboids[i].height - means[2], 2);
            stds[3] += pow(history->cuboids[i].area - means[3], 2);
            stds[4] += pow(history->cuboids[i].volume - means[4], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[3]);
        stds[1] = sqrt(stds[1] / history->count[3]);
        stds[2] = sqrt(stds[2] / history->count[3]);
        stds[3] = sqrt(stds[3] / history->count[3]);
        stds[4] = sqrt(stds[4] / history->count[3]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;

    case Cube:
        parameters = 3;

        if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        for (i = 0; i < parameters; i++)
        {
            means[i] = 0;
            stds[i] = 0;
        }

        for (i = 0; i < history->count[4]; i++)
        {
            means[0] += history->cubes[i].length;
```

```
            means[1] += history->cubes[i].area;
            means[2] += history->cubes[i].volume;
        }

        means[0] /= history->count[4];
        means[1] /= history->count[4];
        means[2] /= history->count[4];

        for (i = 0; i < history->count[4]; i++)
        {
            stds[0] += pow(history->cubes[i].length - means[0], 2);
            stds[1] += pow(history->cubes[i].area - means[1], 2);
            stds[2] += pow(history->cubes[i].volume - means[2], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[4]);
        stds[1] = sqrt(stds[1] / history->count[4]);
        stds[2] = sqrt(stds[2] / history->count[4]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;

    case Sphere:
        parameters = 3;

        if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        for (i = 0; i < parameters; i++)
        {
            means[i] = 0;
            stds[i] = 0;
        }

        for (i = 0; i < history->count[5]; i++)
        {
            means[0] += history->spheres[i].radius;
            means[1] += history->spheres[i].area;
            means[2] += history->spheres[i].volume;
        }

        means[0] /= history->count[5];
        means[1] /= history->count[5];
        means[2] /= history->count[5];

        for (i = 0; i < history->count[5]; i++)
        {
            stds[0] += pow(history->spheres[i].radius - means[0], 2);
            stds[1] += pow(history->spheres[i].area - means[1], 2);
            stds[2] += pow(history->spheres[i].volume - means[2], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[5]);
        stds[1] = sqrt(stds[1] / history->count[5]);
        stds[2] = sqrt(stds[2] / history->count[5]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;

    case Cone:
        parameters = 4;

        if ((means = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        if ((stds = (double *)malloc(parameters * sizeof(double))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }

        for (i = 0; i < parameters; i++)
        {
            means[i] = 0;
            stds[i] = 0;
        }

        for (i = 0; i < history->count[6]; i++)
        {
```

```
            means[0] += history->cones[i].radius;
            means[1] += history->cones[i].height;
            means[2] += history->cones[i].area;
            means[3] += history->cones[i].volume;
        }

        means[0] /= history->count[6];
        means[1] /= history->count[6];
        means[2] /= history->count[6];
        means[3] /= history->count[6];

        for (i = 0; i < history->count[6]; i++)
        {
            stds[0] += pow(history->cones[i].radius - means[0], 2);
            stds[1] += pow(history->cones[i].height - means[1], 2);
            stds[2] += pow(history->cones[i].area - means[2], 2);
            stds[3] += pow(history->cones[i].volume - means[3], 2);
        }

        stds[0] = sqrt(stds[0] / history->count[6]);
        stds[1] = sqrt(stds[1] / history->count[6]);
        stds[2] = sqrt(stds[2] / history->count[6]);
        stds[3] = sqrt(stds[3] / history->count[6]);

        DisplayHistoryTable(shape, history, means, stds);
        free(means);
        free(stds);
        break;
    }
}

#endif
```