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

#include <ctype.h>
#include <string.h>
#include <stdbool.h>

#define _USE_MATH_DEFINES
#include <math.h>

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

#include "functions/print.h"
#include "functions/selection.h"
#include "functions/calculation.h"

int main()
{
    int dimension, i;
    enum shape shape;
    struct History history;

    // Initialize history count.
    for (i = 0; i < 7; i++)
    {
        history.count[i] = 0;
    }

    DisplayTitle("assets/title.txt");

    while (true)
    {
        DimensionSelection(&dimension);

        if (!GeometrySelection(&shape, dimension))
        {
            continue;
        }

        CalculateProperties(shape, &history);

        while (ProcessSelection())
        {
            CalculateHistoricalProperties(&history);
        }
    }

    return 0;
}
```

```c
#ifndef SELECTION
#define SELECTION

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

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

#include "print.h"

char *toLower(char *string)
{
    unsigned char *char_ptr = (unsigned char *)string;

    while (*char_ptr)
    {
        *char_ptr = tolower(*char_ptr);
        char_ptr++;
    }
}

bool ShapeSelection(enum shape *shape)
{
    char *input;

    ShapeSelectionInstructions();

    while (true)
    {
        if ((input = (char *)malloc(100 * sizeof(char))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }
        fgets(input, 100 * sizeof(char), stdin);
        toLower(input);

        if (strcmp(input, "rectangle\n") == 0 || strcmp(input, "1\n") == 0)
        {
            free(input);
            *shape = Rectangle;
            return true;
        }
        else if (strcmp(input, "square\n") == 0 || strcmp(input, "2\n") == 0)
        {
            free(input);
            *shape = Square;
            return true;
        }
        else if (strcmp(input, "circle\n") == 0 || strcmp(input, "3\n") == 0)
        {
            free(input);
            *shape = Circle;
            return true;
        }
        else if (strcmp(input, "back\n") == 0)
        {
            free(input);
            return false;
        }
        else if (strcmp(input, "exit\n") == 0)
        {
            free(input);
            exit(0);
        }
        else
        {
            WrongShapeInput();
        }
    }
}

bool ObjectSelection(enum shape *shape)
{
    char *input;

    ObjectSelectionInstructions();

    while (true)
    {
        if ((input = (char *)malloc(100 * sizeof(char))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }
        fgets(input, 100 * sizeof(char), stdin);
        toLower(input);

        if (strcmp(input, "cuboid\n") == 0 || strcmp(input, "1\n") == 0)
        {
            free(input);
            *shape = Cuboid;
            return true;
        }
        else if (strcmp(input, "cube\n") == 0 || strcmp(input, "2\n") == 0)
        {
            free(input);
            *shape = Cube;
            return true;
        }
        else if (strcmp(input, "sphere\n") == 0 || strcmp(input, "3\n") == 0)
        {
            free(input);
            *shape = Sphere;
            return true;
        }
        else if (strcmp(input, "cone\n") == 0 || strcmp(input, "4\n") == 0)
        {
            free(input);
            *shape = Cone;
            return true;
        }
        else if (strcmp(input, "back\n") == 0)
        {
            free(input);
            return false;
        }
        else if (strcmp(input, "exit\n") == 0)
        {
            free(input);
            exit(0);
        }
        else
        {
            WrongObjectInput();
        }
    }
}

bool GeometrySelection(enum shape *shape, int dimension)
{
    switch (dimension)
    {
    case 2:
        return ShapeSelection(&(*shape));
        break;
    case 3:
        return ObjectSelection(&(*shape));
        break;
    }
    return false;
}

void DimensionSelection(int *dimension)
{
    char *input;

    DimensionSelectionInstructions();

    while (true)
    {
        if ((input = (char *)malloc(100 * sizeof(char))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }
        fgets(input, 100 * sizeof(char), stdin);
        toLower(input);

        if (strcmp(input, "2d\n") == 0 || strcmp(input, "1\n") == 0)
        {
            free(input);
            *dimension = 2;
            return;
        }
        else if (strcmp(input, "3d\n") == 0 || strcmp(input, "2\n") == 0)
        {
            free(input);
            *dimension = 3;
            return;
        }
        else if (strcmp(input, "exit\n") == 0)
        {
            free(input);
            exit(0);
        }
        else
        {
            WrongDimensionInput();
        }
    }
}

void UnitSelection(enum unit *unit)
{
    char *input;

    UnitSelectionInstructions();

    while (true)
    {
        if ((input = (char *)malloc(100 * sizeof(char))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }
        fgets(input, 100 * sizeof(char), stdin);
        toLower(input);

        if (strcmp(input, "m\n") == 0 || strcmp(input, "1\n") == 0)
        {
            *unit = m;
            free(input);
            return;
        }
        else if (strcmp(input, "dm\n") == 0 || strcmp(input, "2\n") == 0)
        {
            *unit = dm;
            free(input);
            return;
        }
        else if (strcmp(input, "cm\n") == 0 || strcmp(input, "3\n") == 0)
        {
            *unit = cm;
            free(input);
            return;
        }
        else if (strcmp(input, "mm\n") == 0 || strcmp(input, "4\n") == 0)
        {
            *unit = mm;
            free(input);
            return;
        }
        else
        {
            WrongUnitInput();
        }
    }
}

bool ProcessSelection()
{
    char *input;

    ProcessSelectionInstructions();

    while (true)
    {
        if ((input = (char *)malloc(100 * sizeof(char))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }
        fgets(input, 100 * sizeof(char), stdin);
        toLower(input);

        if (strcmp(input, "history\n") == 0 || strcmp(input, "1\n") == 0)
        {
            free(input);
            return true;
        }
        else if (strcmp(input, "calculate\n") == 0 || strcmp(input, "2\n") == 0)
        {
            free(input);
            return false;
        }
        else if (strcmp(input, "exit\n") == 0 || strcmp(input, "3\n") == 0)
        {
            free(input);
            exit(0);
        }
        else
        {
            WrongProcessInput();
        }
    }
}

void ShapeAndObjectSelection(enum shape *shape)
{
    char *input;

    ShapeAndObjectSelectionInstructions();

    while (true)
    {

        if ((input = (char *)malloc(100 * sizeof(char))) == NULL)
        {
            NoMemoryAlert();
            exit(1);
        }
        fgets(input, 100 * sizeof(char), stdin);
        toLower(input);

        if (strcmp(input, "rectangle\n") == 0 || strcmp(input, "1\n") == 0)
        {
            free(input);
            *shape = Rectangle;
            return;
        }
        else if (strcmp(input, "square\n") == 0 || strcmp(input, "2\n") == 0)
        {
            free(input);
            *shape = Square;
            return;
        }
        else if (strcmp(input, "circle\n") == 0 || strcmp(input, "3\n") == 0)
        {
            free(input);
            *shape = Circle;
            return;
        }
        else if (strcmp(input, "cuboid\n") == 0 || strcmp(input, "4\n") == 0)
        {
            free(input);
            *shape = Cuboid;
            return;
        }
        else if (strcmp(input, "cube\n") == 0 || strcmp(input, "5\n") == 0)
        {
            free(input);
            *shape = Cube;
            return;
        }
        else if (strcmp(input, "sphere\n") == 0 || strcmp(input, "6\n") == 0)
        {
            free(input);
            *shape = Sphere;
            return;
        }
        else if (strcmp(input, "cone\n") == 0 || strcmp(input, "7\n") == 0)
        {
            free(input);
            *shape = Cone;
            return;
        }
        else if (strcmp(input, "exit\n") == 0)
        {
            free(input);
            exit(0);
        }
        else
        {
            WrongShapeAndObjectInput();
        }
    }
}

#endif
```

```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);
        history->cuboids[history->count[3]].volume = history->cuboids[history->count[3]].width * history->cuboids[history->count[3]].length * history->cuboids[history->count[3]].height;

        DisplayResults(shape, history->cuboids[history->count[3]].area, history->cuboids[history->count[3]].volume);
        history->count[3]++;

        break;

    case Cube:
        GetCubeParameter(&(*history), &unit);

        history->cubes[history->count[4]].area = 6 * history->cubes[history->count[4]].length * history->cubes[history->count[4]].length;
        history->cubes[history->count[4]].volume = history->cubes[history->count[4]].length * history->cubes[history->count[4]].length * history->cubes[history->count[4]].length;

        DisplayResults(shape, history->cubes[history->count[4]].area, history->cubes[history->count[4]].volume);
        history->count[4]++;

        break;

    case Sphere:
        GetSphereParameter(&(*history), &unit);

        history->spheres[history->count[5]].area = 4 * M_PI * history->spheres[history->count[5]].radius * history->spheres[history->count[5]].radius;
        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;

        DisplayResults(shape, history->spheres[history->count[5]].area, history->spheres[history->count[5]].volume);
        history->count[5]++;

        break;

    case Cone:
        GetConeParameter(&(*history), &unit);

        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));
        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;

        DisplayResults(shape, history->cones[history->count[6]].area, history->cones[history->count[6]].volume);
        history->count[6]++;

        break;
    }
}

void CalculateHistoricalProperties(struct History *history)
{
    enum shape shape;
    int i, parameters;
    double *means, *stds;

    ShapeAndObjectSelection(&shape);

    switch (shape)
    {
    case Rectangle:
        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[0]; i++)
        {
            means[0] += history->rectangles[i].width;
            means[1] += history->rectangles[i].length;
            means[2] += history->rectangles[i].perimeter;
            means[3] += history->rectangles[i].area;
        }

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

        for (i = 0; i < history->count[0]; i++)
        {
            stds[0] += pow(history->rectangles[i].width - means[0], 2);
            stds[1] += pow(history->rectangles[i].length - means[1], 2);
```

```c
            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
```

```c
#ifndef PRINT
#define PRINT

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

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

#define MAX_LEN 128

void DisplayImage(FILE *fptr)
{
    char readString[MAX_LEN];

    while (fgets(readString, sizeof(readString), fptr) != NULL)
        printf("%s", readString);
}

void DisplayTitle(char *filename)
{
    FILE *fptr = NULL;

    if ((fptr = fopen(filename, "r")) == NULL)
    {
        fprintf(stderr, "Error opening %s!\n", filename);
        exit(1);
    }
    DisplayImage(fptr);
}

void NoMemoryAlert()
{
    printf("\n============================================================================\n");
    printf("============================ Not enough memory! ============================\n");
    printf("============================================================================\n\n");
}

void DimensionSelectionInstructions()
{
    printf("\n============================================================================\n");
    printf("Calculate 2D or 3D object's properties? Type \"Exit\" if you want to leave the program:\n");
    printf("1. 2D\n2. 3D\n");
    printf("Type in your choice here: ");
}

void WrongDimensionInput()
{
    printf("\n============================================================================\n");
    printf("============== Invalid input! Please follow the instructions! ==============\n");
    printf("============ Key in \"2D\", \"3D\" or the choice index (\"1\" or \"2\"). ===========\n");
    printf("===================== Type \"Exit\" to leave the program. ====================\n");
    printf("============================================================================\n\n");
    printf("Type in your choice again here: ");
}

void ShapeSelectionInstructions()
{
    printf("\nSelect the shape to calculate its properties. Type \"Exit\" if you want to leave the program or \"Back\" if you want to reselect the
dimension:\n");
    printf("1. Rectangle\n2. Square\n3. Circle\n");
    printf("Type in your choice here: ");
}

void WrongShapeInput()
{
    printf("\n============================================================================\n");
    printf("============== Invalid input! Please follow the instructions! ==============\n");
    printf("================= Key in \"Rectangle\", \"Square\" or \"Circle\". ================\n");
    printf("===================== Type \"Exit\" to leave the program. ====================\n");
    printf("================== Type \"Back\" to reselect the dimension. ==================\n");
    printf("============================================================================\n\n");
    printf("Type in your choice again here: ");
}

void ObjectSelectionInstructions()
{
    printf("\nSelect the object to calculate its properties. Type \"Exit\" if you want to leave the program or \"Back\" if you want to reselect the
dimension:\n");
    printf("1. Cuboid\n2. Cube\n3. Sphere\n4. Cone\n");
    printf("Type in your choice here: ");
}

void WrongObjectInput()
{
    printf("\n============================================================================\n");
    printf("============== Invalid input! Please follow the instructions! ==============\n");
    printf("=============== Key in \"Cuboid\", \"Cube\", \"Sphere\" or \"Cone\". ===============\n");
    printf("===================== Type \"Exit\" to leave the program. ====================\n");
    printf("=================== Type \"Back\" to reselect the dimension. =================\n");
    printf("============================================================================\n\n");
    printf("Type in your choice again here: ");
}

void UnitSelectionInstructions()
{
    printf("\nSelect the input unit:\n");
    printf("1. m\n2. dm\n3. cm\n4. mm\n");
    printf("Select unit: ");
}

void WrongUnitInput()
{
    printf("\n============================================================================\n");
    printf("============== Invalid input! Please follow the instructions! ==============\n");
    printf("====================== Enter \"m\", \"dm\", \"cm\" or \"mm\". ======================\n");
    printf("============= Or enter the choice index: \"1\", \"2\", \"3\" or \"4\". =============\n");
    printf("============================================================================\n\n");
    printf("Enter again here: ");
}

void DisplayResults(enum shape shape, double result_1, double result_2)
{
    bool is2D = false;

    printf("\nCalculation results:\n");
    printf("    ______________________________________________________________________________________________");

    if (shape == Rectangle || shape == Square || shape == Circle)
    {
        is2D = true;
    }

    if (is2D)
    {
        if (shape != Circle)
        {
            printf("\n   |    Perimeter  | ");
        }
        else
        {
            printf("\n   | Circumference | ");
        }

        printf("%12.2g m   | %12.2g dm   | %12.2g cm   | %12.2g mm   |\n", result_1, result_1 * 10, result_1 * 1E2, result_1 * 1E3);
        printf("   |_______________|__________________|___________________|___________________|___________________|\n");
        printf("   |      Area     | %12.2g m^2 | %12.2g dm^2 | %12.2g cm^2 | %12.2g mm^2 |\n", result_2, result_2 * 1E2, result_2 * 1E4, result_2 * 1E6);
        printf("   |_______________|__________________|___________________|___________________|___________________|\n");
    }
    else
    {
        printf("___________\n");
        printf("   |    Surface area  | %12.2g m^2   | %12.2g dm^2   | %12.2g cm^2   | %12.2g mm^2   |\n", result_1, result_1 * 1E2, result_1 * 1E4, result_1 *
1E6);
        printf("   |__________________|____________________|_____________________|_____________________|_____________________|\n");
        printf("   |      Volume      | %12.2g m^3   | %12.2g dm^3   | %12.2g cm^3   | %12.2g mm^3   |\n", result_2, result_2 * 1E3, result_2 * 1E6, result_2 *
1E9);
        printf("   |__________________|____________________|_____________________|_____________________|_____________________|\n");
    }
}

void ParamaterSelectionInstructions(char *parameter)
{
    printf("\nEnter the %s parameter\n", parameter);
    printf("Enter the value here: ");
}

void NumericInputAlert(bool isNumeric)
{
    if (isNumeric)
    {
        printf("\n============================================================================\n");
        printf("========================= Enter a positive number! =========================\n");
        printf("============================================================================\n");
    }
    else
    {
        printf("\n============================================================================\n");
        printf("============================== Enter a number! =============================\n");
        printf("============================================================================\n");
    }
    printf("Enter again here: ");
}

void ProcessSelectionInstructions()
{
    printf("\nSelect:\n");
    printf("1. History\t- To view the calculation history.\n");
    printf("2. Calculate\t- To calculate again.\n");
    printf("3. Exit\t\t- To leave the program.\n");
    printf("Enter your choice here: ");
}

void WrongProcessInput()
{
    printf("\n============================================================================\n");
    printf("============== Invalid input! Please follow the instructions! ==============\n");
    printf("============ Key in \"History\", \"Calculate\" or \"Exit\" ===========\n");
    printf("============================================================================\n\n");
    printf("Type in your choice again here: ");
}

void ShapeAndObjectSelectionInstructions()
{
    printf("\nSelect any of the option\n");
    printf("1. Rectangle\n2. Square\n3. Circle\n");
    printf("4. Cuboid\n5. Cube\n6. Sphere\n7. Cone\n");
    printf("Type in your choice here: ");
}

void WrongShapeAndObjectInput()
{
    printf("\n============================================================================\n");
    printf("============== Invalid input! Please follow the instructions! ==============\n");
    printf("============ Key in \"Rectangle\", \"Square\", \"Circle\", \"Cuboid\", \"Cube\", \"Sphere\", \"Cone\" ===========\n");
    printf("============================================================================\n\n");
    printf("Type in your choice again here: ");
```

```
}

void DisplayHistoryTable(enum shape shape, struct History *history, double *means, double *stds)
{
    int i;

    switch (shape)
    {
    case Rectangle:
        if (history->count[0] == 0)
        {
            printf("\n============================================================================\n");
            printf("================ The rectangle calculation history is empty. ===============\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Rectangle\n");
            printf("    ____________________________________________________________________________________________________\n");
            printf("   |         Index        |       Width      |       Lenth      |     Perimeter    |        Area        |\n");
            printf("   |______________________|__________________|__________________|__________________|____________________|\n");

            for (i = 0; i < history->count[0]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", i + 1, history->rectangles[i].width, history-
>rectangles[i].length, history->rectangles[i].perimeter, history->rectangles[i].area);
                printf("   |______________________|__________________|__________________|__________________|____________________|\n");
            }

            printf("   |         Mean         | %12.2g m   | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", means[0], means[1], means[2], means[3]);
            printf("   |______________________|__________________|__________________|__________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", stds[0], stds[1], stds[2], stds[3]);
            printf("   |______________________|__________________|__________________|__________________|____________________|\n");
        }

        break;

    case Square:
        if (history->count[1] == 0)
        {
            printf("\n============================================================================\n");
            printf("================= The square calculation history is empty. =================\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Square\n");
            printf("    _________________________________________________________________________________\n");
            printf("   |         Index        |    Side Lenth    |     Perimeter    |        Area        |\n");
            printf("   |______________________|__________________|__________________|____________________|\n");

            for (i = 0; i < history->count[1]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", i + 1, history->squares[i].length, history->squares[i].perimeter,
history->squares[i].area);
                printf("   |______________________|__________________|__________________|____________________|\n");
            }

            printf("   |         Mean         | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", means[0], means[1], means[2]);
            printf("   |______________________|__________________|__________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", stds[0], stds[1], stds[2]);
            printf("   |______________________|__________________|__________________|____________________|\n");
        }

        break;

    case Circle:
        if (history->count[2] == 0)
        {
            printf("\n============================================================================\n");
            printf("================= The circle calculation history is empty. =================\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Circle\n");
            printf("    _________________________________________________________________________________\n");
            printf("   |         Index        |      Radius      |   Circumference  |        Area        |\n");
            printf("   |______________________|__________________|__________________|____________________|\n");

            for (i = 0; i < history->count[2]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", i + 1, history->circles[i].radius, history->circles[i].circumference,
history->squares[i].area);
                printf("   |______________________|__________________|__________________|____________________|\n");
            }

            printf("   |         Mean         | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", means[0], means[1], means[2]);
            printf("   |______________________|__________________|__________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m   | %12.2g m^2   |\n", stds[0], stds[1], stds[2]);
            printf("   |______________________|__________________|__________________|____________________|\n");
        }

        break;

    case Cuboid:
        if (history->count[3] == 0)
        {
            printf("\n============================================================================\n");
            printf("================= The cuboid calculation history is empty. =================\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Cuboid\n");
            printf("    _________________________________________________________________________________________________________________________\n");
            printf("   |         Index        |       Width      |       Lenth      |      Height      |    Surface Area    |       Volume       |\n");
            printf("   |______________________|__________________|__________________|__________________|____________________|____________________|\n");

            for (i = 0; i < history->count[3]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m   | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", i + 1, history->cuboids[i].width, history-
>cuboids[i].length, history->cuboids[i].height, history->cuboids[i].area, history->cuboids[i].volume);
                printf("   |______________________|__________________|__________________|__________________|____________________|____________________|\n");
            }

            printf("   |         Mean          | %12.2g m   | %12.2g m   | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", means[0], means[1], means[2], means[3],
means[4]);
            printf("   |______________________|__________________|__________________|__________________|____________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m   | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", stds[0], stds[1], stds[2], stds[3],
stds[4]);
            printf("   |______________________|__________________|__________________|__________________|____________________|____________________|\n");
        }

        break;

    case Cube:
        if (history->count[4] == 0)
        {
            printf("\n============================================================================\n");
            printf("================== The cube calculation history is empty. ==================\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Cube\n");
            printf("    ___________________________________________________________________________________\n");
            printf("   |         Index        |    Side Length   |    Surface Area    |       Volume       |\n");
            printf("   |______________________|__________________|____________________|____________________|\n");

            for (i = 0; i < history->count[4]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", i + 1, history->cubes[i].length, history->cubes[i].area, history-
>cubes[i].volume);
                printf("   |______________________|__________________|____________________|____________________|\n");
            }

            printf("   |         Mean          | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", means[0], means[1], means[2]);
            printf("   |______________________|__________________|____________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", stds[0], stds[1], stds[2]);
            printf("   |______________________|__________________|____________________|____________________|\n");
        }

        break;

    case Sphere:
        if (history->count[5] == 0)
        {
            printf("\n============================================================================\n");
            printf("================= The sphere calculation history is empty. =================\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Sphere\n");
            printf("    ___________________________________________________________________________________\n");
            printf("   |         Index        |      Radius      |    Surface Area    |       Volume       |\n");
            printf("   |______________________|__________________|____________________|____________________|\n");

            for (i = 0; i < history->count[5]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", i + 1, history->spheres[i].radius, history->spheres[i].area,
history->spheres[i].volume);
                printf("   |______________________|__________________|____________________|____________________|\n");
            }

            printf("   |         Mean          | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", means[0], means[1], means[2]);
            printf("   |______________________|__________________|____________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", stds[0], stds[1], stds[2]);
            printf("   |______________________|__________________|____________________|____________________|\n");
        }

        break;

    case Cone:
        if (history->count[6] == 0)
        {
            printf("\n============================================================================\n");
            printf("================== The cone calculation history is empty. ==================\n");
            printf("============================================================================\n");
        }
        else
        {
            printf("\nCalculation Histroy of Cone\n");
            printf("    ______________________________________________________________________________________________________\n");
            printf("   |         Index        |      Radius      |      Height      |    Surface Area    |       Volume       |\n");
            printf("   |______________________|__________________|__________________|____________________|____________________|\n");

            for (i = 0; i < history->count[6]; i++)
            {
                printf("   | %11d          | %12.2g m   | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", i + 1, history->cones[i].radius, history-
>cones[i].height, history->cones[i].area, history->cones[i].volume);
                printf("   |______________________|__________________|__________________|____________________|____________________|\n");
            }

            printf("   |         Mean          | %12.2g m   | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", means[0], means[1], means[2], means[3]);
            printf("   |______________________|__________________|__________________|____________________|____________________|\n");
            printf("   |  Standard Deviation  | %12.2g m   | %12.2g m   | %12.2g m^2   | %12.2g m^3   |\n", stds[0], stds[1], stds[2], stds[3]);
            printf("   |______________________|__________________|__________________|____________________|____________________|\n");
        }
        break;
    }
}

#endif
```

```
#ifndef STRUCT
#define STRUCT

struct Rectangle
{
    double width;
    double length;

    double perimeter;
    double area;
};

struct Square
{
    double length;

    double perimeter;
    double area;
};

struct Circle
{
    double radius;
    double circumference;

    double area;
};

struct Cuboid
{
    double width;
    double length;
    double height;

    double area;
    double volume;
};

struct Cube
{
    double length;

    double area;
    double volume;
};

struct Sphere
{
    double radius;

    double area;
    double volume;
};

struct Cone
{
    double radius;
    double height;

    double area;
    double volume;
};

struct History
{
    int count[7];
    struct Rectangle rectangles[10];
    struct Square squares[10];
    struct Circle circles[10];
    struct Cuboid cuboids[10];
    struct Cube cubes[10];
    struct Sphere spheres[10];
    struct Cone cones[10];
};

#endif
```

```c
#ifndef ENUM
#define ENUM

enum shape
{
    Rectangle,
    Square,
    Circle,
    Cuboid,
    Cube,
    Sphere,
    Cone
};

enum unit
{
    m,
    dm,
    cm,
    mm
};

#endif
```