Classes2

1. Content

2. Objective

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

- Indicate that a member function doesn't change the state of an object
- Define and use static data members and static functions.
- Indicate that a member is static in an UML diagram
- Indicate that a member function is const in an UML diagram

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

3. Create one solution containing several projects

In this course programming2 you will group all the exercises of one week in one Visual Studio solution with name Wxx (xx being the week number).

To do this, follow the instructions in 00_General / ManualsAndGuides / HowToCreateAMultiProjectSolution.pdf

4. 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. **Classes2Basics**. Other names will be rejected.

4.1. Classes2Basics

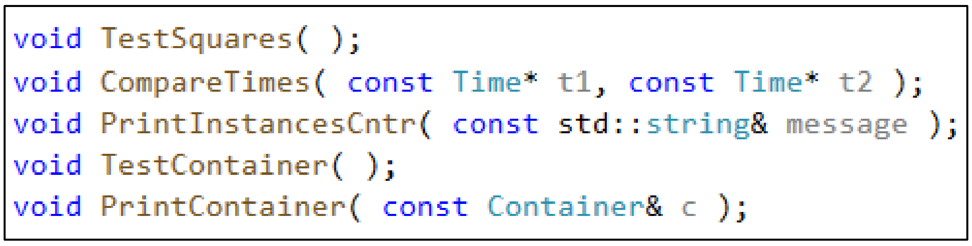
Add a new console project with name **Classes2Basics** to the W02 solution.

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

Copy and add the **Square** and **Time** class files from the resources zip file on Leho. They were already introduced in programming 1.

4.1.1. Readable code

Keep your code readable, define functions. In the end we have these functions in this project.

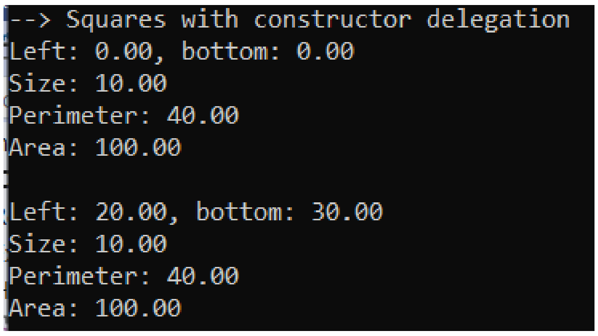


```cpp
void TestSquares( );
void CompareTimes( const Time* t1, const Time* t2 );
void PrintInstancesCntr( const std::string& message );
void TestContainer( );
void PrintContainer( const Container& c );
```

4.1.2. Constructor delegation in Square class (TestSquares)

Change the definition of one of the Square constructors: use constructor delegation.

Verify whether the Square class still works fine: create a Square instance using the changed constructor and print the object. Verify that the data members are all well initialized.



```
--> Squares with constructor delegation
Left: 0.00, bottom: 0.00
Size: 10.00
Perimeter: 40.00
Area: 100.00

Left: 20.00, bottom: 30.00
Size: 10.00
Perimeter: 40.00
Area: 100.00
```

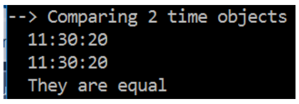
4.1.3. Const members (CompareTimes)

Following exercises is about **const member functions, thus functions that don't change the state of an object**.

Declare and define following function.

```
void CompareTimes( const Time* t1, const Time* t2 );
```

This function prints both Time objects using their Print method. And then verifies whether they contain the same time by comparing the seconds, minutes and hours using the Time methods that query these data. And it prints the result of this test. Like this.

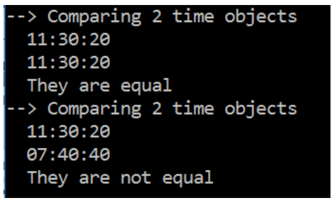


```
--> Comparing 2 time objects
  11:30:20
  11:30:20
  They are equal
```

This code leads to errors if you didn't indicate that the Print function isn't a const function. Solve this error without changing the parameter list of the CompareTimes function, but change your Time class. Indicate in the Time class which methods don't change the state of a Time object.

Now the function should build without any problems.

In the main function define 2 Time objects using dynamic memory allocation and call the CompareTimes function first passing the same Time objects and after that by passing different Time objects.



```
--> Comparing 2 time objects
  11:30:20
  11:30:20
  They are equal
--> Comparing 2 time objects
  11:30:20
  07:40:40
  They are not equal
```

In all further class definitions and UML, we expect you to specify – using const - when a method doesn't change the state of an object.

4.1.4. Static class members (PrintInstancesCntr)

a. Extend the Time class

Extend the Time class like this:

- With a static data member – **m_NrInstances** - that holds a counter of the number of Time objects and
- With a static function - **GetNrInstances**( ) - that queries this number.
- When an object of this class is created, the counter has to be incremented (adapt the constructors).

- When an object of this class is deleted the counter has to be decremented (add the destructor definition).

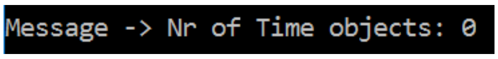
The GetNrInstances method doesn't change the state of a Time object. **Can you indicate this static method as being const? Can you explain why?**

b. Print the number of instances

Add a function **PrintInstancesCntr** to Classes2Basics.cpp. This is the declaration.

```
void PrintInstancesCntr( const std::string& message );
```

Also define this function: it calls the function GetNrInstances of the Time class and prints the result preceded by the message – as indicated by the parameter of the function – on the console. For example if the parameter contains "Message":

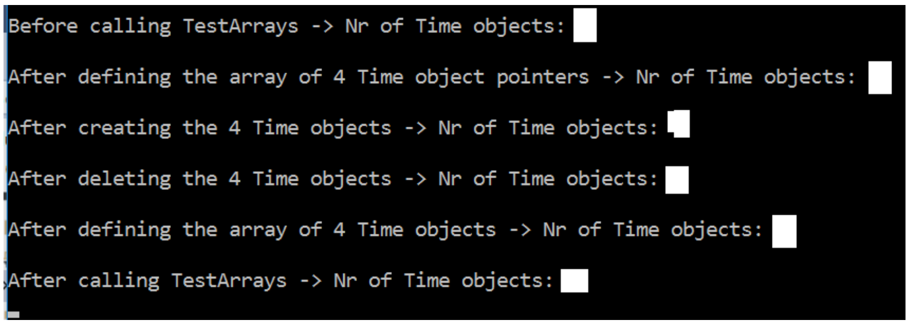


```
Message -> Nr of Time objects: 0
```

Now call this method several times, with a message that indicates where it is called

- Before calling TestArrays, there should be no instances (objects)
- After calling TestArrays, there should be no instances
- In TestArrays: after defining the array of 4 **Time object pointers**
- In TestArrays: after creating the 4 Time objects
- In TestArrays: after deleting these 4 Time objects
- In TestArrays: after defining the array of 4 **Time objects**

In the end you get these message on the console.



```
Before calling TestArrays -> Nr of Time objects:

After defining the array of 4 Time object pointers -> Nr of Time objects:

After creating the 4 Time objects -> Nr of Time objects:

After deleting the 4 Time objects -> Nr of Time objects:

After defining the array of 4 Time objects -> Nr of Time objects:

After calling TestArrays -> Nr of Time objects:
```

Ask the teacher for more info if you don't understand the results.

4.2. Container class

This class will be further elaborated in many subsequent labs of Programming 2!

In previous lab, you were introduced to the standard library vector. To profoundly understand how it works and behaves, we will now "reverse engineer" the std::vector class. Knowing how to create and delete dynamic arrays, let's make a Container class that manages a collection of integers and that resizes automatically when the capacity is not big enough. Do this in a new project called Container.

Once you programmed this container class you should perfectly understand how the internals of the std::vector class do work.

This is the UML class diagram. Decide yourself which methods don't change the state of the object and should be const (it is not indicated in this UML).

```
Container
- m_Size: int
- m_Capacity: int
- m_pElement: int*

+ Container(capacity: int = 10)
+ ~Container( )
+ Size( ): int
+ Capacity( ): int
+ Get(index: int ): int
+ Set(index: int, newValue: int ): void
+ PushBack(element: int ): void
- Reserve(newCapacity: int ): void
```

a. Data members

m_Size: indicates how many values this container contains

m_Capacity: indicates how many values the container can contain without having to resize

m_pElement: contains the address of the first element.

b. Methods

Constructor: allocates memory for a number of int-types as indicated by the *capacity* parameter

Destructor: deallocates the memory

PushBack: adds a new element at the end of the container, after its current last element. This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space (using Reserve, see below, e.g. make the new capacity twice the current one + 1) if - and only if - the new container size surpasses the current container capacity.

Size: returns the number of added elements

Capacity: returns how many elements are allocated

Get : returns the element at index as indicated by the *index* parameter

Set : changes the element at index as indicated by the *index* parameter, it gets the value as indicated by *newValue*

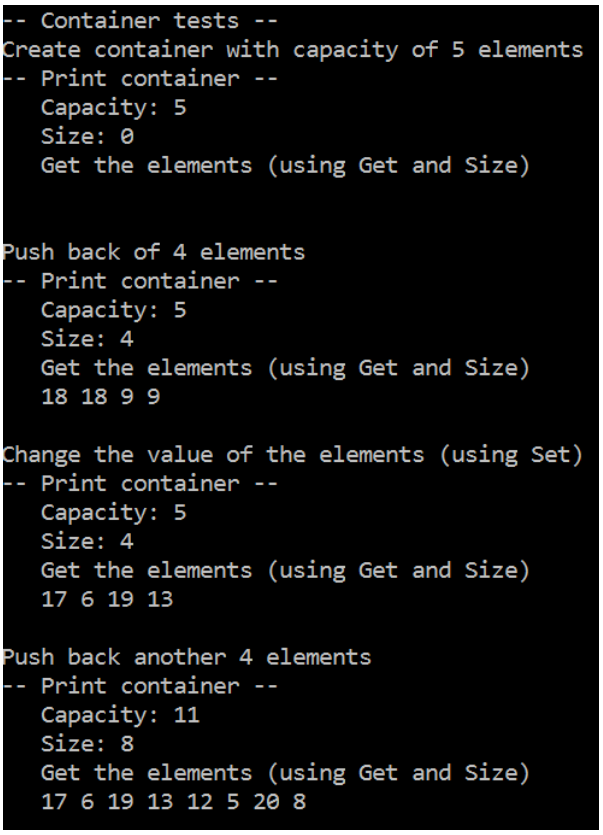
Reserve: extends the capacity of the container

- allocates memory as indicated by the *newCapacity* parameter
- Copies the current elements into this new memory
- Deallocates the old memory occupied by the copied elements
- Makes m_pElement point to the new allocated memory
- Changes m_Capacity to the new capacity.

c. Test this class

You get two function definitions to test the Container class (TestContainer and PrintContainer). They are in the given file **ContainerTestFunctions.cpp**. Copy both function definitions in your Classes2Basics.cpp, add the declarations yourself and call the **TestContainer** function in your main function. Do not change the test code to make your Container class pass these tests.

This is an example output of the test.



```
-- Container tests --
Create container with capacity of 5 elements
-- Print container --
   Capacity: 5
   Size: 0
   Get the elements (using Get and Size)


Push back of 4 elements
-- Print container --
   Capacity: 5
   Size: 4
   Get the elements (using Get and Size)
   18 18 9 9

Change the value of the elements (using Set)
-- Print container --
   Capacity: 5
   Size: 4
   Get the elements (using Get and Size)
   17 6 19 13

Push back another 4 elements
-- Print container --
   Capacity: 11
   Size: 8
   Get the elements (using Get and Size)
   17 6 19 13 12 5 20 8
```

4.3. Smileys

4.3.1. Create the project

Add a new project with name **Smileys** to your W02 solution.

Copy the **Resources** folder in the project folder.

4.3.2. General

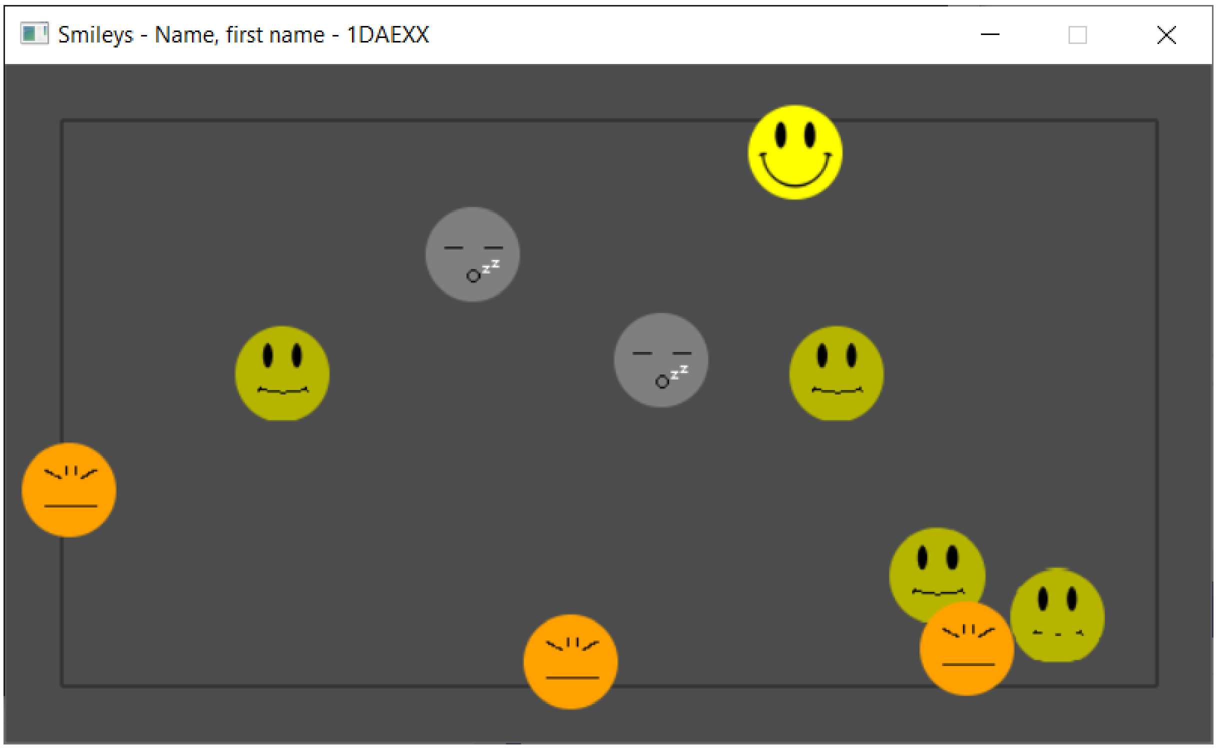
In this project several smileys **move** inside a rectangular container (window). They change direction when reaching the container’s edges.

A smiley can be in one of 4 states: neutral, happy, sleeping, or scared. A smiley is **happy** when it has the highest position of all smileys and it gets **scared** when not completely inside the safe area. When it is hit with the mouse, it starts **sleeping** or awakes. When it is sleeping it doesn’t move.

Using the **arrow up and down keys** one can increment/decrement the speed of the smileys.

When pressing the delete-key, the sleeping smileys are deleted.

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



4.3.3. Smiley class

The data and functionality of a smiley will be encapsulated in a Smiley class.

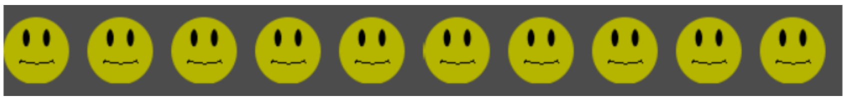
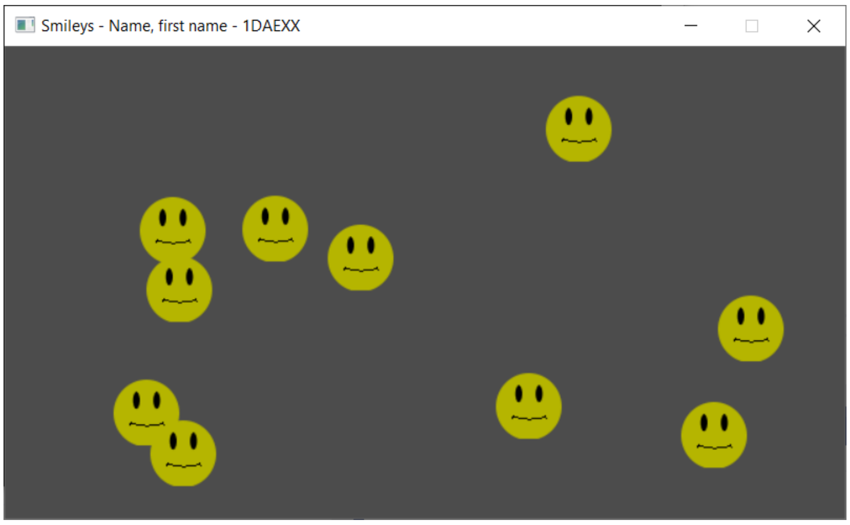
Add 2 new items **Smiley.cpp** and **Smiley.h** to the Game Files.

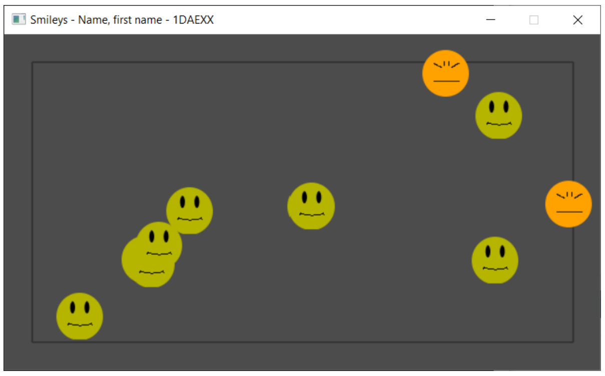
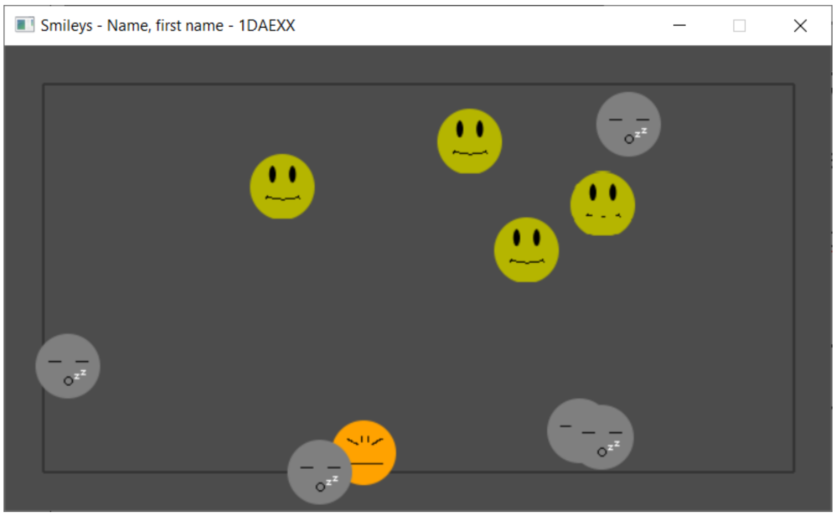
To gain some time you get the content of Smiley.h and Smiley.cpp. The method definitions are missing, don't define them all at once, work step-by-step as described below. This is far more motivating because after each step you can build, run and admire the result.

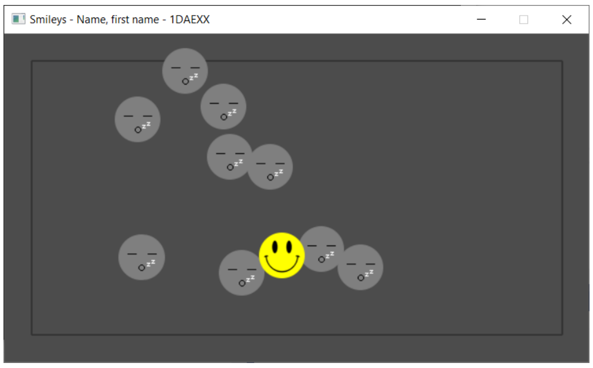
Also decide which methods don't change the state of the object and indicate them as being **const**.

4.3.4. Steps

Smiley class	Game class
Define the **constructor** and **destructor**.	Define an array of 10 Smiley pointers, you can use a static array. Use a **static const** variable for the size of the array. At the start of the game, create 10 Smiley objects, positioning them on half the window height and next to each other. At the end of the Game, delete the smiley objects. Use helper (thus private) functions, e.g. CreateSmileys, DeleteSmileys.

Build and debug: In Game.cpp, enable a breakpoint after the code that creates the Smiley objects. When the program stops at this breakpoint, verify the content of the 10 objects in the Locals window, then continue execution. Stop the application using the SDL window close button and have a look at the output window, verify that no memory leaks were reported.	
Smiley class	**Game class**
In the **Draw** method, draw the neutral smiley	In the Draw method, use the helper function "DrawSmileys" to draw the Smileys by calling their Draw method.
Build and run. Verify that 10 smileys are drawn at their creation positions. All showing the neutral frame. 	
Smiley class	**Game class**
Define the **Update** method: make the Smiley move according its velocity, and make it bounce against the edges of the bounding rectangle. Postpone the "in safe area check".	In the Update method, call the Update method of the 10 smileys. The bounding area is the window itself. The safe rectangle is at a distance of 30 pixels from the window edges
Build and run. Now the smileys should bounce around and never leave the window.  	
Smiley class	**Game class**
IsInSafeArea: returns **true** when the Smiley is completely inside the safe	Visualize the safe area rectangle

area else **false** is returned. **Update**: Add the in safe area check. **Draw**: if the smiley is not in the safe area then draw the scary smiley	
Build and run. Now, when a smiley goes outside the safe area, it should show the scary face.  Smileys - Name, first name - 1DAEXX	
Smiley class	**Game class**
HitTest: complete the definition of this method **Draw**: when the smiley is sleeping then draw the sleeping smiley. **Update**: A sleeping smiley shouldn't move	Process the **mouse down event**. When the left mouse button is pressed, call the HitTest method of the smileys with the mouse position as parameter value. Use a helper function: HitTestSmileys, with the mouse position as parameter and call it in the event function.
Build, run and toggle the awake state of some smileys. Sleeping smileys should show the sleeping image frame and shouldn't move anymore. Awakened smileys should start moving again.  Smileys - Name, first name - 1DAEXX	
Smiley class	**Game class**

Define the **IsSleeping**, **SetHighest** and **GetPosition** methods. **Draw**: when the smiley is the highest one it should draw the happy smiley.	In the Update function, after the smileys got their new positions, ask each smiley its position and tell the one with the largest vertical position that it is the highest one. Only non-sleeping smileys can become highest. Helper function: DetermineHighestSmiley
Build, run and verify that only the highest smiley has a happy face. When they are all sleeping there should be no happy one. 	
Smiley class	**Game class**
Define the methods **IncreaseSpeed/ DecreaseSpeed.**	Process the **arrow up/down key-up event press**: Call the IncreaseSpeed / DecreaseSpeed methods of the smileys Helper functions: IncreaseSmileysSpeed DecreaseSmileysSpeed
Build and run. Verify that pressing these keys changes the speed of the smileys accordingly.	
Smiley class	**Game class**
-	Process the delete (SDLK_DELETE) key-up event: delete the sleeping smileys (helper function DeleteSleepers)
Build and run. Verify the correct working of this new functionality by pressing the DELETE key in various situations.	

4.4. MiniGame

4.4.1. PowerUp Manager

Go to the MiniGame assignment and implement part 3.2.

4.5. ShooterGame

4.5.1. Create the project

Add a new project with name **ShooterGame to the W02** solution

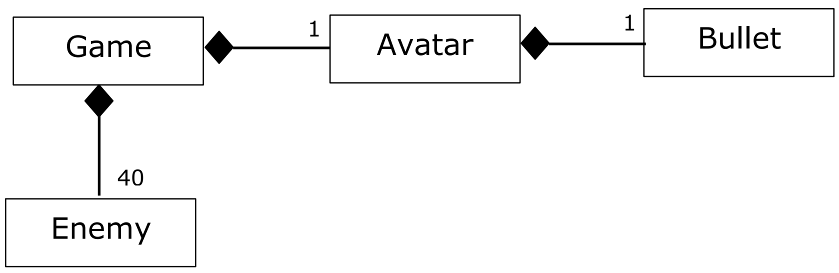
4.5.2. General

This exercise is intentionally not described in detail, because it is important that you start solving these kinds of exercises yourself.

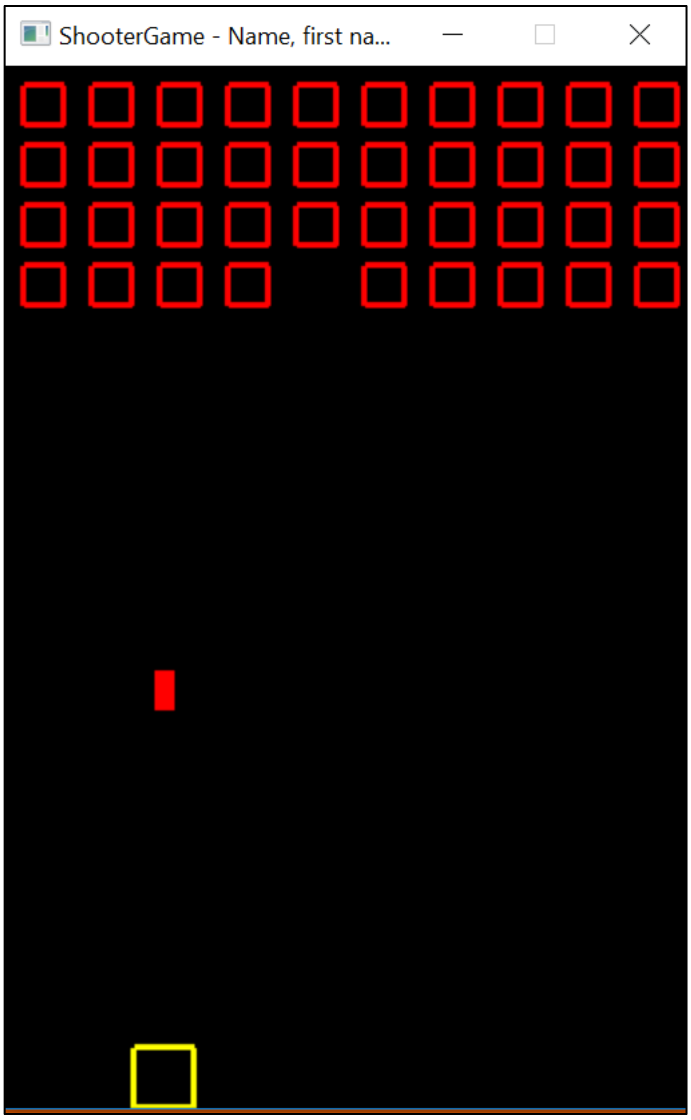
In this project you make a simple shooter game. At the top of the window, we have 40 red not moving square **enemies**.

At the bottom we have a yellow **avatar** that can move horizontally and that can shoot a **bullet**. When the bullet overlaps an enemy, the enemy is killed, and the bullet and this enemy disappear.

You define an Enemy, Avatar, and Bullet class. This UML indicates their relationship.



At the end of this exercise, you have a window like this.



4.5.3. Enemy

a. Define the class

In this game, the enemy has a red rectangular form. At the start it is alive, but dies when it is hit.

It offers following services:

- Draw: draws itself, when it is dead it doesn't draw anything.

- DoHitTest: returns true when the given rectangular object overlaps this enemy. In this case the enemy dies.

This is the UML diagram, decide yourself about the const methods.

Enemy

`- m_Center: Point2f`
`- m_Width: float`
`- m_Height: float`
`- m_IsDead: bool`

```
+ Enemy( )
+ Enemy(center: const Point2f&, width: float, height: float )
+ Draw( ): void
+ DoHitTest(other: const Rectf&  ): bool
+ SetCenter(center: const Point2f&  ): void
+ SetDimensions(width: float, height: float ): void
+ IsDead( ): bool
```

b. Do some basic testing

First do some basic tests in the Game class, after each step build, run and verify. Like this:

- Create one enemy object and draw it. Verify that it is drawn on the right position and with the right dimensions.
- In the mouse pressed event, call the DoHitTest with a Rectf object located at the mouse position and with a small width and height.
  Verify that when you click outside the enemy it doesn't disappear and when you click on it, it disappears.

c. 40 enemies

Previous test code is no longer necessary, now:

- Define an array of 40 Enemy object pointers

- In the Initialize method set their centers arranging them in 4 rows at the top of the window and give them the desired dimensions.
- Draw them

4.5.4. Avatar that moves

a. Define the class

The avatar has a yellow rectangular form and it moves horizontally with a given speed when the left/right arrow key is down (use SDL_GetKeyboardState). It stops at the window boundaries.

It offers following services:

- Draw: it draws itself

- Update: it changes its position when the move keys are down (Helper function: HandleMoveKeysState), staying within the boundaries (Helper function: Clamp)

This is the UML diagram, decide yourself about the const methods.

Avatar
- m_Center: Point2f - m_Width: float - m_Height: float - m_Speed: float - m_Boundaries: Rectf

```
+ Avatar( )
+ Avatar(center: const Point2f&, width: float, height: float )
+ Update(elapsedSec: float, pEnemies: Enemy*, numEnemies: int ): void
+ Draw( ): void
+ SetCenter(center: const Point2f&  ): void
+ SetDimensions(width: float, height: float ): void
+ SetBoundaries(boundaries: const Rectf& ): void
- Clamp( ): void
- HandleMoveKeysState(elapsedSec: float ): void
```

b. Create the avatar object

In the game class:

- Create an object of the Avatar class, position it in the middle at the bottom of the window
- Draw it
- Update it

Build, run and test this new functionality.

4.5.5. Bullet class

a. Define the class

A bullet has the form of a filled red rectangle, when it is shot it becomes activated and starts moving with the given velocity at the given position. When it leaves the window boundaries it becomes deactivated again.

It offers following services:

- **Draw**: only when it is activated, it draws itself
- **Shoot**: only when it is not activated yet, it becomes activated, gets the position as indicated by the center parameter and starts moving with the given velocity.
- **Update**: only when it is activated, it changes its position and when it leaves the boundaries (CheckBoundaries) it becomes deactivated again.

This is the UML diagram, decide yourself about the const methods.

Bullet
-m_Center: Point2f -m_Width: float -m_Height: float -m_Velocity: Vector2f -m_Boundaries: Rectf -m_IsActivated: bool
+ Bullet( ) + Bullet(width: float, height: float) + Draw( ): void + Update(elapsedSec: float, pEnemies: Enemy *, numEnemies: int) : void + Shoot(center: const Point2f&, velocity : const Vector2f&) : void + SetDimensions(width: float, height: float) : void + SetBoundaries(boundaries: const Rectf&) : void - CheckBoundaries( ): void

b. Create a Bullet object in the Avatar class

This needs an updates in the UML diagram

Avatar
… - m_Bullet: Bullet
…

c. Shoot the bullet

When the arrow-up pressed event occurs, shoot the bullet, give it as start position the position of the avatar.

Also Draw and Update the Bullet object in the Draw and Update method of the Avatar.

The updated UML diagram

Avatar
…
… + ProcessKeyDownEvent(e: const SDL_KeyboardEvent&): void

Don't forget to call the Avatar's ProcessKeyDownEvent in the Game class.

Build, run and verify this new functionality. When the bullet has left the boundaries, you should be able to shoot again the bullet.

4.5.6. Bullets kill the enemies

Now let's add the last functionality to our game. An enemy is killed by the bullet when they overlap. Therefor the bullet needs the enemies information when it is moving. We can pass the enemies to the avatar, that can pass them to the bullet. Change the Update method of both classes.

a. Avatar

Avatar
…
… + Update(elapsedSec: float, **pEnemies**: **Enemy***, **numEnemies**: **int**): void

b. Bullet

After having calculated its new position, it asks the enemies whether it overlaps them (CheckEnemiesHit). When it overlaps one of them, the bullet becomes deactivated again.

Bullet
…

```
…
+ Update(elapsedSec: float, pEnemies: Enemy*, numEnemies: int ): void
- CheckEnemiesHit(pEnemies: Enemy*, numEnemies: int ): void
```

Now our game is finished, feel free to use images for the enemies, bullet, avatar or to add extra functionality.

5. Submission instructions

You have to upload the folder *1DAExx_02_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_02_name_firstname* folder and upload it before the start of the first lab next week.

6. References

6.1. Classes in C++

http://www.learncpp.com chapter 8 parts 10, 11 and 12