Classes1

1. Content

2. Objectives

At the end of these exercises you should be able to:

- Define a class in 2 separate files a declaration and an implementation file.
- Define member functions (methods) and member variables.
- Make decisions on private vs public access specifiers.
- Implement encapsulation.
- Define and use (overloaded) constructors.
- Correctly use a constructor member initializer list.
- Know when a constructor / destructor are called (either implicitly or explicitly)
- Create classes that have a composition relationship
- Read UML class diagrams
- Describe a class using an UML diagram
- Describe a composition relationship in UML.

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. **ClassesBasics**. Other names will be rejected.

3.1. ClassesBasics

Create a new project with name **ClassesBasics** in your **1DAExx_10_name_firstname** folder.

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

In the file **ClassesBasics.cpp** you'll write code that uses and tests a class with name **Square**, a class that you'll define yourself.

Tip: put the code that tests the Square class in a separate function and call this test function in the main function.

In this exercise you'll experiment with:

- defining a new type (class),
- instantiating objects of the class on the heap (dynamic allocation)
- defining constructors
- passing objects to a function,
- deleting objects on the heap

3.1.1. Square class

Add 2 new items to your project **Square.h** and **Square.cpp**.

Include the **pch.h** file at the top of Square.cpp.

And complete these Square files with the definition of the Square class as specified hereafter.

a. The data members of the Square class

This class represents an axis aligned square. What could be the properties (data members) of this kind of square?

- The position on the window
- The length of its sides

Add in Square.h, the definition of the data members that will hold these properties. Notice that their names start with m_

Let's first define the 3 properties without initializing them.

Square
- m_Left: float - m_Bottom: float - m_Size: float

b. Content of data members after creating objects

Now we'll do some tests of the Square class. Therefor declare and define a test function – DoSquareTests - in ClassesBasics.cpp and call it in the main function.

Data members that are not initialized

In the test function DoSquareTests create 1 local object of this class on the heap as in following screenshot. Notice that a pointer type variable name starts with p.

```
Square* pSq3 = new Square{};
```

Rebuild and notice you do not get a warning. Add a breakpoint after the creation of this object, run and have a look at the content of the Square object in the Locals window. What do you notice?

Requesting the size property value via the constructor

A Square object with size equals 0 makes no sense. We should at least know the size of a Square object when it is created. Let's add a constructor that requires a value for the size property whenever an object of this Square class is created. And in the initializer list, copy the value of this size parameter in the corresponding data member.

```
Square( float size );
```

Stop at the breakpoint and check out the values of the other member variables. What do you notice? Now solve the problem by initializing the remaining member variables in the constructor member initializer list and do the test again.

Requesting size and position values via an overloaded constructor

Now we can't choose the position of the Square objects, their bottom left corners all coincide with the origin.

Define a second constructor which takes 3 parameters (size, left and bottom) and again use these parameter values to initialize the corresponding data members.

```
Square( float size, float left, float bottom );
```

Create at least one extra object using this constructor.

Build and debug, have a look at the data members. They should all be initialized with the correct values.

c. Define a destructor

Suppose you created some resources (e.g. textures) in the constructor. Then before deleting the object, these resources should be released again.

Add a destructor to the Square class that prints a message on the console telling that the resources are deleted.

```
Square::~Square( )
{
    std::cout << "Square destructor: release the resources\n";
}
```

Launch your application and verify that the destructor message appears as many times as you created an object.

Here you should notice a problem, apparently the destructor is not always called automatically.

When a pointer goes out of scope the related object is not deleted automatically.

Change your code: explicitly delete the Square object(s) that you created on the heap.

Test again, count how many times the destructor is called.

d. You can't delete an object 2 times

Delete the object on the address as indicated by the pointer – and which doesn't exist anymore – a second time. Build and run your application. What happens?

This is called a **double free error**.

A pointer type variable that holds the address of a released object is called a **dangling pointer**.

Print the value of the pointer to the console before and after deleting the object. It is the address of the Square object. You see that its value does not change by deleting the object. There is no way of knowing if a pointer is dangling or not.

Before deleting again, assign the **nullptr** value to the pointer variable. Build and run. What happens now?

So after having deleted an object on the heap, then it is a good idea – especially when the pointer variable is still available - to assign the **nullptr** value to the dangling pointer because the address that it contains isn't valid anymore as the object on that address doesn't exist anymore.

e. Add functionality to the Square class

What services could a Square object offer?

- A print of its own properties and some extra information such as the perimeter and area values
- Changing its position
- Changing its size

Add member functions – also called methods - that perform these tasks. They are indicated in bold in next UML class diagram. Notice also the 2 helper (private) functions that calculate the area and perimeter.

Square
+ Square(size: float) + Square(size: float, left: float, bottom: float) + ~Square( ) **+ Print( ): void** **+ Translate(deltaLeft: float, deltaBottom: float): void** **+ SetSize(newSize: float): void** **- GetArea( ): float** **- GetPerimeter( ): float**

The **Print** method prints information about the Square object like this:

```
Left: 10.00, bottom: 20.00
Size: 3.00
Perimeter: 12.00
Area: 9.00
```

The **Translate** method changes the left and bottom values of the square: increment them with the parameter values of this method.

The word **delta** is often used to indicate the change in some quantity. Sometimes only the letter d is mentioned, e.g. dLeft, dBottom…

The **SetSize** method gives the size a new value.

f. Ask an object to execute one of its functions

In the DoSquareTests function:

- Ask all your Square objects to print themselves.
- Translate some of them, change the size of some and ask again to print their data.

```
Left: 12.00, bottom: 24.00
Size: 3.00
Perimeter: 12.00
Area: 9.00
```

g. Passing objects to a function

This is an important exercise, add your conclusions to your own summary.

Passing the address of an object to a function

In the ClassesBasics.cpp file: declare and define the **TestSquare** function, one that requires a Square **pointer** as a parameter. In this function, call the move function of the Square object pointed to by the pointer argument.

```
void TestSquare(Square* pSquare);
```

In the DoSquareTests function, call this function by passing the address of a square created on the heap.

Using the debugger verify whether the square object has moved after calling this function. Also check the value of the pointer argument.

3.1.2. Time class

This class represents the time in a 24 system**. It keeps only seconds**, thus has only one member variable (**m_Seconds**).

What functionality does it offer?

- An object can be created using one, two or three parameters.
  - One: the seconds
  - Two: the minutes and seconds
  - Three: the hours, minutes and seconds
- Print the time in this format: **Time: xx:yy:zz** where xx is hours, yy is minutes and zz is seconds.
- Return the seconds (e.g. Time 23:16:20 returns 20)
- Return the minutes (e.g. Time 23:16:20 returns 16)
- Return the hours (e.g. Time 23:16:20 returns 23)
- Add seconds to the time

- Add minutes to the time
- Add hours to the time

Functions that return the value of a data member have often a name that starts with **Get**…, for instance GetSeconds, GetrMinutes, …

In ClassesBasics.cpp: declare and define a function **DoTimeTests** in which you test every member function of the Time class (also the 3 constructors) by calling it, followed by a print of the time. Call DoTimeTests in the main function.

Don't forget to delete the time object on the heap.

3.2. GraphicClasses

Create a new SDL project with name **GraphicClasses** in your **1DAExx_10_name_firstname** folder.

Delete the generated file **GraphicClasses.cpp.**

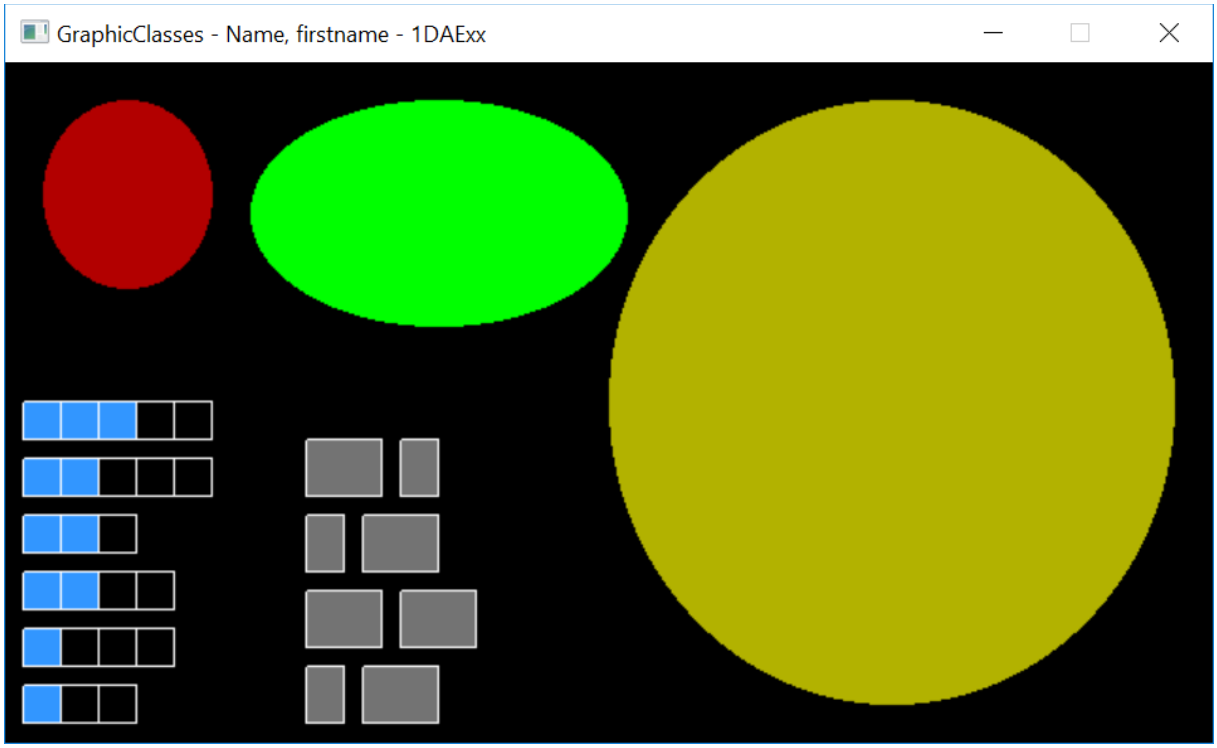
Adapt the window title.

In this project you will define 3 classes that draw on the window. Put the definition of every class in a separate cpp and header file.

After having defined a class, you test it in **Game.cpp**, you:

- Create objects of these classes
- Call their methods

In the end you should have a window that looks like this.



3.2.1. Fraction class

Add new items Fraction.h and Fraction.cpp to this project.

a. Define the class

The fraction class represents a simple fraction (e.g 2 / 4) consisting of an integer numerator (2) and denominator (4).

This is the UML diagram:

Fraction
- m_Numerator: int - m_Denominator: int
+ Fraction(numerator: int, denominator: int) + Draw(pos: const Point2f&, size: float): void + GetValue( ): float

The **Draw** method draws the fraction starting at position **pos** and as a number of cells horizontally next to each other (number of cells = denominator, number of colored cells = numerator). The **size** parameter indicates the size of one cell.

The **GetValue** method returns the value of the fraction, this is the result of dividing the numerator by the denominator.

Remark:

b. Test the Fraction class in GraphicClasses.cpp

Create Fraction objects

Now create **6 instances of the Fraction class on the heap** giving each object different numerator and denominator values using uniform initialization:

- In Game.h, **declare** a **pointer** of **Fraction** type, and build the project. As you notice, there is an **error**:

  ```
  error C2143: syntax error: missing ';' before '*'
  ```

  or

  ```
  E0020    identifier "Fraction" is undefined
  ```

  Intuitively, you would include Fraction.h, but let's not do that. Including one header file in another header file is to be avoided if possible. Knowing that there is a class called Fraction is the only thing the compiler needs to know. For that we can use a **class forward declaration**.
- In Game.h: do a **class forward declaration** of the Fraction class, **before declaring pointers** to Fractions. This **tells** the **compiler** that there is a class called Fraction. This is **ok** when **pointers** are **declared**.

  ```
  // forward class declaration
  class Fraction;
  Fraction* g_pFraction1{ nullptr };
  Fraction* g_pFraction2{ nullptr };
  ```
- Create a function CreateFractions in Game.h and cpp that will use dynamic memory allocation to create the fraction objects on the heap.
- In that function, create these objects. You will notice there is a problem with that:

  ```
  void CreateFractions()
  {
      g_pFraction1 = new Fraction{10,20};
      g_pFraction2 = new Fracti
      g_pFraction3 = new Fracti
      g_pFraction4 = new Fracti
      g_pFraction5 = new Fracti
      g_pFraction6 = new Fracti
  }
  ```

  class Fraction
  forward class declarations
  Search Online
  incomplete type is not allowed
  Search Online

- The compiler has no clue how the constructor of that class looks like. To provide this information, you must here in Game.cpp, include the Fraction header file to define the Fraction class. Now everything should be fine again. Call this function in Start();
- Create a function that deletes all the fraction objects, and call it in End()

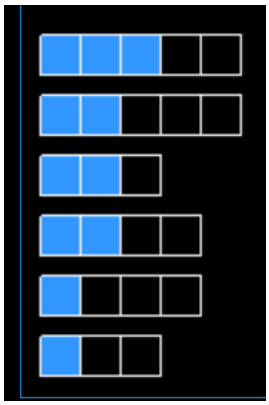
Show the total value

Declare and define a function **PrintFractionsSum** that gets and adds the values of all these Fraction objects – using their GetValue function - and that prints this total value on the console. This total value has to be drawn only one time, so call the PrintFractionsSum function at the start of the game, e.g. in the **Start** function.

```
Total value of created Fraction objects: 2.75
```

Draw the Fraction objects on the window

Declare and define a function **DrawFractions** that draws all the fraction objects under each other on the window - using their Draw function. All drawings on the SDL window should happen during execution of the **Draw** function, so call this DrawFractions function in the Draw function.



3.2.2. Light class

a. Define the class

This class represents a colored light that can be turned on or off. This is the UML class diagram.

Light
- m_Rect: Rectf - m_IsOn: bool - m_Color: Color4f
+ Light(rect: const Rectf&, color: const Color4f&) + Draw( ): void + IsHit(pos: const Point2f&): bool + IsOn( ): bool

The **Draw** method draws a colored square when the light is on, and a gray one when it is off. It has a white border.

The **IsHit** method toggles the light on/off when the parameter pos is within the square boundaries. It return true when the state of the light has changed, otherwise it returns false.

The **IsOn** method returns true when the light is on, otherwise it returns false.

b. Test the Light class in GraphicClasses.cpp

Create Light objects

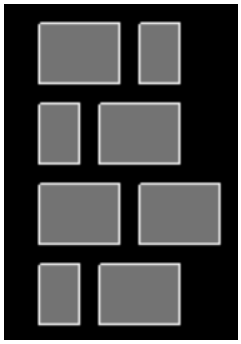
Declare and define a function **CreateLights** that creates 8 Light instances **using dynamic memory allocation and keeping the addresses in Light pointer type variables**. Hereby position them in a grid of 4 rows and 2 columns using (limited) random sizes and random colors. This creation should happen only once, so call this function in the **Start**.

Delete the Light objects

Not deleting these objects on the heap, results in memory leaks. Do not forget to delete the dynamic allocated objects! Declare and define a function **DeleteLights** in which you delete all the dynamically created Light objects. Call the function in the appropriate function.

Draw the Light objects

Declare and define a function **DrawLights** that draws the 8 Light objects on the window using their Draw function. Again, this should be done during the execution of the **Draw** function.



Switch the lights on/off when hit

When the left mouse button is pressed above a light, that light should be turned on/off. So when the left mouse button is pressed, check the mouse position hit on every Light object use IsHit). Therefor first declare and define a function **HitLights** that takes one parameter - the clicked position - and that checks that position on every Light object.

```
void HitLights( const Point2f& pos );
```

Then call this function in the appropriate mouse event function.

Also print the number of switched on lights on the console each time the left mouse is clicked on a Light object, therefor check the return value of the IsHit function. And only if one of the lights changed state, show the message. So no message should be printed when none of the lights is hit.

```
5 lights are on
```

3.2.3. DaeEllipse class

In this exercise you define a DaeEllipse class that represents a colored ellipse. It has this functionality:

- It can draw itself on the window
- It gets activated when the mouse hovers over it
- It becomes deactivated when the mouse leaves it.

A deactivated ellipse is partly transparent, an activated one is completely opaque.

a. Define the class

This is the UML diagram, you decide about helper functions:

DaeEllipse
- m_Center: Point2f - m_RadX: float - m_RadY: float - m_FillColor: Color4f - m_Area: float - m_IsActive: bool
+ DaeEllipse(center: const Point2f&, radX: float, radY: float) + DaeEllipse(center: const Point2f&, radX: float, radY: float, fillColor: const Color4f&) + Draw( ): void + ActivateTest(pos: const Point2f&): void - …

The class has 2 constructors, when the color is not specified then a random color is used. Use constructor delegation. Calculate the area (Area of an Ellipse) in the constructor.

The **ActivateTest** method activates the ellipse when the parameter pos is within the ellipse boundaries (Point in an Ellipse). It deactivates the ellipse when this is

not the case. When a not-activated ellipse becomes activated, this method prints its area to the console.

The **Draw** method draws a filled ellipse. A not activated ellipse is drawn partly transparent.

Don't hesitate to define some helper functions (private), e.g. a function that calculates the area.

b. Test the class in GraphicClasses.cpp

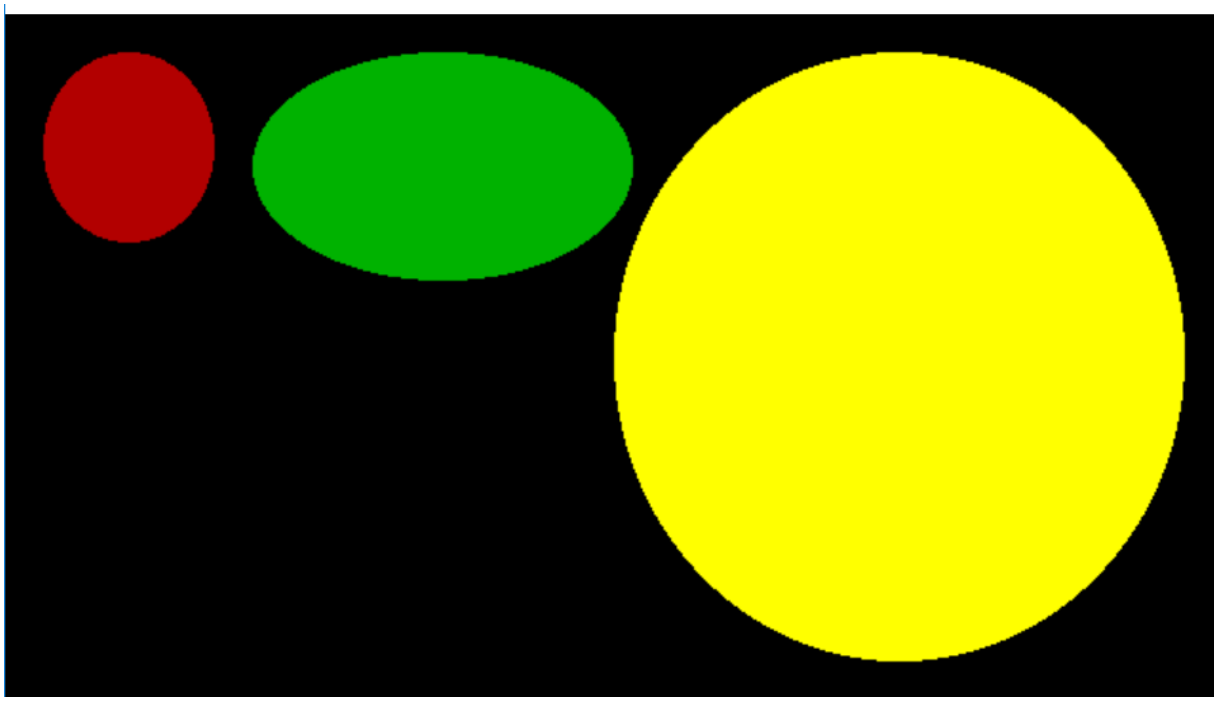
Again declare and define helper functions for each of the following tasks.

Create 3 DaeEllipse objects

At the start of the program, make 3 instances of this class **on the heap**, using both constructors**.**

Draw the DaeEllipse objects

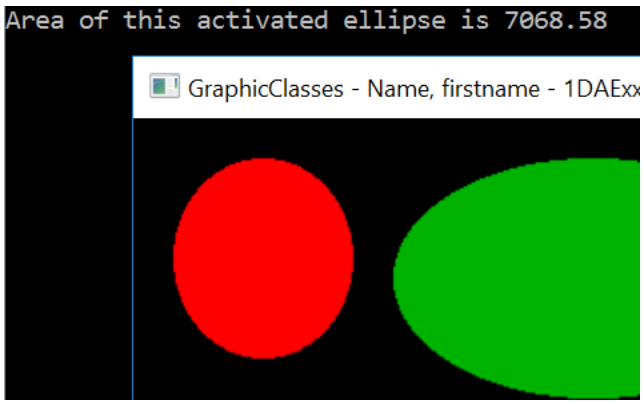
Draw them on the window.



Activate/deactivate the ellipses

When the mouse hovers over an ellipse, it gets activated (is no longer transparent). You should only call the ActivateTest method of the DaeEllipse objects when the mouse moves. Thus call this method in the mouse motion event function.

Build, run and test. Also verify that the area is printed on the console when a DaeEllipse object becomes activated. In the next screenshot the red one is activated (it is no longer transparent) and its area is printed on the console.



Delete the DaeEllipse objects

When the program stops, delete the DaeEllipse objects you created on the heap.

4. Submission instructions

You have to upload the folder *1DAExx_10_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_10_name_firstname* folder and upload it before the start of the first lab next week.

5. References

5.1. Classes in C++

http://www.learncpp.com chapter 8 parts 1, 2, 3, 4, 5, 5a, 5b, 7, 9

5.2. Ellipses

5.2.1. Point in an Ellipse

https://math.stackexchange.com/questions/76457/check-if-a-point-is-within-an-ellipse

5.2.2. Area of an Ellipse

https://www.wikihow.com/Calculate-the-Area-of-an-Ellipse