Templates – sequence containers

1. Content

Templates – sequence containers 1
1. Content 1
2. Objective 2
3. Exercises 2
3.1. FunctionTemplate 2
3.1.1. Create the project 2
3.1.2. Make a templated function 3
3.2. ContainerTemplate 3
3.2.1. General 3
3.2.2. Create the project 3
3.2.3. Initializer-list 3
3.2.4. Transform Container into a template 4
3.2.5. Adapt the TestInitializerList function 4
3.2.6. Test the container template using the given test code 4
3.2.7. Instantiate and use a container of Vector2f types 5
3.3. SinglyLinkedList 5
3.3.1. Project 5
3.3.2. General 5
3.3.3. Node struct and LinkedList class 6
3.3.4. Constructor and operator << 6
3.3.5. PushFront 7
3.3.6. Destructor 8
3.3.7. InsertAfter 8
3.3.8. PopFront method 9
3.3.9. Remove method 10
3.4. TrailingCircles 11
3.4.1. Project 11
3.4.2. Adapt the TrailingCircle class 12
4. Submission instructions 13
5. References 13
5.1. std::initializer_list class 13
5.1.1. std::initializer_list 13
5.1.2. Size member function 13

5.2. Sequence containers.......... 13
5.2.1. Vector.......... 13
5.2.2. Deque.......... 13
5.3. Templates.......... 13

2. Objective

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

- Define a function template
- Define a class template
- Use the templated std::initializer list class
- Define a manager class that manages a container of objects
- Define a singly linked list.
- Use the sequence containers of the standard template library, such as the std::deque, std::vector

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

3. Exercises

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

Always adapt the **window title**: it should mention the name of the project, your name, first name and group.

Create a blank solution with name **W09** in your 1DAExx_09_name_firstname folder

3.1. FunctionTemplate

3.1.1. Create the project

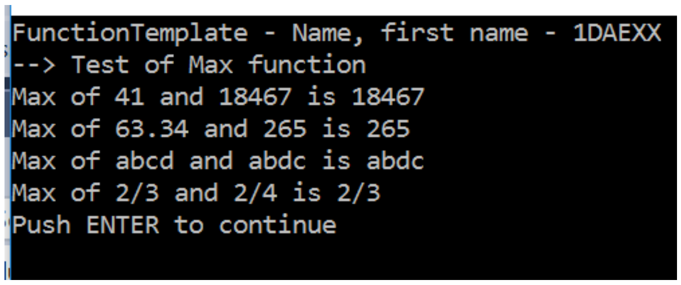
Add a new **project** with name **FunctionTemplate** to the W09 solution.

Delete the generated file **FunctionTemplate.cpp**.

Copy and add the given code files you find in the folder **CodeFiles/01_FunctionTemplate** to the project.

Adding the framework files is not needed.

Building and running the application shouldn't result in any errors. This is printed.



```
FunctionTemplate - Name, first name - 1DAEXX
--> Test of Max function
Max of 41 and 18467 is 18467
Max of 63.34 and 265 is 265
Max of abcd and abdc is abdc
Max of 2/3 and 2/4 is 2/3
Push ENTER to continue
```

Change the message on the console, show your name and group.

3.1.2. Make a templated function

Have a look at the code. The **MaxFunctions** code files contain the declarations and definitions of 4 **Max** functions, which all contain the same code definition, only the parameter types are different. These functions are called in the main function.

Comment these 4 functions declarations and definitions and replace them by 1 function template, build and launch the project again. You should get the same results.

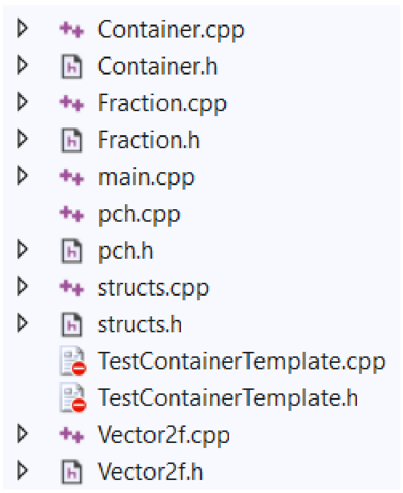
3.2. ContainerTemplate

3.2.1. General

In this project you'll transform the given Container class into a templated class.

But first you'll expand this Container class with an extra constructor, one that takes a parameter of type std::initializer_list.

3.2.2. Create the project



Container.cpp
Container.h
Fraction.cpp
Fraction.h
main.cpp
pch.cpp
pch.h
structs.cpp
structs.h
TestContainerTemplate.cpp
TestContainerTemplate.h
Vector2f.cpp
Vector2f.h

Add a new **project** with name **ContainerTemplate** to the W09 solution and set it as StartUp Project.

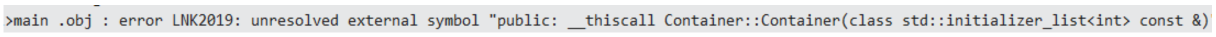
Delete the generated file **ContainerTemplate.cpp**

Copy the code files from the folder **CodeFiles/02_ContainerTemplate** in this project folder.

Add all these files - **except the TestContainerTemplate files** - to the project. The TestContainerTemplate files will be used once the Container class is transformed into a template class. As long as this is not done, you'll get build errors if you add these 2 files as well.

3.2.3. Initializer-list

However building this project leads to a linker error, because the symbol Container::Container(class std::initializer_list<int>) is unresolved.



```
>main .obj : error LNK2019: unresolved external symbol "public: __thiscall Container::Container(class std::initializer_list<int> const &)
```

Have a look at the given Container class. It contains a constructor declaration that has a parameter of type **std::initializer_list**, but it isn't defined yet. This constructor allows you to create a Container instance using a braced-init-list, like this:

```
        Container *pCont1{new Container{ 1,2,3,4,5 }};
```

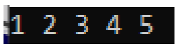
And this constructor is used in the function TestInitializerList() in main.cpp. That's why you get this error.

Add the definition of this constructor in Container.cpp:

- Give the container a capacity that is equal to the size of the given list: Size member function

```
    for (int element : list) { ... }
```

Build and run the project. The 5 numbers are printed:



```
1 2 3 4 5
```

Also check that there are no memory leaks when stopping the application.

3.2.4. Transform Container into a template

Transform the Container class definition - which handles a dynamic array of int-type elements - into a template that can handle a dynamic array of **any type** of elements. However only for element types that satisfy the following requirements:

- Have a default constructor

- Are copy-able

3.2.5. Adapt the TestInitializerList function

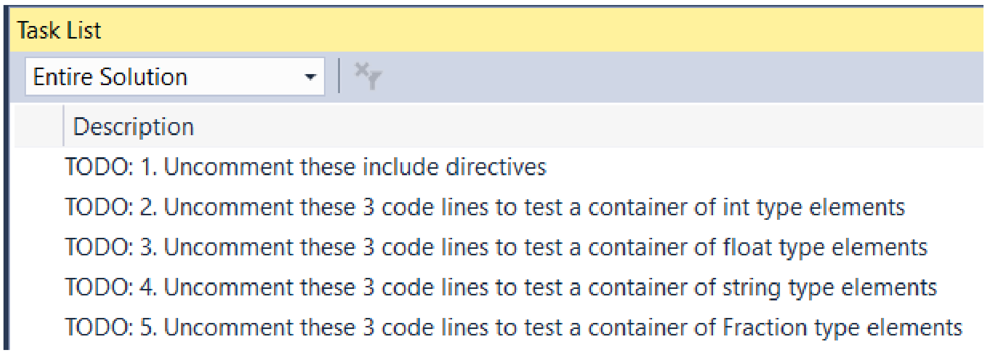
Now this function results in some errors. Adapt the code in this function to solve these errors, indicate that a Container of int types is used in these tests.

3.2.6. Test the container template using the given test code

Add the **TestContainerTemplate code files** to the project. These files contain several Container tests which are called in main.cpp, but the calls are still commented. These tests instantiate and test the Container template for following element types:

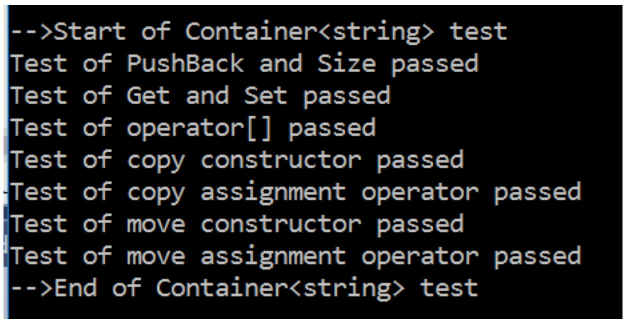
- int
- float
- string
- Fraction

Uncomment and launch the tests **ONE BY ONE**. You can use the Task List window, they are marked as the tasks 1 to 5, double clicking on a task in the Task List leads you to corresponding line in de code window. **If a test fails, then adapt YOUR code and do not change the test code.**



TODO: 1. Uncomment these include directives
TODO: 2. Uncomment these 3 code lines to test a container of int type elements
TODO: 3. Uncomment these 3 code lines to test a container of float type elements
TODO: 4. Uncomment these 3 code lines to test a container of string type elements
TODO: 5. Uncomment these 3 code lines to test a container of Fraction type elements

Below is a screenshot of the test ran for a Container of std::string element types.



```
-->Start of Container<string> test
Test of PushBack and Size passed
Test of Get and Set passed
Test of operator[] passed
Test of copy constructor passed
Test of copy assignment operator passed
Test of move constructor passed
Test of move assignment operator passed
-->End of Container<string> test
```

Verify that all tests passed and that there are no memory leaks.

3.2.7. Instantiate and use a container of Vector2f types

In TestInitializerList you can see that container objects are created without the new operator. This will be extensively discussed in one of the future lessons. They are created the same way Point2f and Vector2f objects are created, not using dynamic memory allocation. Because they are created in a function, they have a limited scope and are located on the stack, not on the heap. The next task asks you to create some containers, do this similar as in TestInitializerList.

Complete the remaining tasks to test a Container of Vector2f they are situated in the function TestVector2fContainer. Task numbers are 6 to 12.

3.3. SinglyLinkedList

3.3.1. Project

Add a new project with name **SinglyLinkedList** to the **W09** solution.

Delete the generated file **SinglyLinkedList.cpp.**

Framework files are not needed.

Copy and add the given code files located in CodeFiles/04_SinglyLinkedList (**LinkedList.h**/**cpp** and **TestLinkedList.cpp)** to the project.

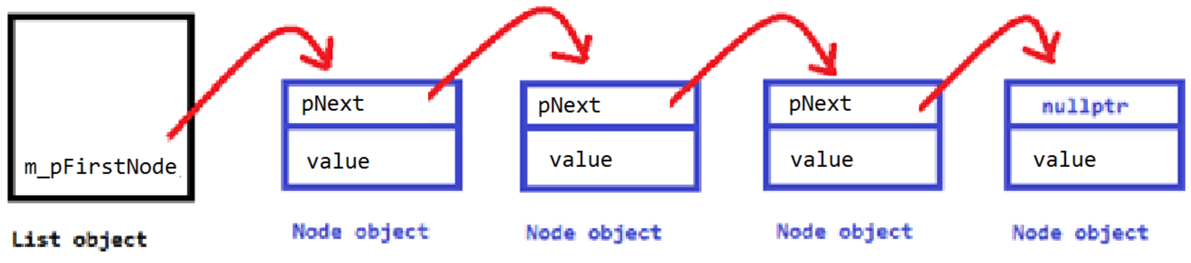
3.3.2. General

In this exercise you'll define a LinkedList class. It is a sequence container, meaning that each element has a certain position in the container, however the elements are not positioned next to each other in memory like in arrays or std::vector. They are distributed randomly in the memory and each element contains the address of the next element in the list, in this way chaining the elements of the list. That's why a value of the list is wrapped in a so-called **Node** object, which contains next to the value also the address of the next element in the list. The last node contains the value nullptr as next address indicating that this is the end of the list.

Thus a node has 2 data members:

- **pNext**: a pointer to the next node in the chain. When it is the last Node object in the list, then this variable contains the value nullptr.
- **value**: containing the data

The List itself contains the address of the first node in the list as the data member **m_pFirstNode.** When the list is empty, this variable has the value nullptr.



The data member **m_Size** contains the number of elements (Node objects) in the list.

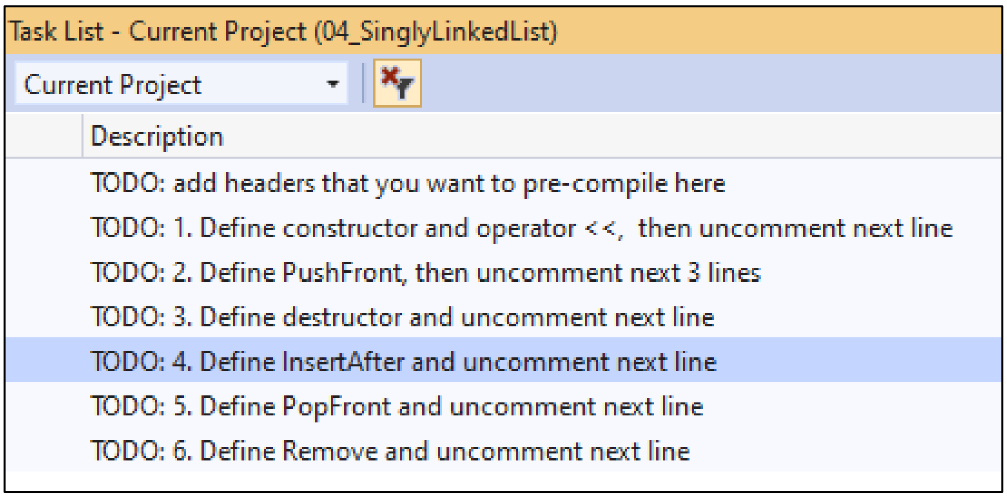
3.3.3. Node struct and LinkedList class

The files **LinkedList.h / cpp** are given. They contain the definition of the Node struct and LinkedList class. However, the LinkedList method definitions are empty. You complete them one by one as described hereafter.

The file **TestLinkedList.cpp** contains tests to verify your code. Follow the sequence as mentioned in the Task list. Following paragraphs follow the same sequence and describe what the methods should do. **If a test fails, then adapt YOUR code and do not change the test code.**

When a test leads to a crash, the **Call Stack** is a great help in locating the source of the problem.

Work **step by step** as indicated in the task list. Each step is described in more detail in the following sections. Don't comment again the succeeded tests, otherwise the possibility exist that some tests will go wrong as some test use the results of previous tests.


Task List - Current Project (04_SinglyLinkedList)

Current Project

Description
TODO: add headers that you want to pre-compile here
TODO: 1. Define constructor and operator <<, then uncomment next line
TODO: 2. Define PushFront, then uncomment next 3 lines
TODO: 3. Define destructor and uncomment next line
TODO: 4. Define InsertAfter and uncomment next line
TODO: 5. Define PopFront and uncomment next line
TODO: 6. Define Remove and uncomment next line

3.3.4. Constructor and operator <<

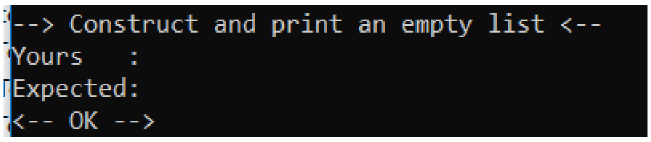
The **constructor** initializes the data members.

The **stream insertion operator** prints the values of the list separated by a space. Implement this method **without using the Size()** method of the linked list. Just write a loop that walks through the list of nodes until a node with pNext equals nullptr is reached.

After having defined these 2 methods, launch the first TODO test mentioned in **TestLinkedList**.

```
TODO: Define constructor and operator <<,  then uncomment next line
```

This should be the output.



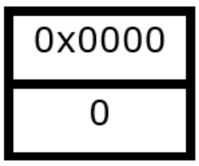
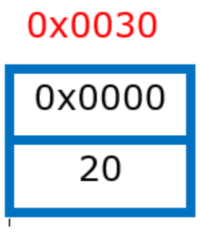
```
--> Construct and print an empty list <--
Yours    :
Expected:
<-- OK -->
```

3.3.5. PushFront

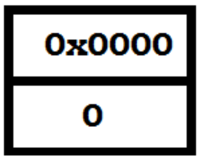
This function adds an element at the beginning of the list.

- It first wraps the element in a Node object
- It then adds this node in front of the list.
- It increments the size of the list

To clarify some things, let's add 2 values to the list and make a drawing of it. In these drawing we use following legend for the LinkedList object (black border) and Node objects (blue border)

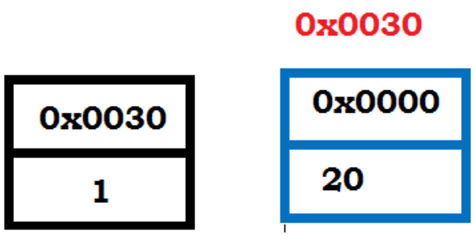
 `Node* m_pFirstNode;` `size_t m_Size;`	The **LinkedList object** has 2 cells, and you recognize it at the black border. The first cell contains the value of **m_pFirstNode** and the second cell the content of **m_Size**.
 `Node* pNext;` `int value;`	The **Node** objects also have 2 cells, however they have a blue border. The first cell contains the **pNext** content, the second cell the value data member. Above these 2 cells is indicated in red the address of this Node object in the memory.

a. The list with no elements



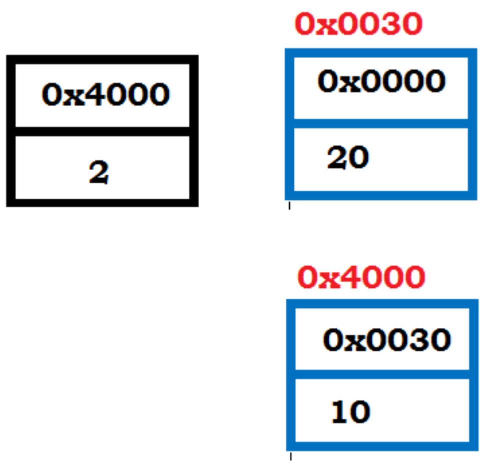
b. Add value 20 to the list

The value is wrapped in a new Node object. Suppose that the address of the Node object is 0x0030. The picture below shows the list after calling PushFront(20).

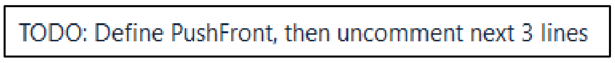


c. Add value 10 to the list

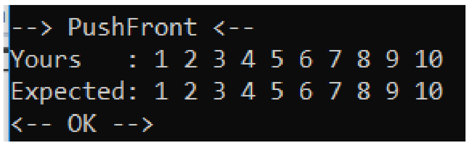
The picture below shows the list after calling PushFront(10). The value 10 is wrapped in a Node that is located at the address 0x4000.



Complete the definition of this function, then uncomment and launch the related TODO test.


```
TODO: Define PushFront, then uncomment next 3 lines
```

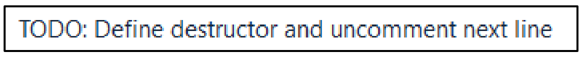
The test prints this, verify that "Yours" has the same elements as "Expected".


```
--> PushFront <--
Yours    : 1 2 3 4 5 6 7 8 9 10
Expected: 1 2 3 4 5 6 7 8 9 10
<-- OK -->
```

Notice that memory leaks are reported in the output window. Indeed, this class needs a destructor as it creates objects (nodes) on the heap.

3.3.6. Destructor

Define the destructor, delete all the nodes in the list. Then uncomment and launch the related TODO test.


```
TODO: Define destructor and uncomment next line
```

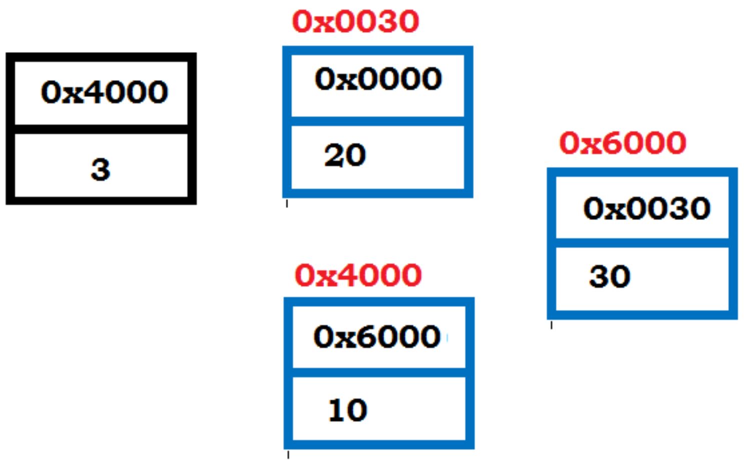
Verify in the output window that memory leaks are no longer reported.

3.3.7. InsertAfter

This method wraps the given *value* in a Node object and then inserts this new Node object after the given one: *pBefore.*

a. Insert value 30 after the first node of the list

The picture below shows the linked list after calling InsertAfter(0x4000, 30). The value 30 is wrapped in a Node located at the address 0x6000 and inserted at the requested position in the list.

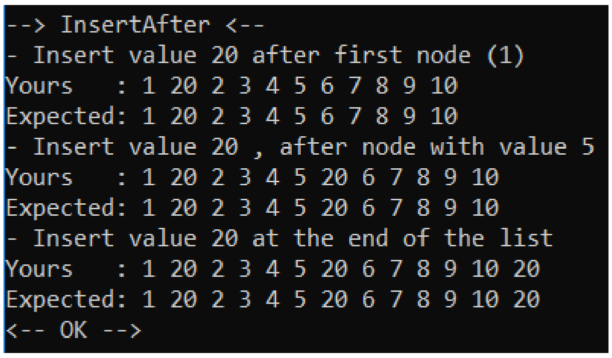


The new Node becomes the second one in the list.

Complete the definition of this function using the above drawings as a reference and uncomment and launch the related TODO test.

```
TODO: Define InsertAfter and uncomment next line
```

The test prints this, verify that "Yours" has the same elements as "Expected".

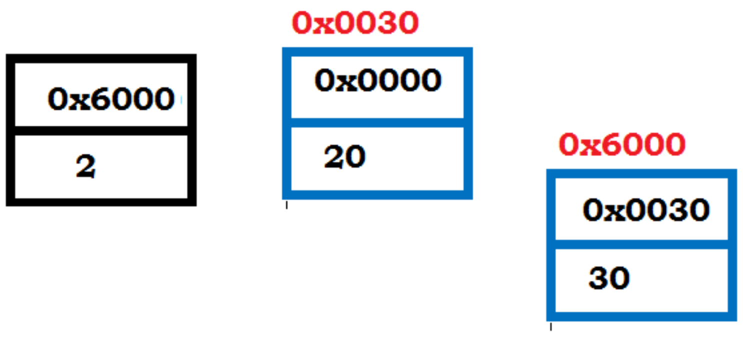


```
--> InsertAfter <--
- Insert value 20 after first node (1)
Yours    : 1 20 2 3 4 5 6 7 8 9 10
Expected: 1 20 2 3 4 5 6 7 8 9 10
- Insert value 20 , after node with value 5
Yours    : 1 20 2 3 4 5 20 6 7 8 9 10
Expected: 1 20 2 3 4 5 20 6 7 8 9 10
- Insert value 20 at the end of the list
Yours    : 1 20 2 3 4 5 20 6 7 8 9 10 20
Expected: 1 20 2 3 4 5 20 6 7 8 9 10 20
<-- OK -->
```

Verify that no leaks are reported in the output window.

3.3.8. PopFront method

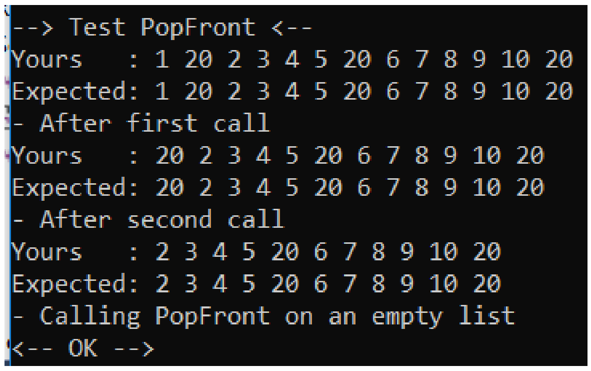
This method removes the first node from the chain and deletes that node. Next picture shows the list – as it was in the previous step – however after the PopFront method has been called.



Define the method, then uncomment and launch the related test.

```
TODO: Define PopFront and uncomment next line
```

The test prints this, verify that "Yours" has the same elements as "Expected".



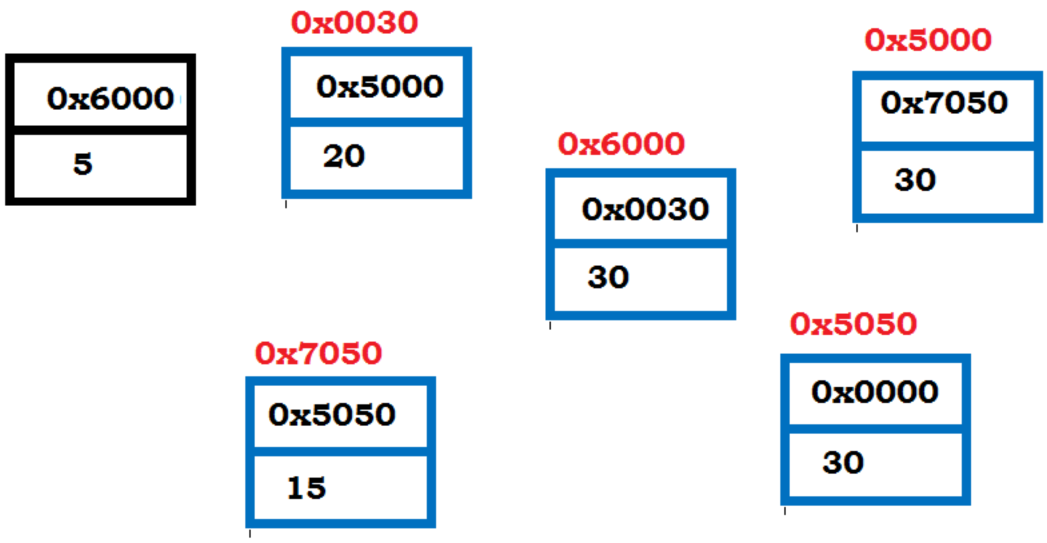
```
--> Test PopFront <--
Yours   : 1 20 2 3 4 5 20 6 7 8 9 10 20
Expected: 1 20 2 3 4 5 20 6 7 8 9 10 20
- After first call
Yours   : 20 2 3 4 5 20 6 7 8 9 10 20
Expected: 20 2 3 4 5 20 6 7 8 9 10 20
- After second call
Yours   : 2 3 4 5 20 6 7 8 9 10 20
Expected: 2 3 4 5 20 6 7 8 9 10 20
- Calling PopFront on an empty list
<-- OK -->
```

Verify that no leaks are reported in the output window.

3.3.9. Remove method

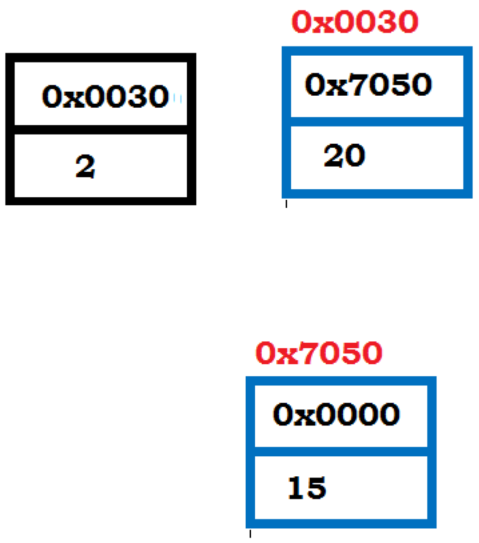
This method removes all nodes from the chain that have the given value and deletes them.

Suppose the list has following nodes.

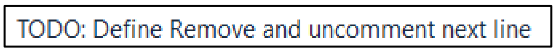


Thus, this is the sequence of the values: **30**, 20, **30**, 15, **30** and it contains 3 times the value 30.

After calling **Remove(30)** the list should have this content.

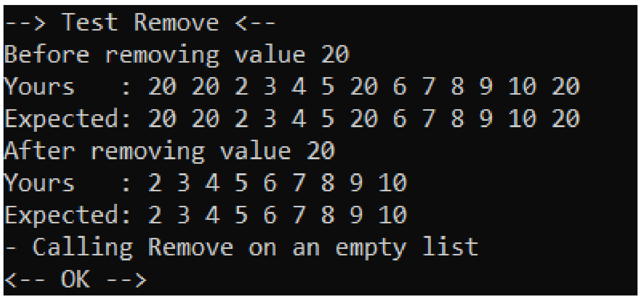


Complete the definition of this function using the above drawings as a reference and uncomment and launch the related test.



TODO: Define Remove and uncomment next line

The test prints this, verify that “Yours” has the same elements as “Expected”.



```
--> Test Remove <--
Before removing value 20
Yours   : 20 20 2 3 4 5 20 6 7 8 9 10 20
Expected: 20 20 2 3 4 5 20 6 7 8 9 10 20
After removing value 20
Yours   : 2 3 4 5 6 7 8 9 10
Expected: 2 3 4 5 6 7 8 9 10
- Calling Remove on an empty list
<-- OK -->
```

Verify that no memory leaks are reported in the output window.

3.4. TrailingCircles

3.4.1. Project

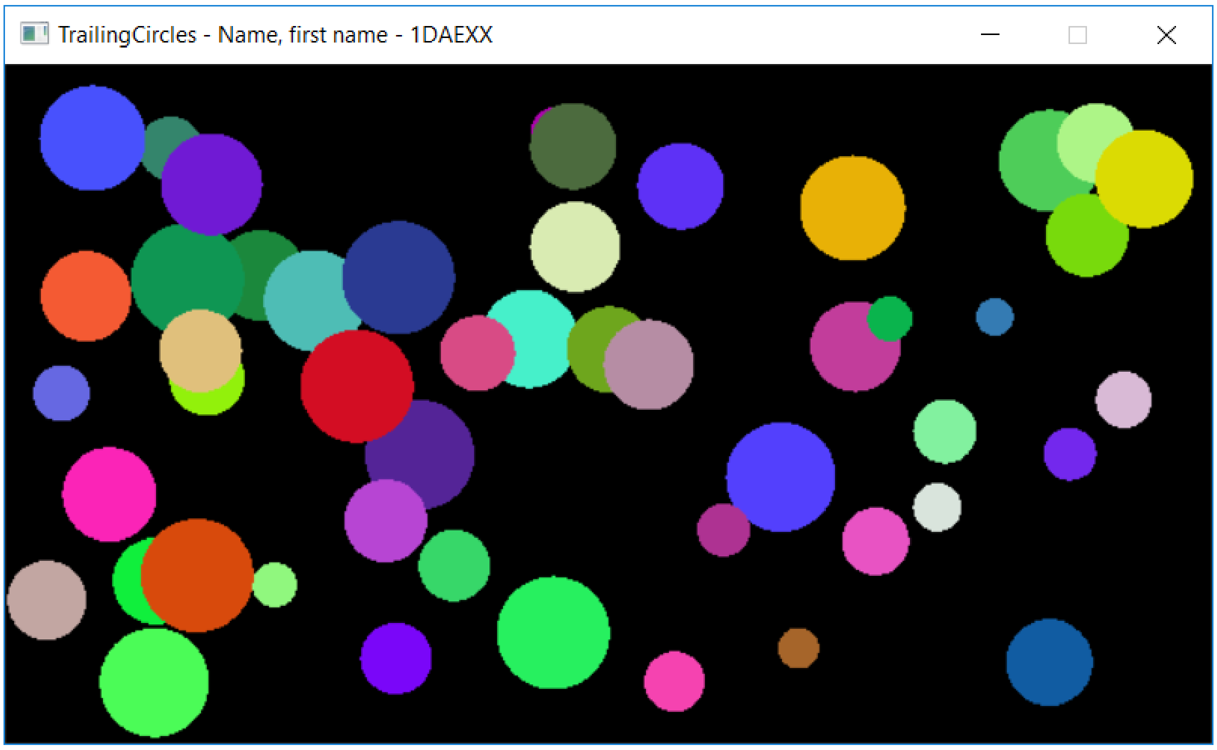
Add a new project with name **TrailingCircles** to the **W09** solution.

Delete the generated file **TrailingCircles.cpp** and add the **framework files**.

Copy the files in the folder **CodeFiles/05_TrailingCircles** into this project folder, allow to overwrite the Game files.

Add the given **TrailingCircle** class files to the project.

In this Game class TrailingCircle objects are created. These objects are circles with a random color and moving at a random velocity and bouncing against the window edges. Build and try it.



3.4.2. Adapt the TrailingCircle class

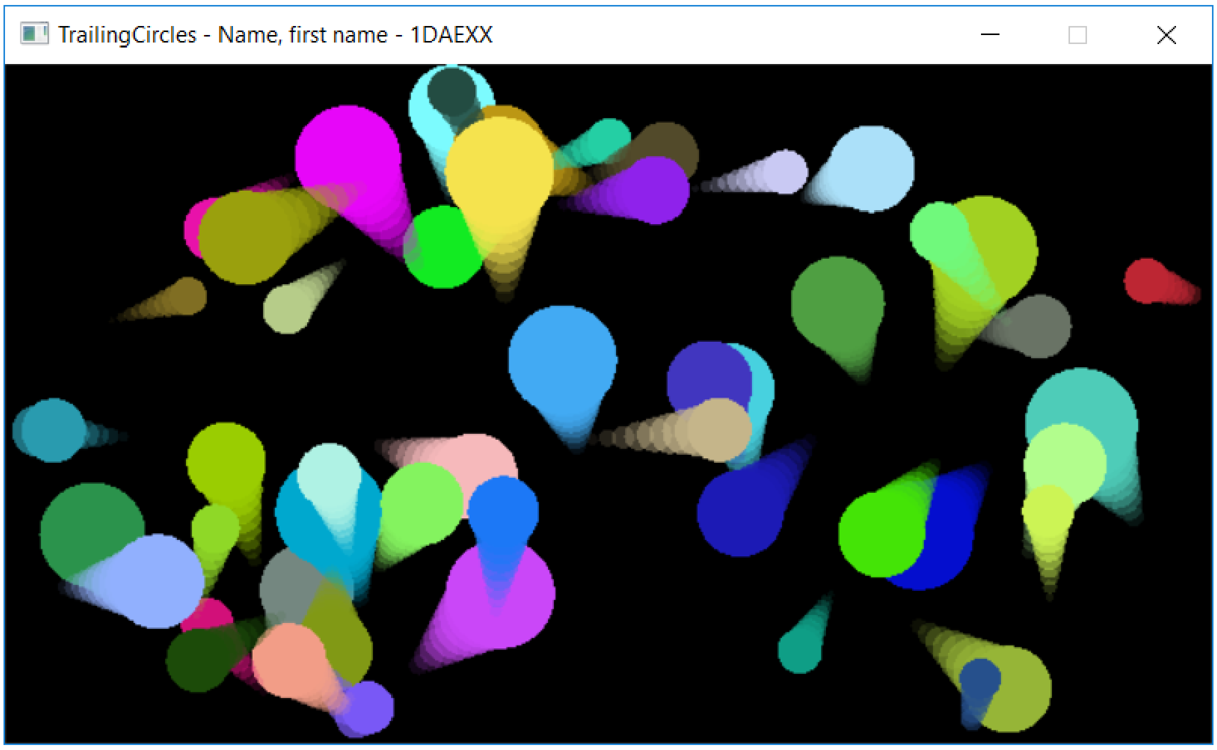
Save the circle 's trails (e.g. each elapsed 80 milliseconds) in a **std::deque** container, only keep the 10 last trails.

Tips:

- Add a new trail position at the end of the deque using **push_back**
- When more than 10 are present, remove the first one in the deque using **pop_front**.

And draw the trail positions with diminishing radius and alpha color channel.

This is the result.



4. Submission instructions

You have to upload the compressed folder *1DAExx_09_name_firstname.* It should contain 2 solutions: W09 and your game: Name_firstname_GameName.

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

5. References

5.1. std::initializer_list class

5.1.1. std::initializer_list

http://en.cppreference.com/w/cpp/utility/initializer_list

5.1.2. Size member function

http://en.cppreference.com/w/cpp/utility/initializer_list/size

5.2. Sequence containers

5.2.1. Vector

http://www.cplusplus.com/reference/vector/vector/

5.2.2. Deque

http://www.cplusplus.com/reference/deque/deque/

5.3. Templates

http://www.cplusplus.com/doc/tutorial/templates/