Copy Semantics – Rule of Three

1. Content

2. Objective

At the end of these exercises you should:

- Be able to define the copy constructor of a class and know when you need to define it
- Be able to define the assignment operator of a class and know when you need to define it
- Know when the copy constructor and assignment operator are 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.

Create an empty solution W05 in the **1DAExx_05_name_firstname** folder.

Give your **projects** the same **name** as mentioned in the title of the exercise, e.g. **RuleOf3Basics**. Other names will be rejected.

3.1. RuleOf3Basics

3.1.1. Create the project

Create a new framework project with name **RuleOf3Basics** in your W05 solution folder.

Remove the generated file **RuleOf3Basics.cpp.**

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

Copy and add the provided last version of the **Container** class files to this project. It does **on purpose** not follow the rule of three to make you experience what can go wrong.

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

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

3.1.2. Questions and answers

While making the exercises, answer the questions listed in the **Q&A Document**.

3.1.3. Container class

The Container class still has some serious shortcomings that can lead to runtime errors. In next exercise you'll add some test code that results in these runtime errors.

First let's declare and define in the **Game** class a private function **AddValues** that adds a requested number (*nr*) of random values in the inclusive interval [*min*, *max*] to a container (*c*).

```
void AddValues(Container &c, int nr, int min, int max);
```

Then declare and define a function **TestContainer** and call it in the Game::Initialize function, and add code to this function as described in following paragraphs.

3.1.4. Creating a Container object from an existing one

a. Code that yields a runtime error

Add next code snippet to the TestContainer definition.

```
void Game::TestContainer( )
{
    Container c1{};
    AddValues( c1, 3, 1, 10 );

    Container c2{ c1 };
}
```

The first line creates an object with automatic duration in a function, so it will be a stack object. The second line adds 3 random values in the interval [1, 10] to it. But the 3rd statement might be new, it creates a second Container object c2 passing the first Container object as a parameter, while we only defined a Container constructor that accepts an optional capacity value as argument, we don't get any build errors! Isn't that very weird?

But when you run the application, it crashes.

Investigate this crash and answer the questions listed under (1) in the Q&A Document.

Before solving this problem, let's have a closer look at the content of the containers just before the crash. Define a breakpoint at the closing curly brace of the function TestContainer(). Run again.

Answer the questions listed under (2) in the Q&A Document.

Before solving this problem, let's add another test. Change one of the elements of the first container after the creation of the second one. Keep the breakpoint activated.

```
void Game::TestContainer( )
{
    Container c1{};
    AddValues( c1, 3, 1, 10 );

    Container c2{ c1 };
    c1[0] = 20;
}
```

Print the value of the first element of each container.

Have another look at the elements in both containers and answer the questions listed under (3) in the Q&A Document.

You should understand this behavior, ask you teacher for more info if you don't.

This surely isn't the behavior one expects from a Container class.

b. Solve these problems

Define the **copy constructor,** it should solve these problems. Build and run the application, the test code should no longer result in a runtime error.

Before continuing, have a closer look at the content of the containers just before they go out of scope, thus at the closing curly brace of the TestContainer function.

Answer the questions listed under (4) in the Q&A Document.

3.1.5. Assigning a Container object to another one

a. Assigning also results in a runtime error

The Container class isn't ok yet. It still can result in a runtime error.

Change the TestContainer function as indicated in next screenshot.

```
void Game::TestContainer( )
{
    Container c1{};
    AddValues( c1, 3, 1, 10 );

    Container c2{ c1 };
    c1[0] = 20;

    Container c3{ 3 };
    AddValues( c3, 3, 10, 20 );
    c3 = c1;
}
```

In the added code snippet, a third Container object with a capacity for 3 elements is created. Then 3 random values in the interval [10, 20] are added to it and Container object c1 is assigned to Container object c3. The compiler accepts this, no build errors. Deactivate the previous breakpoint (Right Mouse Button on break point). When you run this code, the application crashes again.

Investigate this crash and answer the questions listed under (5) in the Q&A Document.

Before solving this problem, again let's have a detailed look at the content of the containers just before the crash. Activate the breakpoint at the closing curly brace of the function TestContainer, so before the containers go out of scope and have a look at the content of c1 and c3.

Answer the questions listed under (6) in the Q&A Document.

b. Solve this problem

Now solve the problem, define the **assignment operator**.

Running the application should no longer give a runtime error.

Set again a breakpoint at the closing curly brace of the function TestContainer. And verify that the arrays of c1 and c3 are located at a different memory location.

Also verify that you no memory leaks are reported when quitting the application.

c. Self-assignment

What happens when you assign a Container object to itself? Add some code to the TestContainer function.

```
void Game::TestContainer( )
{
    Container c1{};
    AddValues( c1, 3, 1, 10 );

    Container c2{ c1 };
    c1[0] = 20;

    Container c3{ 3 };
    AddValues( c3, 3, 10, 20 );
    c3 = c1;

    c3 = c3;
}
```

Look at the content of c3's **m_pElement** data member **before** and **after** this assignment statement. It doesn't make sense to create a new array, copy all the elements and destroy the previous array, when an object is assigned to itself. Prevent this self-assignment.

3.1.6. Converting constructor

Add following code line to the TestContainer function.

```
void Game::TestContainer( )
{
    Container c1{};
    AddValues( c1, 3, 1, 10 );

    Container c2{ c1 };
    c1[0] = 20;

    Container c3{ 3 };
    AddValues( c3, 3, 10, 20 );
    c3 = c1;

    c3 = c3;

    c3 = 4;
}
```

Notice that this gives neither build errors nor runtime errors.

But what happens when this code is executed?

Set a breakpoint at this code line and follow its execution using the debugger and answer questions listed under (7) in the Q&A Document.

This kind of assignment doesn't make sense and it was probably not the intention of the programmer to create a new container with a capacity of 4, so make this implicit conversion impossible. Now this assignment should result in a compiler error.

3.1.7. Copy constructor and assignment operator calls

In previous exercises it was clear when the 1-argument constructor, copy constructor, assignment operator and destructor were called. But this is not always the case. Let's examine this in more detail so that you are able to predict what happens behind the scenes when seeing these kinds of statements.

Add next code line in the TestContainer function.

```
void Game::TestContainer( )
{
    Container c1{};
    AddValues( c1, 3, 1, 10 );

    Container c2{ c1 };
    c1[0] = 20;

    Container c3{ 3 };
    AddValues( c3, 3, 10, 20 );
    c3 = c1;

    c3 = c3;

    //c3 = 4;

    Container c4 = c1;
}
```

Set a breakpoint at this new code line and follow its execution using the debugger and answer questions listed under (8) in the Q&A Document.

Then add this code line

```
c4 = c2;
```

Set a breakpoint at this new code line and follow its execution using the debugger and answer questions listed under (9) in the Q&A Document.

Then define a function that multiplies the elements in Container object c with a factor and then returns this changed Container object.

```
Container Game::CreateMultiplied( Container c, int factor )
{
    for ( int idx{ 0 }; idx < c.Size( ); ++idx )
    {
        c[idx] *= factor;
    }
    return c;
}
```

Then call this function in the TestContainer function, like this:

```
c4 = CreateMultiplied( c1, 2 );
```

Set a breakpoint at this new code line and follow its execution using the debugger and answer questions listed under (10) in the Q&A Document.

Then call the AddValues function, like this:

```
AddValues( c4, 3, 20, 30 );
```

Set a breakpoint at this new code line and follow its execution using the debugger and answer questions listed under (11) in the Q&A Document.

3.1.8. Some extra tests

Add following statements at the end of the TestContainer function.

```
AddValues(c1, 20, 0, 10);
AddValues(c2, 20, 0, 10);
AddValues(c3, 20, 0, 10);
AddValues(c4, 20, 0, 10);
```

Verify that building and running works fine.

3.1.9. Texture class

a. Texture as a static object

Be aware that creating a static Texture object doesn't work.

Try it.

In Game.h

```
private:
    // DATA MEMBERS
    Window m_Window;

    static Texture m_StatDaeTexture;
```

And in Game.cpp

```
#include "pch.h"
#include "Game.h"
#include "Container.h"
#include <iostream>

Texture Game::m_StatDaeTexture{ "Resources/DAE.png" };
```

When you draw this texture, a white rectangle is drawn.

```
void Game::Draw( ) const
{
    ClearBackground( );
    m_StatDaeTexture.Draw(Point2f{ 0.0f,50.0f });
}
```

Add break points to figure out when the texture is initialized.

Answer questions listed under (12) in the Q&A Document.

Also notice the warning in the console.

b. Assigning a Texture object to another one

Define and initialize a non-static Texture object with automatic duration.

```
private:
    // DATA MEMBERS
    Window m_Window;
    static Texture m_StatDaeTexture;
    Texture  m_DaeTexture;
```

```
Game::Game( const Window& window )
    :m_Window{ window }
    , m_DaeTexture{ "Resources/DAE.png" }
{
```

Assign it to the static data member in Game::Initialize. This assignment results in the error "attempting to reference a deleted function".

```
void Game::Initialize( )
{
    m_StatDaeTexture = m_DaeTexture;
}
```

Answer question listed under (13) in the Q&A Document.

Then comment this code line that results in an error.

c. Copying a Texture object

Declare and define a function that draws a given Texture object a given number of times, starting at a given position, like this.

```
void Game::DrawTexture( Texture texture, const Point2f& pos, int nr, int dx, int dy) const
{
    for (int i{ 0 }; i < nr; ++i)
    {
        texture.Draw(Point2f{ pos.x + i * dx, pos.y + i * dy });
    }
}
```

Then call this function in the Draw function – where else - to draw the DAE Texture object m_DaeTexture 4 times on the window.

You get an error "attempting to reference a deleted function". Solve this error without changing the Texture class.

Answer questions listed under (14) in the Q&A Document.

3.2. MiniGame

Go to the Minigame assignment and implement part 3.5.

In this part you can:

- Add a **platform** to stand on and jump through.
- **End Game** sign, touching it ends the game.

Go to the Minigame assignment and implement part 3.6.

In this part you can add a **HUD**

Go to the Minigame assignment and implement part 3.7.

In this part you add **sound**.

3.3. Smileys

The Smileys project has been created with the Programming 1 framework in which **Game::Draw** was not const. Let's see what the consequence is of defining this function as being const.

Copy and add your **Smileys** project in the W05 solution.

Change Game::Draw, indicate that it is const and solve the build errors you get because of this change.

Make the class Smiley follow the rule of three and prevent that classes can be derived from the Smiley class.

3.4. TowerOfHanoi

3.4.1. Create the project

Create a new project with name **TowerOfHanoi** in your W05 solution.

Remove the generated file **TowerOfHanoi.cpp.**

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

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

Adapt the window title.

3.4.2. General

In this project you create the Tower of Hanoi game Tower of Hanoi

You must implement two classes: **Disk** and **Tower**.

You decide about the UI but try to make it user friendly. You can use textures or just draw rectangles to represent the disks and towers. The main focus is the class interactions.

There is an executable that you can use as example. It was so much fun, we let ourselves go on this one.

4. Submission instructions

You have to upload the compressed folder *1DAExx_05_name_firstname.* This folder contains 3 solution folders:

- W05
- MiniGame
- Name_firstname_GameName which also contains your up-to-date report file.

Don't forget to clean the projects before closing them in Visual Studio.

5. References

5.1. Rule of three

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

http://www.drdobbs.com/c-made-easier-the-rule-of-three/184401400

http://stackoverflow.com/questions/4172722/what-is-the-rule-of-three