A5: Dynamic Memory Implementation

- Create your own implementation of malloc(...)/free(...)to allocate/deallocate memory on the heap.
- Create your own implementation of operator new/operator delete using your own malloc(...)/free(...) to create new instances during runtime.
- This assignment should not be too difficult to write if you have some experience with C/C++.
  Instead, the difficult part lies within debugging, writing tests, and ensuring that your program does what you say it does.
- The testsystem only provides some sanity checks for you. The remaining functionality you will have to test and debug yourself, which is a big part of this assignment. The testsystem only tests things explicitly mentioned in this assignment description.
- New and delete will only give you points once your malloc(...) implementation has basic functionality.
  However, the task of the operator new is much, much simpler than the malloc(...) task.

Malloc and Free functions (20p):

Non-functional requirements:

- Your implementation should be in C++
- Organize the heap with respect to performance and memory usage overhead (fragmentation).
- Reduce the number of syscalls made (allocating 100kB should take no more than 30 syscalls, regardless of the number of malloc(...)-calls).
  - Hint: Does brk(...)and sbrk(...)need to be called every time you call malloc(...)?
  - You are also not allowed to use sbrk(0) more than once in your program. (You can assume you are the only one calling sbrk(...)).
  - Think back to A3, what happens internally when you call sbrk(...)? Does it make sense to call sbrk(1) ? What could you do instead?
  - Use strace to examine the syscalls a process makes.
  - You can assume reasonable sizes on the malloc(...)-calls, so e.g. sizes of more than 1 byte.
- Reduce the heap size whenever it makes sense, for example when all the malloc'd data has been freed.
  - This will have some similarities with your solution to the "Reduce the number of syscalls made"-problem, just in the opposite direction.
  - Note: This is not necessarily standard malloc(...) behaviour.
- You may implement free chunk merging as a bonus task (**bonus points!**).
- Your implementation must not contain a main function in any file, except for the testcases in the tests directory.

Functional requirements:

- You are required to use the brk(...)/sbrk(...)functions defined in 4.3BSD or *POSIX* prior to *1-2001*.
- You do not need to use mmap(...), you can ignore MMAP_THRESHOLD
- Detect memory corruption errors. Bring in your own ideas which errors to detect. At least:
  - Buffer overruns/memory corruption invalid free(...)/ double free(...)
  - Handle out of memory correctly.
- In case you run out of memory:
  malloc(...) returns a null-pointer.
- Exit the program with exit code -1 in case of memory corruption, invalid free(...) double free(...)
- The interface of your implementation is required to conform to the *POSIX* standard (see man malloc).
- Do not use malloc(...) or new/delete (except for placement new) in your snp::Memory::malloc(...) implementation.
  - You are required to use brk(...)/sbrk(...) for all your dynamic memory needs.

Tips:

- You can use the placement new operator and add your own classes to organize your implementation. This can make debugging easier as well.
  Usage of the placement new operator will be shown in the A5 extended question hour.
- If you want to use void pointer arithmetic in C++ , you can use the char* instead.
- If you want to use C-style libraries, you can use <cstring>and <cstdlib>for example.
- Some libraries are already included in your malloc/new files, but those are just suggestions, you can use additional ones if you need to.
- Even though you will implement malloc in C++, you can write in a very similar manner to the C code found in the other assignments, if you do not feel comfortable with C++
- Do not use a data structure like std::list , as they use malloc(...) internally. You can use any datastructure that does not use malloc(...) internally.

- Some of the tests are written in C (and you can also decide if you prefer C or C++ for your tests), so you cannot use exceptions. Make sure to add noexcept after your custom function declarations.

New and Delete operators (5p):

Non-functional requirements:

- Your implementation should be in C++
- Put your implementation of new/delete in the provided stubs.
- Only change the already existing function implementations that have a TODO above them. You can add additional functions/variables if you want.
- Your implementation must not contain a main function in any file, except for the testcases in the tests directory.

Functional requirements:

- Use malloc(...) and free(...).
- Your implementation should use exceptions in the case of an error.
  - See std::new for exception usage.
  - Take care that you throw the same exception as std::new.
- We will overload the global new/delete operators, so you can test it just like the standard operator new in C++

Tips:

- You should write your own tests for new/delete.
- You will probably find this task much, much easier than malloc(...)

General Advice:

- Try to test for every requirement found here or on the manpage. If you cannot figure out how to test for a specific requirement, feel free to ask via discord or mail.
- Make sure you have a working debugger, this assignment will be rather difficult without one. I can recommend gdb or VSCode and its built-in debugger UI, if you need help setting it up ask in discord or via mail.
- You are not restricted in how you want to solve this assignment, as long as you don't change the existing code parts you should be fine.
- Attend the A5 extended question hour (17.11.) if you do not feel comfortable with C/ C++or do not know where to start. There you will see some basics on how you can debug, write tests and maybe some small tips for starting out. You can of course also ask me to cover any additional topics that you would like.

Debugging in VSCode:

You can use whatever debugger you prefer. However, if you'd like an easy start you can try the following (assumes gdb and VSCode is installed):

- Download the vscode.zip file. Extract the two files inside into yourrepo/A5/.vscode/(create the folder if it does not exist already).
- Open your A5 folder with VSCode. You should now have three choices to run within the debug section (the bug symbol on the left).
- Now add a breakpoint within your malloc.cpp or .c (click on the left side of the linenumber, you should see a red circle).
- If you select *"tinytest dbg"* and click the green arrow next to it or press F5, you should be all done and ready to debug!
- If you write your own tests, select *"current file dbg"* and open the test you want to debug (make sure it's the currently active file), then press F5.
- Make sure to read up on how to use VSCodes debug UI and all the features it provides, or attend the extended question hour for some basics.

Do not modify the existing parts of the header file

A5/memory.h. You can, however, add new methods/attributes/classes.
You will also find two testcases in the tests directory. Note that the two basic tests we provide do not test all the requirements.
You are expected to test your dynamic memory implementation by writing your own tests and debugging them.
You can add and push as many new files in the /tests/folder as you like.
Pushing your tests to git makes it easier for us to help you, since we can see what you have already tested and what you may have missed.
Any C/ C++ tests you write in your /tests/ folder will be compiled by the provided makefile, so there is no need to compile anything manually.
To compile everything, simple type make
in your repository. You will find the binaries for your tests in the /build/folder, you can start them using ./build/testname.