Move semantics

1. Content

2. Objective

At the end of these exercises you should:

- Be able to define the move constructor of a class
- Be able to define the move assignment operator of a class
- Know when the move constructor is called
- Know when the move assignment operator is called

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 as comment at the top of each cpp file.

Give your **projects** the same **name** as mentioned in the title of the exercise, e.g. **RuleOf5Basics**. Other names will be rejected.

Create a blank solution W07.

3.1. RuleOf5Basics

3.1.1. Create the project

Add a new project with name **RuleOf5Basics** in the W07 solution.

Remove the generated file **RuleOf5Basics.cpp.**

Use the **Framework** files and allow to overwrite **pch.h.**

Copy and add **your** last version of the **Container** class files to this project. In case you don't have any, you can use the ones we provided.

Rearrange the code files in **Framework** and **Game** Filters.

Adapt the window title.

Building and running this project shouldn't give any problems.

3.1.2. Add move semantics to the Container class

Your Container class already has a copy constructor and an assignment operator that performs a deep copy instead of the default shallow copy.

Making a deep copy is not always necessary, for instance if the source Container object is destroyed immediately afterwards. Then the deep copy becomes an expensive copy.

Let's add some code that makes unnecessary deep copies and then adhere the Container class to the rule of 5.

a. Need of a move constructor

Declare and define following helper function in the Game class. It creates a Container object c, adds a given number of elements with a random value within a given interval [min,max] and returns the created Container object.

```
Container Game::CreateContainer(int nrElements, int min, int max)
{
    Container c{ nrElements };
    for (int i{ }; i < nrElements; ++i)
    {
        c.PushBack(rand() % (max - min + 1) + min);
    }
    return c;
}
```

Again declare and define a TestContainer function and call it in the Game:Initialize method.

```
void Game::TestContainer()
{
    std::cout << "--> Test that demonstrates the need of Move constructor\n";
    Container c1{ CreateContainer(20,10,20) };
}
```

Build, run and step through the code surrounded by the red rectangle. How many integers are allocated on the heap during execution of this code?

Notice that a deep copy is made of a source Container object that is destroyed immediately after the copy. That's a waste.

Solve this by adding the **move constructor** to the Container class, step again through this code and notice that now a shallow copy is made, the array pointer is moved the argument object has been emptied before it is destroyed, and this

works fine. Now only **20 integers** should be **allocated dynamically**, verify this by stepping through your code.

b. Need of a move assignment operator

Extend the TestContainer method with a second test.

```
void Game::TestContainer()
{
    std::cout << "--> Test that demonstrates the need of Move constructor\n";
    Container c1{ CreateContainer(20,10,20) };

    std::cout << "--> Test that demonstrates the need of Move assignment operator\n";
    c1 = CreateContainer(10, 20, 30);
}
```

Build, run and step through the code indicated by the red rectangle. How many integers are allocated dynamically during execution of the code surrounded by the red rectangle.

Notice that again a deep copy is made of a Container object that is destroyed immediately afterwards. So again, a waste.

Solve this problem, add the **move assignment operator** to the Container class, execute the code again and notice that now a shallow copy is made, the argument is emptied before its destructor is executed and that everything works fine.

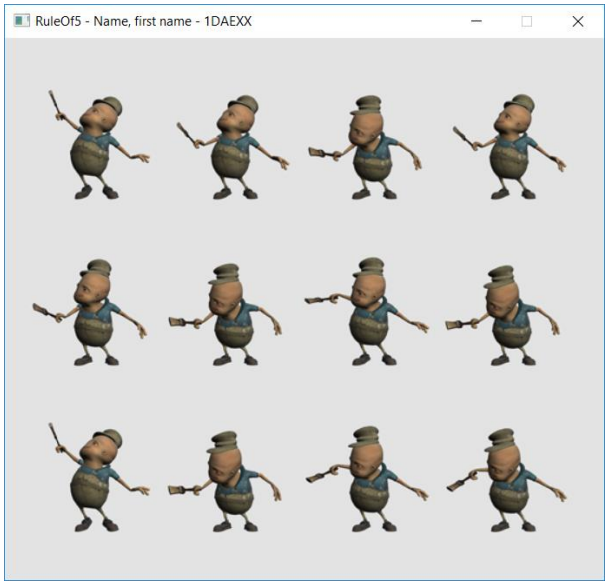
Now only **10 integers** should be **allocated dynamically** during execution of the code inside the red rectangle**, verify this by steppin**g through your code.

c. Feedback

Ask the lecturer to verify the move code of your Container class.

3.1.3. Sprite class

Remember that in programming 1, you defined a Sprite class with the animated Tibo? Add the given Sprite class files to the project.



a. Rule of 5

This Sprite class does not adhere to the rule of 5. Define and implement the copy and move constructors / assignment operators.

Define a function TestSpriteClass in which you test this new Sprite functionality.

Remember that a Texture object can not be copied, check its declaration. You will need to add a string member variable to the Sprite class to avoid the copy operation.

Add console messages to the 4 copy/move methods so that you can check out when they are called.

In TestSpriteClass, add this code (see codefiles for source):

```
std::cout << "\n--> Sprite class: Test rule of 5 started\n";
{
    Sprite sprite1{ "Resources/RunningKnight.png", 8, 1, 0.08f };
    Sprite sprite2{ "Resources/RunningKnight.png", 8, 1, 0.08f };
    Sprite sprite3{ sprite1 };
    sprite2 = sprite1;
    Sprite sprite4{ CreateSprite("Resources/Tibo.png", 5, 5, 1 / 10.f) };
    sprite1 = CreateSprite("Resources/Tibo.png", 5, 5, 1 / 10.f);
    std::cout << "std::move: ";
    sprite2 = std::move(sprite1);
}
std::cout << "--> Sprite class: Test rule of 5 ended\n";
```

Before running, try to guess what functions will be called for each line of code. Write down your guess as a line comment. Then run and check your guesses. Try to understand why each function was called.

b. 12 Sprite objects

This is an extra exercise on creating objects, saving these objects in an stl vector and draw them in a grid.

The interesting part here is how we create the objects and push them in the container.

Add a member std::vector that can store Sprite objects.

Add a member function CreateSprites. In a loop, add m_NrRows * m_NrCols sprite objects to the container like this:

```
m_Sprites.push_back(Sprite{ "Resources/Tibo.png",5,5,1.f / (rand() % (16 - 10 + 1) + 10) });
```

Run and watch the console. You notice how many objects are being copied, moved and deleted? Can you guess why? Also print the capacity of the container and run again.

Now mark the move functions as noexept (both in declaration as in definition), and what do you notice? More efficient!

Now, set the capacity to m_NrRows * m_NrCols before the loop starts. (reserve)

Notice how much more efficient this is?

This is the best we can get, but still we have a move and a destructor function being called.

The only way to make this even **more efficient** is to **store pointers** in the **container**, and to make manually take care of the lifetime thus making the Sprite objects using dynamic memory allocation.

This is the recommended way to work with containers: do only store basic built-in types, and (smart) pointers.

Continue by refactor your code to store pointers. Admire the result on the console and see that destructors are no longer being fired.

Continue by drawing these sprites too.

Check for memory leaks!

3.1.4. What about the framework classes?

Have a look at the Core, Game and Texture classes.

Do they have a destructor?

And if so, do they apply the rule of 5?

3.1.5. MiniGame

Check all classes of the minigame and make sure they apply the rule of 5.

4. References

4.1. Rule of five

https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)

http://en.cppreference.com/w/cpp/language/rule_of_three