```
// NOTE: Github repository with project file and libraries to run code included:
https://github.com/Huberti248/AnnualBalance
/*
Resigned:
- Allow to copy from table?
- Speed up launch time?
- Fix artefacts which happen from time to time?
- Fix indistinct comma in inputs?
*/
#include <SDL.h>
#include <SDL_image.h>
#include <SDL_mixer.h>
#include <SDL_net.h>
#include <SDL_ttf.h>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>
#include <random>
//#include <SDL_gpu.h>
//#include <SFML/Network.hpp>
//#include <SFML/Graphics.hpp>
#include <algorithm>
#include <atomic>
#include <codecvt>
#include <functional>
#include <locale>
#include <mutex>
#include <thread>
#include <imgui.h>
#include <imgui_impl_sdl.h>
#include <imgui_impl_sdlrenderer.h>
#include <misc/cpp/imgui_stdlib.h>
#ifdef __ANDROID__
#include "vendor/PUGIXML/src/pugixml.hpp"
#include <android/log.h> //__android_log_print(ANDROID_LOG_VERBOSE,
"SacewiczTabliceJednowymiarowe", "Example number log: %d", number);
#include <jni.h>
#include "vendor/GLM/include/glm/glm.hpp"
#include "vendor/GLM/include/glm/gtc/matrix_transform.hpp"
#include "vendor/GLM/include/glm/gtc/type_ptr.hpp"
#else
#include <filesystem>
#include <pugixml.hpp>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#ifdef __EMSCRIPTEN__
namespace fs = std::__fs::filesystem;
#else
```

```
namespace fs = std::filesystem;
#endif
using namespace std::chrono_literals;
#endif
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#include <emscripten/html5.h>
#endif

// 240 x 240 (smart watch)
// 240 x 320 (QVGA)
// 360 x 640 (Galaxy S5)
// 640 x 480 (480i - Smallest PC monitor)

#define ROW_COUNT 12
#define COLUMN_COUNT 6
#define INCOME_COLUMN 1
#define COSTS_COLUMN 2
#define REVENUE_COLUMN 3
#define PROFIT_LOSS_AND_ZERO_COLUMN 4
#define SUMMARY_COLUMN 5

#define i8 int8_t
#define i16 int16_t
#define i32 int32_t
#define i64 int64_t
#define u8 uint8_t
#define u16 uint16_t
#define u32 uint32_t
#define u64 uint64_t

int windowWidth = 640;
int windowHeight = 480;
SDL_Point mousePos;
SDL_Point realMousePos;
bool keys[SDL_NUM_SCANCODES];
bool buttons[SDL_BUTTON_X2 + 1];
SDL_Window* window;
SDL_Renderer* renderer;

void logOutputCallback(void* userdata, int category, SDL_LogPriority priority,
const char* message)
{
    std::cout << message << std::endl;
}

int random(int min, int max)
{
    return min + rand() % ((max + 1) - min);
}

int SDL_QueryTextureF(SDL_Texture* texture, Uint32* format, int* access, float* w,
float* h)
{
    int wi, hi;
    int result = SDL_QueryTexture(texture, format, access, &wi, &hi);
    if (w) {
        *w = wi;
    }
```

```
    if (h) {
        *h = hi;
    }
    return result;
}

SDL_bool SDL_PointInFRect(const SDL_Point* p, const SDL_FRect* r)
{
    return ((p->x >= r->x) && (p->x < (r->x + r->w)) && (p->y >= r->y) && (p->y <
(r->y + r->h))) ? SDL_TRUE : SDL_FALSE;
}

std::ostream& operator<<(std::ostream& os, SDL_FRect r)
{
    os << r.x << " " << r.y << " " << r.w << " " << r.h;
    return os;
}

std::ostream& operator<<(std::ostream& os, SDL_Rect r)
{
    SDL_FRect fR;
    fR.w = r.w;
    fR.h = r.h;
    fR.x = r.x;
    fR.y = r.y;
    os << fR;
    return os;
}

SDL_Texture* renderText(SDL_Texture* previousTexture, TTF_Font* font, SDL_Renderer*
renderer, const std::string& text, SDL_Color color)
{
    if (previousTexture) {
        SDL_DestroyTexture(previousTexture);
    }
    SDL_Surface* surface;
    if (text.empty()) {
        surface = TTF_RenderUTF8_Blended(font, " ", color);
    }
    else {
        surface = TTF_RenderUTF8_Blended(font, text.c_str(), color);
    }
    if (surface) {
        SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
        SDL_FreeSurface(surface);
        return texture;
    }
    else {
        return 0;
    }
}

struct Text {
    std::string text;
    SDL_Surface* surface = 0;
    SDL_Texture* t = 0;
    SDL_FRect dstR{};
    bool autoAdjustW = false;
    bool autoAdjustH = false;
```

```
    float wMultiplier = 1;
    float hMultiplier = 1;

    ~Text()
    {
        if (surface) {
            SDL_FreeSurface(surface);
        }
        if (t) {
            SDL_DestroyTexture(t);
        }
    }

    Text()
    {
    }

    Text(const Text& rightText)
    {
        text = rightText.text;
        if (surface) {
            SDL_FreeSurface(surface);
        }
        if (t) {
            SDL_DestroyTexture(t);
        }
        if (rightText.surface) {
            surface = SDL_ConvertSurface(rightText.surface, rightText.surface-
>format, SDL_SWSURFACE);
        }
        if (rightText.t) {
            t = SDL_CreateTextureFromSurface(renderer, surface);
        }
        dstR = rightText.dstR;
        autoAdjustW = rightText.autoAdjustW;
        autoAdjustH = rightText.autoAdjustH;
        wMultiplier = rightText.wMultiplier;
        hMultiplier = rightText.hMultiplier;
    }

    Text& operator=(const Text& rightText)
    {
        text = rightText.text;
        if (surface) {
            SDL_FreeSurface(surface);
        }
        if (t) {
            SDL_DestroyTexture(t);
        }
        if (rightText.surface) {
            surface = SDL_ConvertSurface(rightText.surface, rightText.surface-
>format, SDL_SWSURFACE);
        }
        if (rightText.t) {
            t = SDL_CreateTextureFromSurface(renderer, surface);
        }
        dstR = rightText.dstR;
        autoAdjustW = rightText.autoAdjustW;
        autoAdjustH = rightText.autoAdjustH;
```

```
        wMultiplier = rightText.wMultiplier;
        hMultiplier = rightText.hMultiplier;
        return *this;
    }

    void setText(SDL_Renderer* renderer, TTF_Font* font, std::string text,
SDL_Color c = { 255, 255, 255 })
    {
        this->text = text;
#if 1 // NOTE: renderText
        if (surface) {
            SDL_FreeSurface(surface);
        }
        if (t) {
            SDL_DestroyTexture(t);
        }
        if (text.empty()) {
            surface = TTF_RenderUTF8_Blended(font, " ", c);
        }
        else {
            surface = TTF_RenderUTF8_Blended(font, text.c_str(), c);
        }
        if (surface) {
            t = SDL_CreateTextureFromSurface(renderer, surface);
        }
#endif
        if (autoAdjustW) {
            SDL_QueryTextureF(t, 0, 0, &dstR.w, 0);
        }
        if (autoAdjustH) {
            SDL_QueryTextureF(t, 0, 0, 0, &dstR.h);
        }
        dstR.w *= wMultiplier;
        dstR.h *= hMultiplier;
    }

    void setText(SDL_Renderer* renderer, TTF_Font* font, int value, SDL_Color c = {
255, 255, 255 })
    {
        setText(renderer, font, std::to_string(value), c);
    }

    void draw(SDL_Renderer* renderer)
    {
        if (t) {
            SDL_RenderCopyF(renderer, t, 0, &dstR);
        }
    }
};

int SDL_RenderDrawCircle(SDL_Renderer* renderer, int x, int y, int radius)
{
    int offsetx, offsety, d;
    int status;

    offsetx = 0;
    offsety = radius;
    d = radius - 1;
    status = 0;
```

```
    while (offsety >= offsetx) {
        status += SDL_RenderDrawPoint(renderer, x + offsetx, y + offsety);
        status += SDL_RenderDrawPoint(renderer, x + offsety, y + offsetx);
        status += SDL_RenderDrawPoint(renderer, x - offsetx, y + offsety);
        status += SDL_RenderDrawPoint(renderer, x - offsety, y + offsetx);
        status += SDL_RenderDrawPoint(renderer, x + offsetx, y - offsety);
        status += SDL_RenderDrawPoint(renderer, x + offsety, y - offsetx);
        status += SDL_RenderDrawPoint(renderer, x - offsetx, y - offsety);
        status += SDL_RenderDrawPoint(renderer, x - offsety, y - offsetx);

        if (status < 0) {
            status = -1;
            break;
        }

        if (d >= 2 * offsetx) {
            d -= 2 * offsetx + 1;
            offsetx += 1;
        }
        else if (d < 2 * (radius - offsety)) {
            d += 2 * offsety - 1;
            offsety -= 1;
        }
        else {
            d += 2 * (offsety - offsetx - 1);
            offsety -= 1;
            offsetx += 1;
        }
    }

    return status;
}

int SDL_RenderFillCircle(SDL_Renderer* renderer, int x, int y, int radius)
{
    int offsetx, offsety, d;
    int status;

    offsetx = 0;
    offsety = radius;
    d = radius - 1;
    status = 0;

    while (offsety >= offsetx) {

        status += SDL_RenderDrawLine(renderer, x - offsety, y + offsetx,
            x + offsety, y + offsetx);
        status += SDL_RenderDrawLine(renderer, x - offsetx, y + offsety,
            x + offsetx, y + offsety);
        status += SDL_RenderDrawLine(renderer, x - offsetx, y - offsety,
            x + offsetx, y - offsety);
        status += SDL_RenderDrawLine(renderer, x - offsety, y - offsetx,
            x + offsety, y - offsetx);

        if (status < 0) {
            status = -1;
            break;
        }
```

```
        if (d >= 2 * offsetx) {
            d -= 2 * offsetx + 1;
            offsetx += 1;
        }
        else if (d < 2 * (radius - offsety)) {
            d += 2 * offsety - 1;
            offsety -= 1;
        }
        else {
            d += 2 * (offsety - offsetx - 1);
            offsety -= 1;
            offsetx += 1;
        }
    }

    return status;
}

struct Clock {
    Uint64 start = SDL_GetPerformanceCounter();

    float getElapsedTime()
    {
        Uint64 stop = SDL_GetPerformanceCounter();
        float secondsElapsed = (stop - start) /
(float)SDL_GetPerformanceFrequency();
        return secondsElapsed * 1000;
    }

    float restart()
    {
        Uint64 stop = SDL_GetPerformanceCounter();
        float secondsElapsed = (stop - start) /
(float)SDL_GetPerformanceFrequency();
        start = SDL_GetPerformanceCounter();
        return secondsElapsed * 1000;
    }
};

SDL_bool SDL_FRectEmpty(const SDL_FRect* r)
{
    return ((!r) || (r->w <= 0) || (r->h <= 0)) ? SDL_TRUE : SDL_FALSE;
}

SDL_bool SDL_IntersectFRect(const SDL_FRect* A, const SDL_FRect* B, SDL_FRect*
result)
{
    int Amin, Amax, Bmin, Bmax;

    if (!A) {
        SDL_InvalidParamError("A");
        return SDL_FALSE;
    }

    if (!B) {
        SDL_InvalidParamError("B");
        return SDL_FALSE;
    }
```

```
    if (!result) {
        SDL_InvalidParamError("result");
        return SDL_FALSE;
    }

    /* Special cases for empty rects */
    if (SDL_FRectEmpty(A) || SDL_FRectEmpty(B)) {
        result->w = 0;
        result->h = 0;
        return SDL_FALSE;
    }

    /* Horizontal intersection */
    Amin = A->x;
    Amax = Amin + A->w;
    Bmin = B->x;
    Bmax = Bmin + B->w;
    if (Bmin > Amin)
        Amin = Bmin;
    result->x = Amin;
    if (Bmax < Amax)
        Amax = Bmax;
    result->w = Amax - Amin;

    /* Vertical intersection */
    Amin = A->y;
    Amax = Amin + A->h;
    Bmin = B->y;
    Bmax = Bmin + B->h;
    if (Bmin > Amin)
        Amin = Bmin;
    result->y = Amin;
    if (Bmax < Amax)
        Amax = Bmax;
    result->h = Amax - Amin;

    return (SDL_FRectEmpty(result) == SDL_TRUE) ? SDL_FALSE : SDL_TRUE;
}

SDL_bool SDL_HasIntersectionF(const SDL_FRect* A, const SDL_FRect* B)
{
    float Amin, Amax, Bmin, Bmax;

    if (!A) {
        SDL_InvalidParamError("A");
        return SDL_FALSE;
    }

    if (!B) {
        SDL_InvalidParamError("B");
        return SDL_FALSE;
    }

    /* Special cases for empty rects */
    if (SDL_FRectEmpty(A) || SDL_FRectEmpty(B)) {
        return SDL_FALSE;
    }
```

```
    /* Horizontal intersection */
    Amin = A->x;
    Amax = Amin + A->w;
    Bmin = B->x;
    Bmax = Bmin + B->w;
    if (Bmin > Amin)
        Amin = Bmin;
    if (Bmax < Amax)
        Amax = Bmax;
    if (Amax <= Amin)
        return SDL_FALSE;

    /* Vertical intersection */
    Amin = A->y;
    Amax = Amin + A->h;
    Bmin = B->y;
    Bmax = Bmin + B->h;
    if (Bmin > Amin)
        Amin = Bmin;
    if (Bmax < Amax)
        Amax = Bmax;
    if (Amax <= Amin)
        return SDL_FALSE;

    return SDL_TRUE;
}

float clamp(float n, float lower, float upper)
{
    return std::max(lower, std::min(n, upper));
}

std::string getCurrentDate()
{
    std::time_t t = std::time(0);
    std::tm* now = std::localtime(&t);
    std::stringstream ss;
    ss << (now->tm_year + 1900) << "-" << (now->tm_mon + 1) << "-" << now->tm_mday;
    return ss.str();
}

std::string readWholeFile(std::string path)
{
    std::stringstream ss;
    std::ifstream ifs(path, std::ifstream::in);
    ss << ifs.rdbuf();
    return ss.str();
}

void saveToFile(std::string path, std::string str)
{
    std::stringstream ss;
    ss << str;
    std::ofstream ofs(path);
    ofs << ss.str();
}

int eventWatch(void* userdata, SDL_Event* event)
{
```

```
    // WARNING: Be very careful of what you do in the function, as it may run in a
different thread
    if (event->type == SDL_APP_TERMINATING || event->type ==
SDL_APP_WILLENTERBACKGROUND) {
    }
    return 0;
}

float incomeSum(std::vector<std::string> numbers)
{
    float sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        int row = i % ROW_COUNT;
        int column = i / ROW_COUNT;
        if (column == INCOME_COLUMN) {
            try {
                sum += std::stof(numbers[i].empty() ? std::string("0") :
numbers[i]);
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    return sum;
}

float costsSum(std::vector<std::string> numbers)
{
    float sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        int row = i % ROW_COUNT;
        int column = i / ROW_COUNT;
        if (column == COSTS_COLUMN) {
            try {
                sum += std::stof(numbers[i].empty() ? std::string("0") :
numbers[i]);
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    return sum;
}

float revenueSum(std::vector<std::string> numbers)
{
    float sum = 0;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int revenueIndex = i + (REVENUE_COLUMN * ROW_COUNT);
        try {
            sum += std::stof(numbers[revenueIndex].empty() ? "0" :
numbers[revenueIndex]);
        }
        catch (std::invalid_argument e) {
        }
    }
    return sum;
}
```

```
struct Result {
    float value = 0;
    int monthNumber = 0;
};

Result minIncome(std::vector<std::string> numbers)
{
    int notEmptyIncomeCellIndex = -1;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int incomeIndex = i + (INCOME_COLUMN * ROW_COUNT);
        if (!numbers[incomeIndex].empty()) {
            try {
                std::stof(numbers[incomeIndex]);
                notEmptyIncomeCellIndex = incomeIndex;
                break;
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    if (notEmptyIncomeCellIndex == -1) {
        return { 0, -1 };
    }
    else {
        float min = std::stof(numbers[notEmptyIncomeCellIndex]);
        int index = (notEmptyIncomeCellIndex % ROW_COUNT) + 1;
        for (int i = 1; i < ROW_COUNT; ++i) {
            int incomeIndex = i + (INCOME_COLUMN * ROW_COUNT);
            if (!numbers[incomeIndex].empty()) {
                try {
                    if (std::stof(numbers[incomeIndex]) < min) {
                        min = std::stof(numbers[incomeIndex]);
                        index = (incomeIndex % ROW_COUNT) + 1;
                    }
                }
                catch (std::invalid_argument e) {
                }
            }
        }
        return { min, index };
    }
}

Result maxIncome(std::vector<std::string> numbers)
{
    int notEmptyIncomeCellIndex = -1;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int incomeIndex = i + (INCOME_COLUMN * ROW_COUNT);
        if (!numbers[incomeIndex].empty()) {
            try {
                std::stof(numbers[incomeIndex]);
                notEmptyIncomeCellIndex = incomeIndex;
                break;
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    if (notEmptyIncomeCellIndex == -1) {
```

```
        return { 0, -1 };
    }
    else {
        float max = std::stof(numbers[notEmptyIncomeCellIndex]);
        int index = (notEmptyIncomeCellIndex % ROW_COUNT) + 1;
        for (int i = 1; i < ROW_COUNT; ++i) {
            int incomeIndex = i + (INCOME_COLUMN * ROW_COUNT);
            if (!numbers[incomeIndex].empty()) {
                try {
                    if (std::stof(numbers[incomeIndex]) > max) {
                        max = std::stof(numbers[incomeIndex]);
                        index = (incomeIndex % ROW_COUNT) + 1;
                    }
                }
                catch (std::invalid_argument e) {
                }
            }
        }
        return { max, index };
    }
}

float averageIncome(std::vector<std::string> numbers)
{
    int notEmptyIncomeCellsCount = 0;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int incomeIndex = i + (INCOME_COLUMN * ROW_COUNT);
        if (!numbers[incomeIndex].empty()) {
            ++notEmptyIncomeCellsCount;
        }
    }
    if (notEmptyIncomeCellsCount == 0) {
        return 0;
    }
    else {
        return incomeSum(numbers) / notEmptyIncomeCellsCount;
    }
}

Result minCost(std::vector<std::string> numbers)
{
    int notEmptyCostsCellIndex = -1;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int costIndex = i + (COSTS_COLUMN * ROW_COUNT);
        if (!numbers[costIndex].empty()) {
            try {
                std::stof(numbers[costIndex]);
                notEmptyCostsCellIndex = costIndex;
                break;
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    if (notEmptyCostsCellIndex == -1) {
        return { 0, -1 };
    }
    else {
        float min = std::stof(numbers[notEmptyCostsCellIndex]);
```

```
        int index = (notEmptyCostsCellIndex % ROW_COUNT) + 1;
        for (int i = 1; i < ROW_COUNT; ++i) {
            int costIndex = i + (COSTS_COLUMN * ROW_COUNT);
            if (!numbers[costIndex].empty()) {
                try {
                    if (std::stof(numbers[costIndex]) < min) {
                        min = std::stof(numbers[costIndex]);
                        index = (costIndex % ROW_COUNT) + 1;
                    }
                }
                catch (std::invalid_argument e) {
                }
            }
        }
        return { min, index };
    }
}

Result maxCost(std::vector<std::string> numbers)
{
    int notEmptyCostCellIndex = -1;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int costIndex = i + (COSTS_COLUMN * ROW_COUNT);
        if (!numbers[costIndex].empty()) {
            try {
                std::stof(numbers[costIndex]);
                notEmptyCostCellIndex = costIndex;
                break;
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    if (notEmptyCostCellIndex == -1) {
        return { 0, -1 };
    }
    else {
        float max = std::stof(numbers[notEmptyCostCellIndex]);
        int index = (notEmptyCostCellIndex % ROW_COUNT) + 1;
        for (int i = 1; i < ROW_COUNT; ++i) {
            int costIndex = i + (COSTS_COLUMN * ROW_COUNT);
            if (!numbers[costIndex].empty()) {
                try {
                    if (std::stof(numbers[costIndex]) > max) {
                        max = std::stof(numbers[costIndex]);
                        index = (costIndex % ROW_COUNT) + 1;
                    }
                }
                catch (std::invalid_argument e) {
                }
            }
        }
        return { max, index };
    }
}

float averageCosts(std::vector<std::string> numbers)
{
    int notEmptyCostsCellsCount = 0;
```

```
    for (int i = 0; i < ROW_COUNT; ++i) {
        int costIndex = i + (COSTS_COLUMN * ROW_COUNT);
        if (!numbers[costIndex].empty()) {
            ++notEmptyCostsCellsCount;
        }
    }
    if (notEmptyCostsCellsCount == 0) {
        return 0;
    }
    else {
        return costsSum(numbers) / notEmptyCostsCellsCount;
    }
}

Result minRevenue(std::vector<std::string> numbers)
{
    int notEmptyRevenueCellIndex = -1;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int revenueIndex = i + (REVENUE_COLUMN * ROW_COUNT);
        if (!numbers[revenueIndex].empty()) {
            try {
                std::stof(numbers[revenueIndex]);
                notEmptyRevenueCellIndex = revenueIndex;
                break;
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    if (notEmptyRevenueCellIndex == -1) {
        return { 0, -1 };
    }
    else {
        float min = std::stof(numbers[notEmptyRevenueCellIndex]);
        int index = (notEmptyRevenueCellIndex % ROW_COUNT) + 1;
        for (int i = 1; i < ROW_COUNT; ++i) {
            int revenueIndex = i + (REVENUE_COLUMN * ROW_COUNT);
            if (!numbers[revenueIndex].empty()) {
                try {
                    if (std::stof(numbers[revenueIndex]) < min) {
                        min = std::stof(numbers[revenueIndex]);
                        index = (revenueIndex % ROW_COUNT) + 1;
                    }
                }
                catch (std::invalid_argument e) {
                }
            }
        }
        return { min, index };
    }
}

Result maxRevenue(std::vector<std::string> numbers)
{
    int notEmptyRevenueCellIndex = -1;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int revenueIndex = i + (REVENUE_COLUMN * ROW_COUNT);
        if (!numbers[revenueIndex].empty()) {
            try {
```

```
                std::stof(numbers[revenueIndex]);
                notEmptyRevenueCellIndex = revenueIndex;
                break;
            }
            catch (std::invalid_argument e) {
            }
        }
    }
    if (notEmptyRevenueCellIndex == -1) {
        return { 0, -1 };
    }
    else {
        float max = std::stof(numbers[notEmptyRevenueCellIndex]);
        int index = (notEmptyRevenueCellIndex % ROW_COUNT) + 1;
        for (int i = 1; i < ROW_COUNT; ++i) {
            int revenueIndex = i + (REVENUE_COLUMN * ROW_COUNT);
            if (!numbers[revenueIndex].empty()) {
                try {
                    if (std::stof(numbers[revenueIndex]) > max) {
                        max = std::stof(numbers[revenueIndex]);
                        index = (revenueIndex % ROW_COUNT) + 1;
                    }
                }
                catch (std::invalid_argument e) {
                }
            }
        }
        return { max, index };
    }
}

float averageRevenue(std::vector<std::string> numbers)
{
    int notEmptyRevenueCellsCount = 0;
    for (int i = 0; i < ROW_COUNT; ++i) {
        int revenueIndex = i + (REVENUE_COLUMN * ROW_COUNT);
        if (!numbers[revenueIndex].empty()) {
            ++notEmptyRevenueCellsCount;
        }
    }
    if (notEmptyRevenueCellsCount == 0) {
        return 0;
    }
    else {
        return revenueSum(numbers) / notEmptyRevenueCellsCount;
    }
}

template <class charT>
struct no_separator : public std::numpunct_byname<charT> {
    explicit no_separator(const char* name, size_t refs = 0)
        : std::numpunct_byname<charT>(name, refs)
    {
    }

protected:
    virtual string_type do_grouping() const
    {
        return "\000";
```

```
    }
};

std::string toStringWithTwoDecimalPlaces(float number)
{
    std::stringstream ss;
    ss.setf(std::ios::fixed);
    ss.precision(2);
    std::locale loc("pl");
    ss.imbue(std::locale(std::locale(""), new
no_separator<char>("German_germany")));
    ss << number;
    return ss.str();
}

int main(int argc, char* argv[])
{
    std::locale::global(std::locale("pl"));
    std::srand(std::time(0));
    SDL_LogSetAllPriority(SDL_LOG_PRIORITY_VERBOSE);
    SDL_LogSetOutputFunction(logOutputCallback, 0);
    SDL_Init(SDL_INIT_EVERYTHING);
    TTF_Init();
    SDL_GetMouseState(&mousePos.x, &mousePos.y);
    window = SDL_CreateWindow("RocznyBilans", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, windowWidth, windowHeight, SDL_WINDOW_RESIZABLE |
SDL_WINDOW_MAXIMIZED);
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
    TTF_Font* robotoF = TTF_OpenFont("res/roboto.ttf", 72);
    int w, h;
    SDL_GetWindowSize(window, &w, &h);
    SDL_RenderSetScale(renderer, w / (float)windowWidth, h / (float)windowHeight);
    SDL_AddEventWatch(eventWatch, 0);
    bool running = true;
    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    ImGuiIO& io = ImGui::GetIO();
    (void)io;
    ImGui::StyleColorsLight();
    ImGui_ImplSDL2_InitForSDLRenderer(window);
    ImGui_ImplSDLRenderer_Init(renderer);
    ImVector<ImWchar> ranges;
    ImFontGlyphRangesBuilder builder;
    builder.AddText(u8"zażółć gęślą jaźńZAŻÓŁĆ GĘŚLĄ JAŹŃ");
    builder.AddRanges(io.Fonts->GetGlyphRangesDefault());
    builder.BuildRanges(&ranges);
    io.Fonts->AddFontFromFileTTF("res/Jura/static/Jura-Regular.ttf", 9, 0,
ranges.Data);
    io.Fonts->Build();
    bool showDemoWindow = false;
    std::vector<std::string> numbers;
    for (int i = 0; i < ROW_COUNT; ++i) {
        for (int j = 0; j < COLUMN_COUNT; ++j) {
            numbers.push_back("");
        }
    }
    while (running) {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
```

```
            ImGui_ImplSDL2_ProcessEvent(&event);
            if (event.type == SDL_QUIT || event.type == SDL_KEYDOWN &&
event.key.keysym.scancode == SDL_SCANCODE_ESCAPE) {
                running = false;
                // NOTE: On mobile remember to use eventWatch function (it doesn't
reach this code when terminating)
            }
            if (event.type == SDL_WINDOWEVENT && event.window.event ==
SDL_WINDOWEVENT_RESIZED) {
                SDL_RenderSetScale(renderer, event.window.data1 /
(float)windowWidth, event.window.data2 / (float)windowHeight);
            }
            if (event.type == SDL_KEYDOWN) {
                keys[event.key.keysym.scancode] = true;
            }
            if (event.type == SDL_KEYUP) {
                keys[event.key.keysym.scancode] = false;
            }
            if (event.type == SDL_MOUSEBUTTONDOWN) {
                buttons[event.button.button] = true;
            }
            if (event.type == SDL_MOUSEBUTTONUP) {
                buttons[event.button.button] = false;
            }
            if (event.type == SDL_MOUSEMOTION) {
                float scaleX, scaleY;
                SDL_RenderGetScale(renderer, &scaleX, &scaleY);
                mousePos.x = event.motion.x / scaleX;
                mousePos.y = event.motion.y / scaleY;
                realMousePos.x = event.motion.x;
                realMousePos.y = event.motion.y;
            }
        }
        ImGui_ImplSDLRenderer_NewFrame();
        ImGui_ImplSDL2_NewFrame(window);
        io.MousePos.x = mousePos.x;
        io.MousePos.y = mousePos.y;
        ImGui::NewFrame();
        if (showDemoWindow) {
            ImGui::ShowDemoWindow(&showDemoWindow);
        }
        static bool displayHeaders = true;
        static bool use_work_area = true;
        static ImGuiTableFlags flags = ImGuiWindowFlags_NoDecoration |
ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize;
        const ImGuiViewport* viewport = ImGui::GetMainViewport();
        ImGui::SetNextWindowPos(use_work_area ? viewport->WorkPos : viewport->Pos);
        ImGui::SetNextWindowSize(use_work_area ? viewport->WorkSize : viewport-
>Size);
        if (ImGui::Begin("window", 0, flags)) {
            static ImGuiTableFlags tableFlags = ImGuiTableFlags_SizingFixedFit |
ImGuiTableFlags_Resizable | ImGuiTableFlags_BordersOuter | ImGuiTableFlags_BordersV
| ImGuiTableFlags_ContextMenuInBody | ImGuiTableFlags_NoHostExtendX;
            if (ImGui::BeginTable("table", COLUMN_COUNT, tableFlags)) {
                if (displayHeaders) {
                    ImGui::TableSetupColumn(u8"Numer miesiąca");
                    ImGui::TableSetupColumn("Przychody");
                    ImGui::TableSetupColumn("Koszty");
                    ImGui::TableSetupColumn(u8"Dochód");
```

```
                    ImGui::TableSetupColumn(u8"Relacja między przychodem a
kosztem");
                    ImGui::TableSetupColumn("Podsumowanie");
                    ImGui::TableHeadersRow();
                }
                for (int i = 0; i < ROW_COUNT; ++i) {
                    ImGui::TableNextRow();
                    for (int j = 0; j < COLUMN_COUNT; ++j) {
                        ImGui::TableSetColumnIndex(j);
                        int index = i + (j * ROW_COUNT);
                        int incomeIndex = i + (INCOME_COLUMN * ROW_COUNT);
                        int costsIndex = i + (COSTS_COLUMN * ROW_COUNT);
                        if (j == 0) {
                            std::string text = std::to_string(i + 1);
                            ImGui::Text("%d", std::stoi(text));
                        }
                        else if (j == INCOME_COLUMN || j == COSTS_COLUMN) {
                            ImGui::PushStyleColor(ImGuiCol_FrameBg, IM_COL32(163,
148, 147, 255));
                            ImGui::SetNextItemWidth(-FLT_MIN);
                            ImGui::InputText(("##" +
std::to_string(index)).c_str(), &numbers[index]);
                            ImGui::PopStyleColor();
                        }
                        if (j == REVENUE_COLUMN) {
                            try {
                                numbers[index] =
toStringWithTwoDecimalPlaces(std::stof(numbers[incomeIndex].empty() ?
std::string("0") : numbers[incomeIndex]) - std::stof(numbers[costsIndex].empty() ?
"0" : numbers[costsIndex]));
                            }
                            catch (std::invalid_argument e) {
                                numbers[index] = "ERROR";
                            }
                            ImGui::Text(numbers[index].c_str());
                        }
                        if (j == PROFIT_LOSS_AND_ZERO_COLUMN) {
                            std::string out;
                            try {
                                float revenue =
std::stof(numbers[incomeIndex].empty() ? std::string("0") : numbers[incomeIndex]) -
std::stof(numbers[costsIndex].empty() ? "0" : numbers[costsIndex]);
                                if (revenue > 0) {
                                    out = "zysk";
                                }
                                else if (revenue < 0) {
                                    out = "strata";
                                }
                                else {
                                    out = "zero";
                                }
                            }
                            catch (std::invalid_argument e) {
                                out = "ERROR";
                            }
                            ImGui::Text(out.c_str());
                        }
                        if (i == 0 && j == SUMMARY_COLUMN) {
                            ImGui::Text((u8"Suma przychodów to " +
```

```
toStringWithTwoDecimalPlaces(incomeSum(numbers))).c_str());
                        }
                        if (i == 1 && j == SUMMARY_COLUMN) {
                            Result result = minIncome(numbers);
                            ImGui::Text((u8"Minimalny przychód to " +
toStringWithTwoDecimalPlaces(result.value) + u8" w miesiącu " +
std::to_string(result.monthNumber)).c_str());
                        }
                        if (i == 2 && j == SUMMARY_COLUMN) {
                            Result result = maxIncome(numbers);
                            ImGui::Text((u8"Maksymalny przychód to " +
toStringWithTwoDecimalPlaces(result.value) + u8" w miesiącu " +
std::to_string(result.monthNumber)).c_str());
                        }
                        if (i == 3 && j == SUMMARY_COLUMN) {
                            ImGui::Text((u8"Średni przychód to " +
toStringWithTwoDecimalPlaces(averageIncome(numbers))).c_str());
                        }
                        if (i == 4 && j == SUMMARY_COLUMN) {
                            ImGui::Text((u8"Suma kosztów to " +
toStringWithTwoDecimalPlaces(costsSum(numbers))).c_str());
                        }
                        if (i == 5 && j == SUMMARY_COLUMN) {
                            Result result = minCost(numbers);
                            ImGui::Text((u8"Minimalny koszt to " +
toStringWithTwoDecimalPlaces(result.value) + u8" w miesiącu " +
std::to_string(result.monthNumber)).c_str());
                        }
                        if (i == 6 && j == SUMMARY_COLUMN) {
                            Result result = maxCost(numbers);
                            ImGui::Text((u8"Maksymalny koszt to " +
toStringWithTwoDecimalPlaces(result.value) + u8" w miesiącu " +
std::to_string(result.monthNumber)).c_str());
                        }
                        if (i == 7 && j == SUMMARY_COLUMN) {
                            ImGui::Text((u8"Średni koszt to " +
toStringWithTwoDecimalPlaces(averageCosts(numbers))).c_str());
                        }
                        if (i == 8 && j == SUMMARY_COLUMN) {
                            ImGui::Text((u8"Suma dochodów to " +
toStringWithTwoDecimalPlaces(revenueSum(numbers))).c_str());
                        }
                        if (i == 9 && j == SUMMARY_COLUMN) {
                            Result result = minRevenue(numbers);
                            ImGui::Text((u8"Minimalny dochód to " +
toStringWithTwoDecimalPlaces(result.value) + u8" w miesiącu " +
std::to_string(result.monthNumber)).c_str());
                        }
                        if (i == 10 && j == SUMMARY_COLUMN) {
                            Result result = maxRevenue(numbers);
                            ImGui::Text((u8"Maksymalny dochód to " +
toStringWithTwoDecimalPlaces(result.value) + u8" w miesiącu " +
std::to_string(result.monthNumber)).c_str());
                        }
                        if (i == 11 && j == SUMMARY_COLUMN) {
                            ImGui::Text((u8"Średni dochód to " +
toStringWithTwoDecimalPlaces(averageRevenue(numbers))).c_str());
                        }
                    }
```

```
                }
                ImGui::EndTable();
            }
        }
        ImGui::End();
        ImGui::Render();
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
        SDL_RenderClear(renderer);
        ImGui_ImplSDLRenderer_RenderDrawData(ImGui::GetDrawData());
        SDL_RenderPresent(renderer);
    }
    // NOTE: On mobile remember to use eventWatch function (it doesn't reach this
code when terminating)
    return 0;
}
```