Arrays and Sprites

1. Contents

Arrays and Sprites 1
1. Contents 1
2. Learning aims 2
3. Exercises 2
3.1. ArrayBasics 3
3.1.1. Defining arrays 3
3.1.2. Accessing elements 3
3.1.3. Exceeding the array boundaries 3
3.1.4. Size of arrays 4
3.1.5. Organization of the elements in memory 4
3.1.6. Passing an array to a function 5
3.1.7. Using arrays to hold 2D values 5
3.2. BasicAlgorithms 6
3.2.1. Create the project 6
3.2.2. Objective 6
3.2.3. Prepare test data 7
3.2.4. Count function 7
3.2.5. MinElement/ MaxElement functions 7
3.2.6. Swap function 7
3.2.7. Shuffle function 8
3.2.8. BubbleSort function 9
3.3. ArrayApplications 10
3.3.1. Create the project 10
3.3.2. Objective 11
3.3.3. Click polygon 11
3.3.4. Rotating pentagrams 12
3.3.5. Statistics of the rand() function 12
3.3.6. Mouse tracing 14
3.3.7. Select cells in grid 14
3.4. ShuffleCards 15
3.4.1. Create the project 15
3.4.2. Objective 15
3.4.3. Analyze the problem 15

3.4.4. Create/delete the Textures 16
3.4.5. Draw the Textures 16
3.4.6. Shuffle the textures 16
3.5. OxoGame 16
3.5.1. Create the project 16
3.5.2. Objective 17
3.6. Sprites 17
3.6.1. Create the project 17
3.6.2. Objective 18
3.6.3. Sprite struct 18
3.6.4. Knight sprite 18
3.6.5. Animated Tibo 20
4. Submission instructions 20
5. References 21
5.1. Arrays 21
5.1.1. Arrays part 1 21
5.1.2. Arrays part 2 21
5.1.3. Arrays and loops 21
5.2. sizeof operator 21
5.3. Bubble sort 21
5.4. Recursion 21
5.5. Sierpinski triangle 21

2. Learning aims

At the end of these exercises you should:

- Be able to define and initialize arrays
- Know how they are organized in memory
- Know how to access their elements using the [] operator
- Know how to declare a function with an array as a parameter
- Know how to use an array to store 2D values
- Know the bubble sorting algorithm
- Know how to draw an animated sprite using Textures.

We advise you to **make your own summary of topics** that are new to you.

3. Exercises

Your name, first name and group should be mentioned at the top of each cpp file.

Give your **projects** the same **name** as mentioned in the title of the exercise, e.g. **ArrayBasics**. Other names will be rejected.

3.1. ArrayBasics

Create a new project with name **ArrayBasics** in your **1DAExx_08_name_firstname** folder.

Add your name and group as comment at the top of the **ArrayBasics.cpp** file.

Copy the **structs.h** and **structs.cpp** in the project's folder and add it to Visual Studio.

In this application, you will make some basic exercises on arrays.

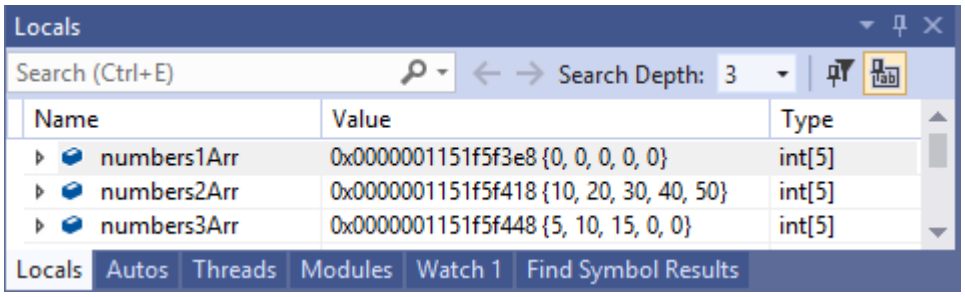
More information about arrays can be found here Arrays

3.1.1. Defining arrays

Define 3 arrays of 5 **int type** elements. Please do not use hard coded magic numbers (literals) for the size (5), use const variables instead.

1. A first array specifying the size and using default initialization.
2. A second array **without** specifying the size but supplying the values using an initialization list.
3. A third array specifying the size and supplying a list of values, the number of values being less than the size.

Have a look at the content and size of these arrays using the Locals window of Visual Studio.



Notice the values of the last elements in the third array.

3.1.2. Accessing elements

Print the value of the first element in one of these arrays using the [] operator.

Print the value of the last element in one of these arrays using the [] operator.

Print all the values of an array using an iteration.

```
--  Accessing  elements --
First element: 5
Last element: 0

Iterating over the elements
5 10 15 0 0
```

3.1.3. Exceeding the array boundaries

Print the elements of the third array hereby exceeding the end boundary. What do you notice? In our example, we accessed the values of the other array!!!

```
Iterating over the elements
5 10 15 0 0 -858993460 -858993460 10 20 30
```

Change (e.g. increment with 1) each element in an array again exceeding the end boundary. What do you notice? In our example, we modified values of the other array!!!

```
Iterating over the elements
6 11 16 1 1 -858993459 -858993459 11 21 31
```

There might be an error message, if yes, what error message did you get? If not, do you understand how dangerous working with arrays can be?

3.1.4. Size of arrays.

Applying the **sizeof** operator on an array returns the total number of bytes of the array (sizeof operator). If you divide this number by the number of bytes of one of its elements, you get the number of elements in the array.

Apply this to second array:

- Get the number of bytes of this array and print it on the console.
- Then get the size of one of its elements, like this `sizeof( numbers2[0])` and print the result on the console
- Then make the division to get the number of elements and print it.

```
--  Size of arrays --
2nd array
  nr of bytes: 20
  size of one element: 4
  nr of elements: 5
```

To be able to print all elements of this second array, this value is needed:

```
--  Size of arrays --
2nd array
  nr of bytes: 20
  size of one element: 4
  nr of elements: 5
11 21 31 40 50
```

3.1.5. Organization of the elements in memory

The elements of an array occupy consecutive memory locations, there are no gaps for other information between them.

Write some code that prints the addresses of the elements of one of these arrays (tip: **&**number2[0] is the address of the first element).

Also define an array of **Point2f** type elements. Print the addresses of these elements as well.

Have a look at the results and notice the difference between:

- the addresses of 2 consecutive elements of the **int** types array
- the addresses of 2 consecutive elements of the **Point2f** types array.

Use the windows calculator in programmer mode to calculate the difference between two consecutive elements.

```
--  The elements occupy consecutive memory locations --
Array of int types
Address of element with index 0 is 0018FCF0
Address of element with index 1 is 0018FCF4
Address of element with index 2 is 0018FCF8
Address of element with index 3 is 0018FCFC
Address of element with index 4 is 0018FD00


Array of Point2f types
Address of element with index 0 is 0018FC5C
Address of element with index 1 is 0018FC64
Address of element with index 2 is 0018FC6C
Address of element with index 3 is 0018FC74
Address of element with index 4 is 0018FC7C
```

3.1.6. Passing an array to a function

- Declare and define a function **PrintElements** that prints the elements of an array of int elements. Besides the array this function needs also the size of the array, so make it a parameter.

```
void PrintElements( int* pNumbers, int size );
```

We always start the name of pointer type variables with p.
Then call this function three times for the three arrays.

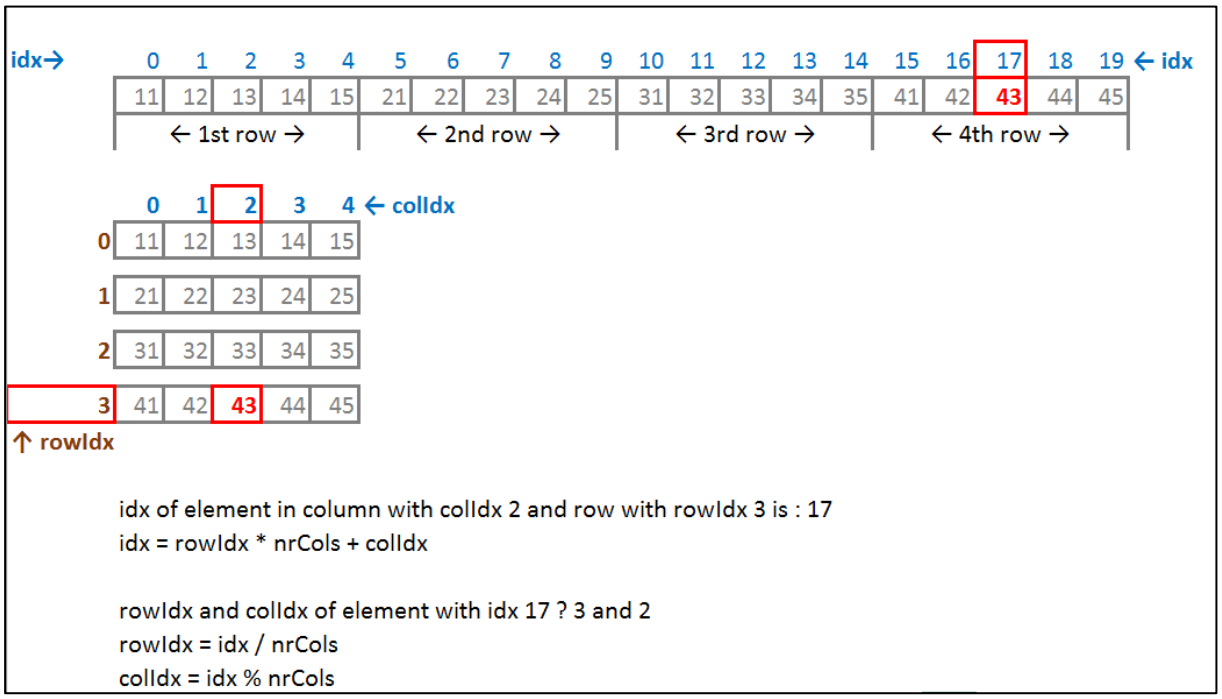
- Make an overloaded function **PrintElements** that prints a range of array elements as specified by a start- and end index parameter.

```
void PrintElements( int* pNumbers, int startIdx, int endIdx );
```

```
--  Passing an array to a function, PrintElements --
Print all elements
0 0 0 0 0
10 20 30 40 50
5 10 15 0 0 0 0 0 0 0

Print a range of elements
20 30 40
```

3.1.7. Using arrays to hold 2D values

Define an array of int types with size 20 that we will use as the elements of e.g. a grid of 4 rows x 5 columns, see following screenshot.



a. Initialize the elements in the array

Some 2D array problems are easier to solve if you can use a **row** and **column index** instead of the array **index**. The above screenshot shows how you can calculate the array index for a given column and row index. Define a function that solves this problem.

```
int GetIndex( int rowIdx, int colIdx, int nrCols );
```

Using a nested for-loop and this function, assign values to the elements, choose values that correspond with the element's row and column number, e.g. the element at the 3rd row and the 4th column should get the value 34.

b. Print elements

Print the elements of this array row per row and each row starting on a new line.

```
-- 2D arrays --
11  12  13  14  15
21  22  23  24  25
31  32  33  34  35
41  42  43  44  45
```

3.2. BasicAlgorithms

3.2.1. Create the project

Create a new project with name **BasicAlgorithms** in your **1DAExx_08_name_firstname** folder.

Add your name and group as comment at the top of the **BasicAlgorithms.cpp** file.

3.2.2. Objective

In this project you will define and use some basic functions that operate on arrays of int types. Carefully think about the parameter list of the functions. All

these functions – except the Swap function - need next to the **array** also **its size** as a parameter.

In the end you should have these functions (parameters are made invisible).

```
int Count(                                    );
int MinElement(                           );
int MaxElement(                           );
void Swap(                                    );
void Shuffle(                                          );
void BubbleSort(                           );
```

After having defined a function, test it by calling it several times with different test data (arrays). Make your code readable: put the tests in functions as well, e.g.

```
void TestCount( );
void TestMinMax( );
void TestSwap( );
void TestShuffle( );
void TestBubbleSort( );
```

3.2.3. Prepare test data

Test these functions on at least 2 arrays with different size and elements.

Initialize the elements with random values in an interval that includes **positive as well as negative values**.

3.2.4. Count function

This function returns the number of elements in the array that compare equal to a given parameter value.

```
-- Test of Count function --
7 -3 9 -8 3 0 -7 1 9 8
2 occurs 0 times in this array

-4 -1 2 0 -3 0 2 4 2 0 0 -5 -1 1
1 occurs 1 times in this array
```

3.2.5. MinElement/ MaxElement functions

Returns the smallest/largest value in the array.

```
-- Test of MinElement and MaxElement functions --
-4 2 5 5 3 2 -7 -7 3 3
-7 is the smallest value in this array
5 is the largest value in this array

-2 0 0 3 0 0 -5 -5 1 -3 -5 -4 3 3
-5 is the smallest value in this array
3 is the largest value in this array
```

3.2.6. Swap function

Swaps the values of two elements in an array as indicated by the parameters *idx1* and *idx2*.

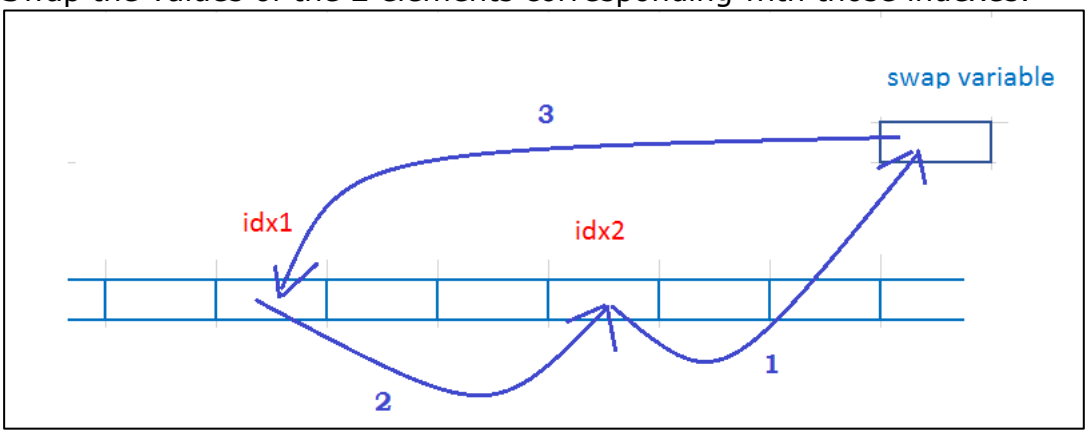
```
-- Test of Swap function --
Before swapping
1 10 -10 -8 5 5 -9 10 5 -7
After swapping the first and the last element
-7 10 -10 -8 5 5 -9 10 5 1

Before swapping
-2 3 2 -4 0 3 0 -4 -3 0 -4 -5 4 -2
After swapping the second and the second last element
-2 4 2 -4 0 3 0 -4 -3 0 -4 -5 3 -2
```

3.2.7. Shuffle function

This function shuffles the elements in an array, by swapping a number of times two random elements in the array. How many swaps are required is one of the parameters of the function.

a. Define the function

1. Generate 2 random numbers - idx1 and idx2 – that are not equal to each other, and are within the index boundaries of the array
2. Swap the values of the 2 elements corresponding with those indexes.



3. Repeat the 2 previous steps the requested number of times.

Test the function on at least two arrays.

b. Usefulness in games

In many games we want the objects to be in a random sequence and with each element occuring only once in the array.

For an array of numbers we could use rand() to have numbers in a random sequence, but the problem of uniqueness is not solved. The shuffle function comes to a rescue. Fill an array with the required numbers e.g. numbers from 1 till 10 and let the Shuffle function operate on it.

```
-- Test of Shuffle function --
Before shuffling
-2 1 4 -10 -5 6 -10 3 5 10
After shuffling 10 times
-10 1 10 3 4 -10 -2 -5 5 6

Before shuffling
-1 3 3 3 -3 3 -3 -5 0 -2 -2 -5 -4 2
After shuffling 20 times
3 -2 3 0 -3 -4 2 -5 -3 -1 -2 -5 3 3

Before shuffling
1 2 3 4 5 6 7 8 9 10
After shuffling 50 times
5 9 4 8 1 10 2 7 3 6
```

3.2.8. BubbleSort function

Bubble sort

Sorting is generally performed by comparing pairs of array elements, and swapping them to give them the desired sorting order (in ascending or descending order).

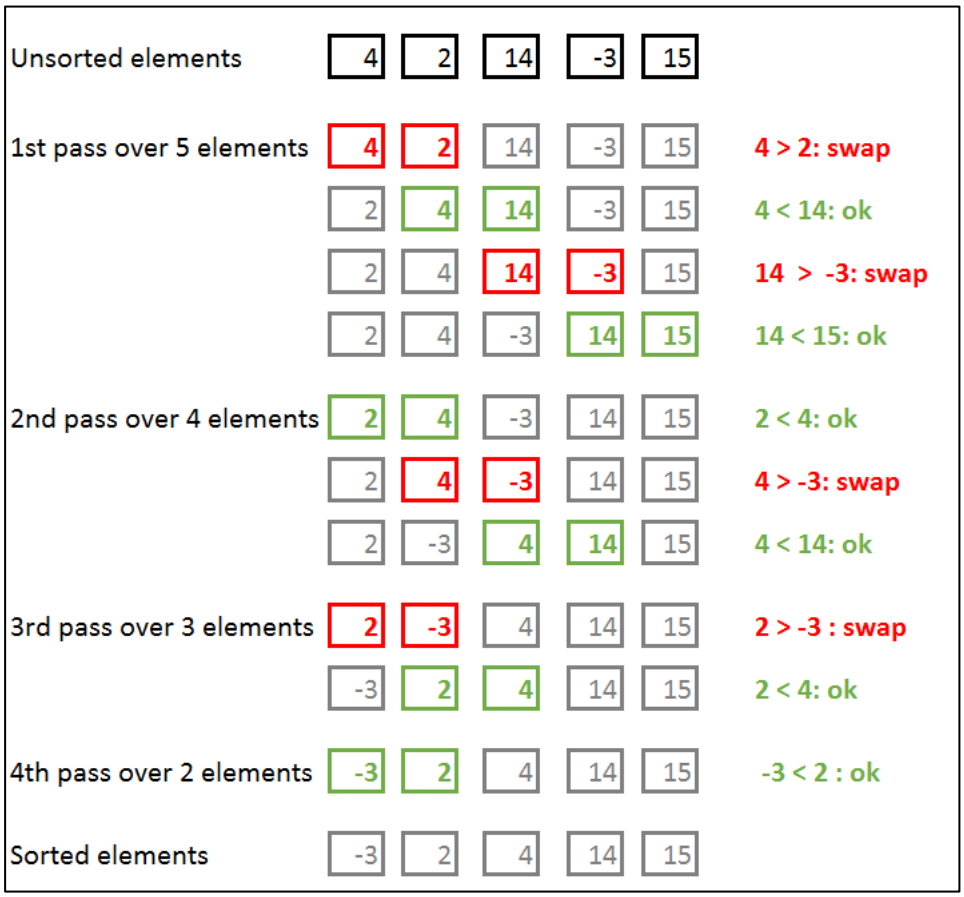
There exist different sorting algorithms, let's implement a rather simple one, the bubble sort algorithm.

The **bubble sort algorithm** makes multiple passes through a list of elements.

In a pass, it swaps the consecutive elements that are not in the correct order. In this way, the largest element bubbles towards the end of the list when sorting in ascending order

Then it goes again through all the elements except the last one, which is in the correct place.

Consequently, each pass the list of elements to compare becomes one element shorter. In the end, you have a sorted list of numbers.



Test your function on at least two arrays and print the results on the console.

```
-- Test of BubbleSort function --
Before sort
-10 1 10 3 4 -10 -2 -5 5 6
After sort
-10 -10 -5 -2 1 3 4 5 6 10

Before sort
3 -2 3 0 -3 -4 2 -5 -3 -1 -2 -5 3 3
After sort
-5 -5 -4 -3 -3 -2 -2 -1 0 2 3 3 3 3
```

Now use recursivity to do the bubble sort.

3.3. ArrayApplications

3.3.1. Create the project

Create a new SDL project with name **ArrayApplications** in your **1DAExx_08_name_firstname** folder.

Delete the generated file **ArrayApplications.cpp.**

Adapt the window title.

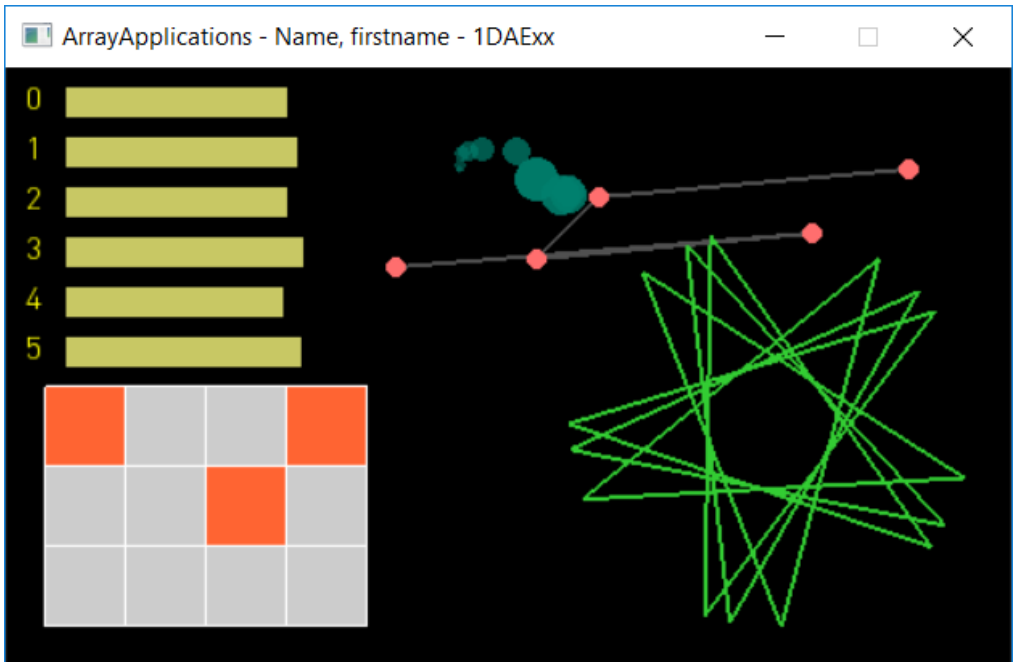
While making this exercise, decide yourself if you need to copy and add your utils files.

Create a folder **Resources**, copy the font file DIN-Light.otf into it.

3.3.2. Objective

In this exercise you make some exercises that display a series of items on the window whereby these items are stored in arrays.

At the end of these exercises, you should have a window that looks like this.



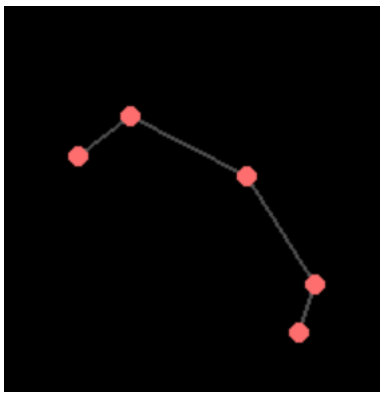
3.3.3. Click polygon

Draw a polygon with vertices corresponding to the 5 last left mouse button clicks. Keep the position of these points in an array of Point2f type elements.

Some tips:

- Define a constant variable that indicates the maximum number of clicked points: 5
- Define an array to hold this amount of clicked points, elements should be of type Point2f.
- Define a variable that indicates the index of the next clicked point, it should start at zero.
- Do not forget to initialize all these variables.
- Handle the mouse-button-up event - add code in the new function **OnMouseUpEvent**- and store the mouse position in the array. When all the elements in the array already contain a clicked point, free the last position by shifting all the elements in the array one position to the left (e.g. 1 towards 0, 2 towards 1). By doing so, the value of the first position is lost. Define to that end a function e.g. **AddClickedPoint.**
- Draw a red point at each clicked point and connect them with lines. Define a function e.g. **DrawClickedPoints**.

Build and test.



When this works fine, then change the maximum number to 10 for instance. If you only need to change the initialization of a variable, then this means that your code is **easy-to-maintain**.

3.3.4. Rotating pentagrams

In a previous lab you made a rotating pentagram but you didn't have any time information, so you just incremented the angle with a fixed value each frame. Now you always know the time a frame took, so you are able to calculate the angle provided that you know the angular speed.

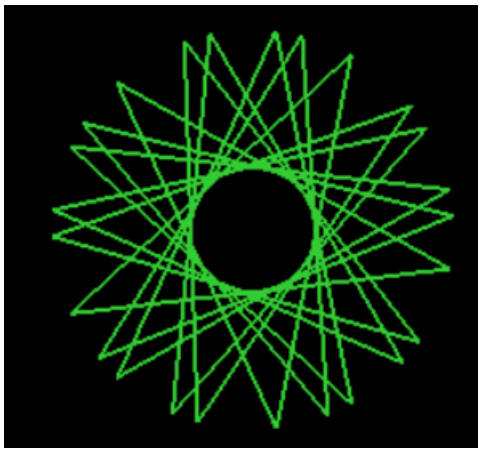
New angle = previous angle + angular speed * elapsed time

In this exercise you draw 5 rotating pentagrams. They all have the same center, radius and color and start at the same angle. However each one has its own angular speed, a random value in the interval [0.2,1.1]. So each pentagram also has its own angle value.

```
struct AngleSpeed
{
    float angle;
    float speed;
};
```

In main.cpp, define an **AngleSpeed** struct holding these 2 pieces of information. And define an array with elements of this new type.

In the Update function you calculate the new angle values. In the Draw function you draw them, like this.



Do not hesitate to define functions, e.g.

InitPentagrams

UpdatePentagrams

DrawPentagrams…

3.3.5. Statistics of the rand() function

We have been using **rand**() for some while. However, how random are these numbers actually; and is each number actually generated as much? To investigate this, you make the following application:

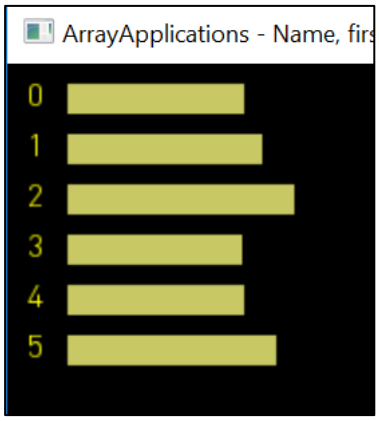
Investigate the frequency of the random numbers in the inclusive interval [0, 5].

In a first stage, display on the console each number and next to it the number of times it has been generated. Then visualize the statistics on the window.

Tips:

- Define an array in which you save the number of times each number is generated by using the formula: rand() % 6.
- Define a second **array** to save the **textures** of the strings representing these numbers.
- In the **Update** function, generate a random number in the investigated interval and increment the corresponding element in the array. Use the generated number as index in the array and increase with 1 the corresponding element. To that end, define and call a function **UpdateRandStats** that does this job.
- In the **Draw** function, visualize the content of the array as you can see in the screen shot below. First, we see the index of each number of the array, followed by a rectangle of which the width is dependent on the value in the array. Define a function **DrawRandStats** to that end.

Build and test.



Now do the statictics for numbers in the [0,3] interval. If you only need to change the initialization of a variable, then this means that your code is **easy-to-maintain**.

3.3.6. Mouse tracing

In this exercise, you draw a filled circle at the position where the mouse was situated. See illustration above.

Get inspiration from the click polygon exercise, define again an array to keep the 10 last mouse positions that you get from the **OnMouseMotionEvent**. Draw for each mouse position in the array a filled circle on the window. Give the older mouse positions a smaller radius and make them more transparent.

Again, do not hesitate to define functions, e.g.

AddMousePos

DrawMousePoints

Alternative way to add mouse positions. When the array is completely filled with mouse positions, adding a new position is done by shifting all the positions to the left, like in the click polygon. Can you find a more performant solution one that doesn't shift all the positions? Try to implement it.

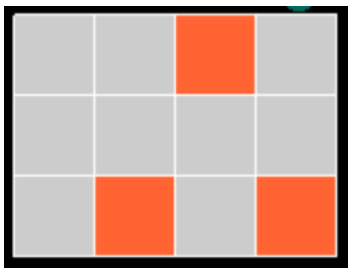
3.3.7. Select cells in grid

In this exercise, draw a 3x4 grid of selectable cells. A selected cell has another color then a not selected one.

The selection state of a cell changes when clicking with the **right mouse button** in it.

Tips:

- Define an array of 12 **bool** type elements; consider it as a 2D array of three rows and four columns.
- The value of an array element indicates whether the corresponding cell is selected.
- Give all the elements at start the value false, which means they are not selected.
- Draw the cells on the window using a color that corresponds with their selection state, also draw a border around each cell.
- When the right mouse is clicked inside the boundaries of a cell, change its selection state.



Define functions, e.g. **ToggleCell**, **DrawGrid**

3.4. ShuffleCards

3.4.1. Create the project

Create a new SDL project with name **ShuffleCards** in your **1DAExx_08_name_firstname** folder.

Delete the generated file **DrawFunctions.cpp.**

Adapt the window title.

While making this exercise, decide yourself if you need to copy and add your utils files.

Create a folder **Resources** and copy the given cards folder into it.

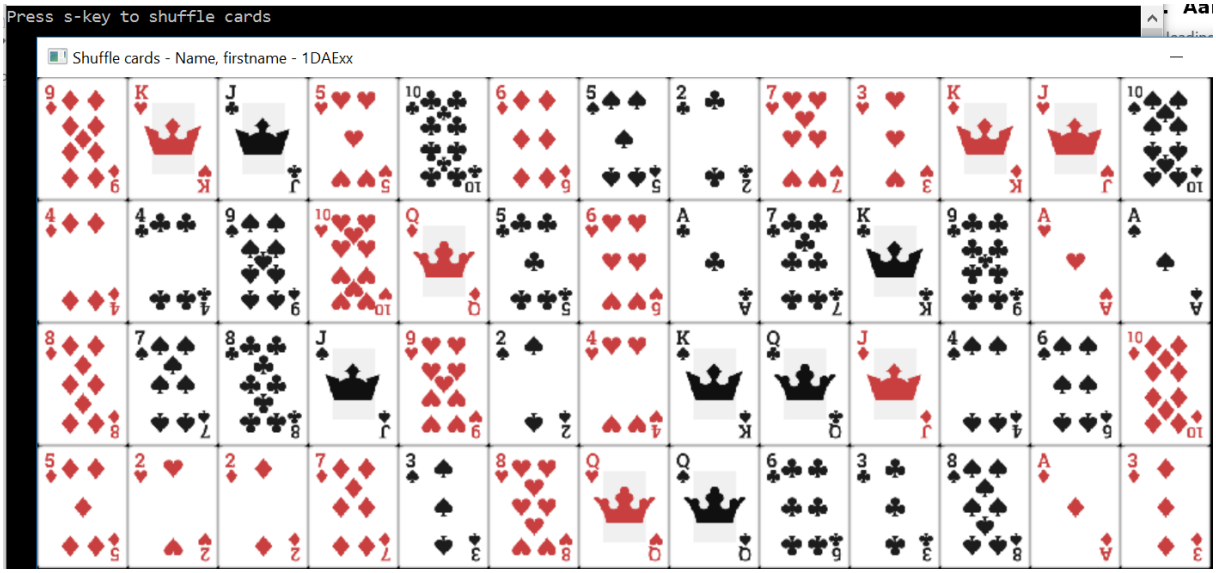
3.4.2. Objective

In this project you will use an array to hold 52 Texture type elements.

Each element corresponds with a card of a 52-card deck.

You draw the cards on the window in 4 rows.

When the s-key is pressed, you shuffle the cards by swapping randomly chosen card pairs a number of times.



3.4.3. Analyze the problem

What data do you need?

- 52 Textures -> Define an array of 52 Texture type elements

```
// Variables
const int g_NrCards{ 52 };
Texture g_Cards[g_NrCards]{};
```

What tasks have to be done?

- Create Texture objects
- Delete Texture objects
- Draw Texture objects
- Shuffle Texture objects.

Define a function for each task.

3.4.4. Create/delete the Textures

Given are the images of a card deck. The names of the image files have a number:

- The first digit indicates the suit (clubs, diamonds, hearts and spades).
- The next 2 digits indicate the rank: 01 to 13.

This naming allows you to create the Textures using iteration(s). Just build the path of the file using string concatenation and the std::to_string function.

Do not forget to release the textures. Also here use a loop.

3.4.5. Draw the Textures

You need to draw 4 rows of 13 cards:

- Cards with index 0 to 12 next to each other at the top
- Cards with index 13 to 25 next to each other below the previous row
- Cards with index 26 to 38 next to each other below the previous row
- Cards with index 39 to 51 next to each other below the previous row

Do this using iteration(s). Draw them at half their size.

3.4.6. Shuffle the textures

You already solved this type of problem for an array of int type elements. Again find 2 random indexes that are not equal and swap the Texture objects residing at these indexes. The swap variable is now a Texture type.

Please put the code that shuffles the cards in a function and call it in the Update function when the S key is pressed (Check the value of the corresponding element in the array returned by the SDL_GetKeyboardState function).

3.5. OxoGame

3.5.1. Create the project

Create a new SDL project with name **OxoGame** in your **1DAExx_08_name_firstname** folder.

Delete the generated file **OxoGame.cpp.**

Adapt the window title.

While making this exercise, decide yourself if you need to copy and add your utils files.

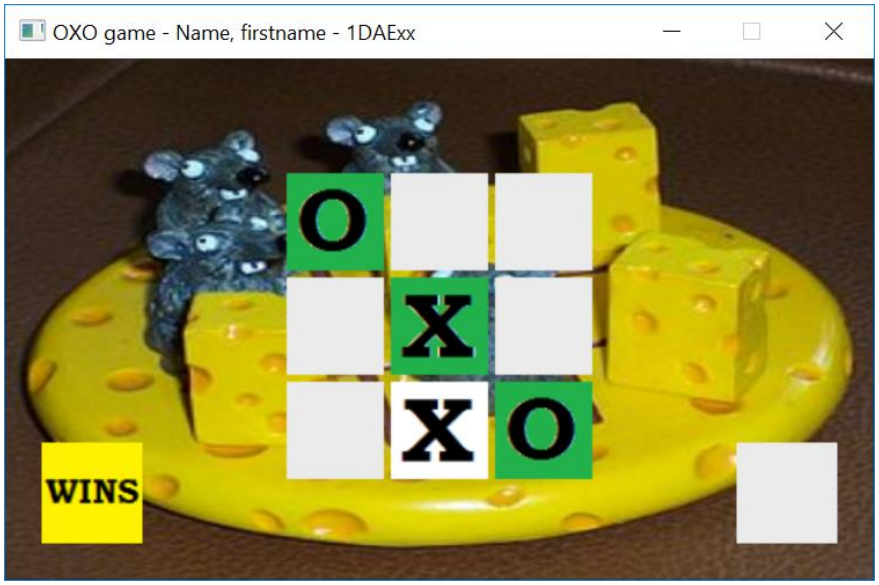
3.5.2. Objective

In this project you will make the '**OXO**' game. This is a variant of the **tic-tac-toe** game or in Dutch the **boter-kaas-en-eieren** (butter-cheese-and-eggs) game. In the 'OXO' game you must try to have the text "oxo" (letter o, not the digit 0) in a horizontal, vertical, or diagonal row of a grid. It is a turned based game, each player can put an X or O in a free cell. The player who makes the "OXO" combination wins the game.

How does it work:

- There are 2 players: one at the left, the other one at the right side.
- The player's choice between X or O is indicated in their bottom corners. They can switch this choice by pushing the **a-key** (left player) or the **l-key** (right player).
- The active player is indicated in green.
- Clicking with the **left mouse** in a free cell, puts the active player's choice in this cell and the other gamer becomes active.
- When there is a winning combination (in vertical, horizontal or diagonal direction) then these cells get a green color and the game is over. The winning player gets a yellow "Wins" indication

In the screenshot below the diagonal contains the "OXO" combination and the left player wins.



The background picture comes from
http://nl.wikipedia.org/wiki/Bestand:Bke.JPG

Its author is : https://commons.wikimedia.org/wiki/User:Richardkiwi

3.6. Sprites

3.6.1. Create the project

Create a new SDL project with name **Sprites** in your **1DAExx_08_name_firstname** folder.

Adapt the window title.

Create a **Resources** folder in the folder of this project and copy the given images into this folder:

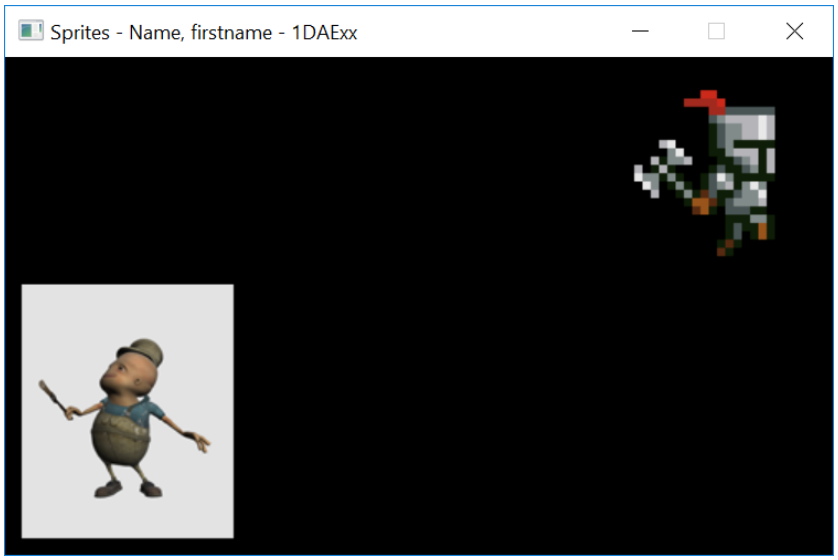
- RunningKnight.png

- Tibo.png

While making this exercise, decide yourself if you need to copy and add your utils files.

3.6.2. Objective

In this exercise, you will learn how to show an animated sprite on the window. In the end, you will have a window that shows an animated running knight and painting Tibo.



3.6.3. Sprite struct

Add next struct definition to your code. We will use variables of this new type to hold the information of an animated sprite.

The definition of this struct is also available on https://pastebin.com/q72NF3ut

```
struct Sprite
{
    Texture texture;
    int frames;
    int cols;
    float frameTime;
    int currentFrame;
    float accumulatedTime;
};
```

3.6.4. Knight sprite

- Define a **global variable of this new type**

```
// Variables
Sprite g_KnightSprite{};
```

- **InitGameResources** : Have a look at RunningKnight.png picture and initialize the g_KnightSprite members accordingly:
  - Create the texture object for the RunningKnight.png picture.
  - Number of columns is 8
  - Let's use a frame rate of 10 frames per second, so frame time is 1/10 seconds (= time 1 Knight frame lasts).

```
void InitKnight()
{
    bool isCreationOk{};
    isCreationOk = TextureFromFile("Resources/Sprites/RunningKnight.png", g_KnightSprite.texture);
    if (!isCreationOk )
    {
        std::cout << "Failed to load image RunningKnight.png.\n";
    }
    g_KnightSprite.cols = 8;
    g_KnightSprite.frames = 8;
    g_KnightSprite.currentFrame = 0;
    g_KnightSprite.accumulatedTime = 0.0f;
    g_KnightSprite.frameTime = 1 / 10.0f;
}
```

- **FreeGameResources**: Don't forget to delete the texture at the end.
- **Update**

Use the elapsed seconds to decide about the current frame number. Use the modulo operator to prevent the frame number from becoming too large:

```
void UpdateKnight(float elapsedSec)
{
    g_KnightSprite.accumulatedTime += elapsedSec;
    if (g_KnightSprite.accumulatedTime > g_KnightSprite.frameTime)
    {
        // Determine next frame number
        ++g_KnightSprite.currentFrame %= g_KnightSprite.frames;
        g_KnightSprite.accumulatedTime -= g_KnightSprite.frameTime;
    }
}
```

- **Draw**

Draw the running knight at the top of the window and enlarge it 5 times

```
void DrawKnight()
{
    // Part of texture that corresponds with the current frame number
    Rectf srcRect{};
    srcRect.width = g_KnightSprite.texture.width / g_KnightSprite.cols;
    srcRect.height = g_KnightSprite.texture.height;
    srcRect.left = g_KnightSprite.currentFrame * srcRect.width;
    srcRect.bottom = srcRect.height;

    // Draw it at the top of the window
    const float scale{ 5.0f };
    const float border{ 10.0f };
    Rectf destRect{};
    destRect.left = g_KnightPosition;
    destRect.bottom = g_WindowHeight - border - (g_KnightSprite.texture.height * scale);
    destRect.width = srcRect.width * scale;
    destRect.height = srcRect.height * scale;

    DrawTexture(g_KnightSprite.texture, destRect, srcRect);
}
```

- **Moving**

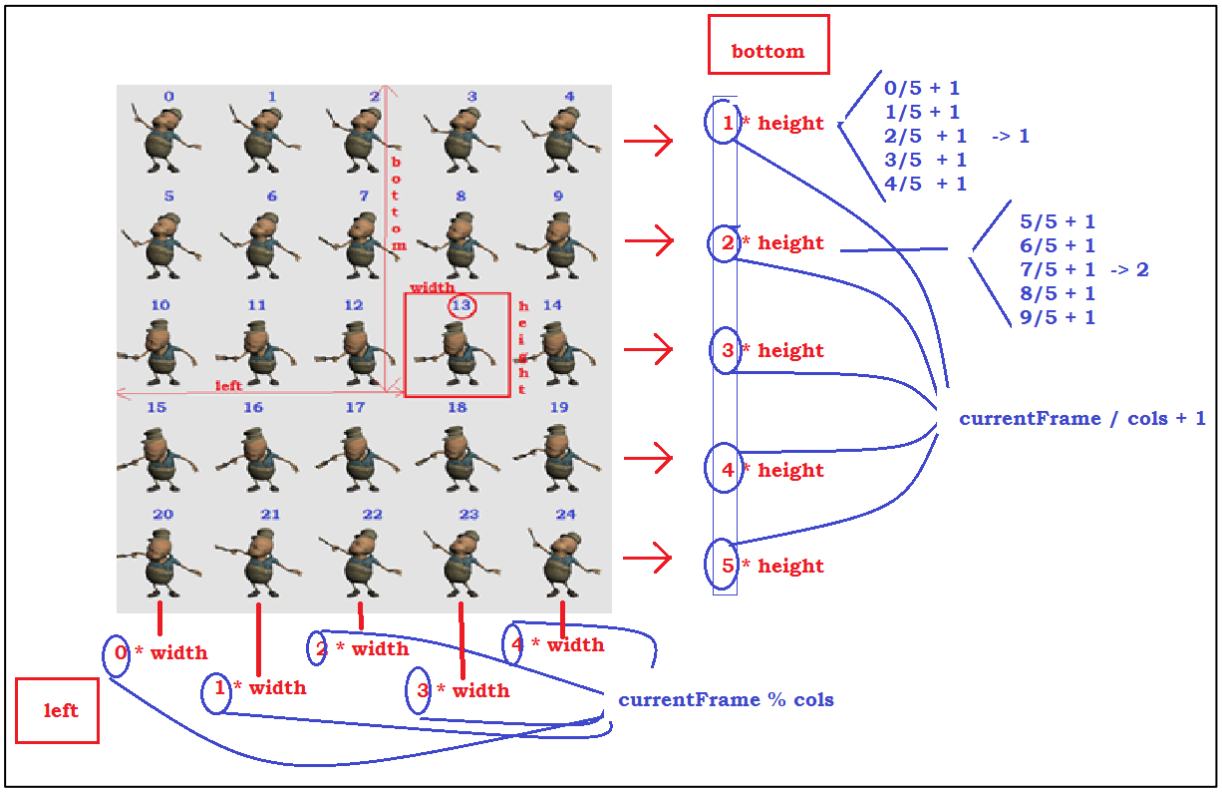
Each frame the knight moves some pixels. The horizontal speed of the knight is represented by the variable name g_KnightSpeed. It is a number of pixels per

second. Use the elapsedSec parameter to calculate its new position according to the elapsed time since last frame was drawn. You will also need a global variable to represent the knight position.

When it leaves the window, it starts at the left window side again.

3.6.5. Animated Tibo

Draw an animated Tibo at the bottom of the window, speed is 15 frames per second. The sprite sheet contains 25 frames arranged in 5 columns. Following screenshot helps you to calculate the bottom and left values of the source rectangular part corresponding with the current frame number.



4. Submission instructions

You have to upload the folder *1DAExx_08_name_firstname, however first clean up each project. Perform the steps below for each project in this folder:*

- In Solution Explorer: Select the solution, RMB, choose **Clean Solution**.
- Then **close** the project in Visual Studio.
- Delete the .vs folder.

Compress this *1DAExx_08_name_firstname* folder and upload it before the start of the first lab in the next lesson week, that's the week after Q1.

5. References

5.1. Arrays

5.1.1. Arrays part 1

https://www.learncpp.com/cpp-tutorial/61-arrays-part-i/

5.1.2. Arrays part 2

https://www.learncpp.com/cpp-tutorial/62-arrays-part-ii/

5.1.3. Arrays and loops

https://www.learncpp.com/cpp-tutorial/63-arrays-and-loops/

5.2. sizeof operator

https://en.cppreference.com/w/cpp/language/sizeof

5.3. Bubble sort

https://en.wikipedia.org/wiki/Bubble_sort

5.4. Recursion

https://www.learncpp.com/cpp-tutorial/7-11-recursion/

http://www.cplusplus.com/articles/D2N36Up4/

5.5. Sierpinski triangle

https://en.wikipedia.org/wiki/Sierpiński_triangle