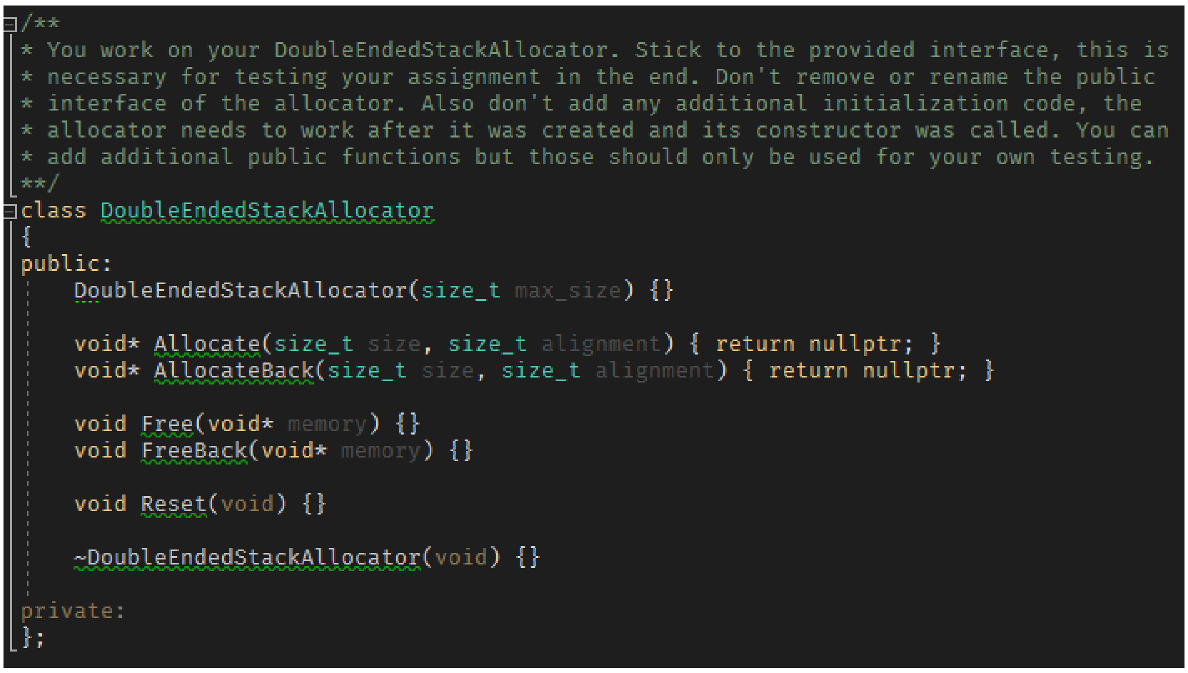
Exercise #1

Implement a double ended stack allocator (25 pts)

The task of the first exercise is to implement a double-ended stack allocator on Windows (x64) using C++. It should be possible to allocate memory from both sides of the allocator. Additionally to allocating and deallocating user requested memory, the allocator should support the debug technique called ‘canaries’ to detect memory overwrites during deallocation. You will receive a skeleton .cpp file containing the allocator interface and some small test functions. Use this .cpp to implement your solution. Hand-in will be handled on Moodle.

Allocator interface (provided in the skeleton .cpp)

It’s important that you stick to the interface. Otherwise running the final tests will not work. You can add additional functionality for your own tests but the presented methods need to stay as they are. You can add as many additional methods as you see fit to fulfill the assignment and general C++ best practices.



```cpp
/**
* You work on your DoubleEndedStackAllocator. Stick to the provided interface, this is
* necessary for testing your assignment in the end. Don't remove or rename the public
* interface of the allocator. Also don't add any additional initialization code, the
* allocator needs to work after it was created and its constructor was called. You can
* add additional public functions but those should only be used for your own testing.
**/
class DoubleEndedStackAllocator
{
public:
    DoubleEndedStackAllocator(size_t max_size) {}

    void* Allocate(size_t size, size_t alignment) { return nullptr; }
    void* AllocateBack(size_t size, size_t alignment) { return nullptr; }

    void Free(void* memory) {}
    void FreeBack(void* memory) {}

    void Reset(void) {}

    ~DoubleEndedStackAllocator(void) {}

private:
};
```

Requirements

- The allocator needs to support allocations of **arbitrary size** (as big as in the max_size range of the allocator) and alignment requirements (power of two).

- Obey C++ programming rules and take care to not invoke undefined behavior
    - How to deal with copy-ctor, assignment op, …? It's your decision in this case

- Allocations and deallocations need to work from both sides
    - During allocation, if there is not enough memory left, assert and return a nullptr

- Free() / FreeBack() take the memory ptr from the user and set the internal state back to point to a location from before the allocation was done. To be more secure one could add LIFO guarantee checking there, but it's not required for the assignment. We assume the user is clever and obeys the LIFO requirement.

- Reset() clears internal state and the whole allocator range is available again

- If the **ENABLE_DEBUG_CANARIES** preprocessor directive is set to 1, the allocator uses canaries to validate the integrity of memory allocations during the deallocation

    - Free() and FreeBack() assert when they detect that a user wrote out-of-bounds of the provided allocation
    - Alignment still works as before

- Although the parameters are passed as size_t the allocator 'only' needs to support allocations with a maximum size of 4,294,967,296 bytes -> sizeof(uint32_t) bytes

Formalities

- Work in **groups of three** students, mention all your names and UIDs in the handed in .cpp file

- Hand in on time, late assignments will get an immediate point deduction of 50%

- Only a **single .cpp** file should be handed in on Moodle before the deadline and make sure it compiles on windows in Visual Studio 2019 with at least C++11

- **Don't** use any additional libraries (boost, …) in this exercise. All you will need can be done in plain C++ and the windows API.

- **Comment** and **test** your assignment. Please use meaningful comments that explain why you are implementing it the way you are. I don't need comments repeating function names or variable assignments. Please **describe** your decisions as code comments as it would be a user documentation. This is part of the grading, so try to not forget about it - the goal is that I can follow your train of thought on **why** you decided to implement certain things the way you did.

- **Don’t** copy from the internet or your colleagues. I know the blogs, the github repos and the other sources. Handing in a copy/plagiat will result in 0 points. A not 100% perfect or faulty implementation is way better than a copy!

Bonus exercise (5 pts)

To gain the bonus for this exercise, the allocator should be able to grow during its lifetime. This should be accomplished by using the **virtual memory** API on Windows. While the basic allocator will allocate a big chunk of memory initially and then cater the user’s allocation from within this block, the growing one will only allocate in page sized chunks and grow when necessary.

In order to implement this, use the **VirtualAlloc(...)** family of function calls provided by the Windows API. Make sure to still support the basic requirements of the allocator when implementing the growing functionality.

Note: When freeing allocations, you don’t need to also take care of freeing already committed pages. Just ‘free’ the internal allocations and let the pages stay committed. If you decide to do it anyways, be careful to only de-committ pages that are really free and can be handed back to the OS.