TU Munich, Department of Informatics
Chair XXV: Data Science and Engineering
Prof. Dr. Thomas Neumann

Exercise for the *Practical Course: Systems Programming in C++* Summer Term 2021

Michael Freitag, Moritz Sichert ({freitagm,sichert}@in.tum.de)
https://db.in.tum.de/teaching/ss21/c++praktikum

Sheet Nr. 07 (due on 10.06.2021 at 23:59)

Create a fork of the Git repository at `https://gitlab.db.in.tum.de/cpplab21/sheet07`. The repository contains a subdirectory for each exercise on this sheet, into which you should put your answers. **Do not forget to `git push` your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

Exercise 1 (100 points)

In C++ containers are usually implemented with templates so that a single implementation can be used for arbitrary types. Additionally, containers also have a template parameter that specifies an *allocator*. Allocators are special classes that can allocate and deallocate memory.

In this exercise you will implement a doubly-linked list that is parametrized by its value type and the allocator it uses. You will also implement two different kinds of allocators, one that just uses `std::malloc()` and `std::free()` (just like the standard allocator in the standard library), and one that uses are more sophisticated strategy to avoid many calls to `std::malloc()`. Finally, you will implement the template function `find()` that can find values in any container that supports iteration and also your implementation of the list.

`malloc()` and `free()`: The standard library defines those functions to manually allocate and deallocate memory. `std::malloc()` takes an integer as parameter that indicates how many bytes should be allocated and returns a `void` pointer to the allocated memory. This pointer is guaranteed to be aligned appropriately for any object type. `std::free()` takes a `void` pointer that was returned by a previous call to `std::malloc()` and deallocates the memory. Both functions usually need to communicate with the operating system which means that calling them often can lead to bad performance.

Allocator: For the purposes of this exercise we define an allocator to be a template class that has exactly one type template parameter `T` and meets all of the following requirements. First of all it must be default constructible so that a container can use it without having to know which arguments to pass to it. It should also be movable so that containers can implement their move operations efficiently. To actually allocate and deallocate memory, it must have the following member functions:

```
T* allocate()
void deallocate(T*)
```

The first function allocates storage for one object of type `T` and returns it. Note that this should *not* initialize the object. The second function takes a pointer that was previously returned by `allocate()` and tries to reclaim the memory. `deallocate()` should not call the destructor of `T`.

An allocator must also have a nested templated type alias that allows to get an allocator for type `U` from an existing allocator for type `T`. This type alias is called `rebind`. For a given allocator `Allocator<T>` the `rebind` type alias should have a type template argument `U` so that `Allocator<T>::rebind<U>` is the same as `Allocator<U>`.

Pooled Allocator: The goal of a pooled allocator is to avoid calling `std::malloc()` often and to achieve spatial locality between allocated objects. For this, it manages objects in exponentially growing *chunks*. Each chunk stores a certain number of objects contiguously in memory which can lead to good spatial locality and therefore better performance. Every chunk is allocated in exactly one call to `std::malloc()` and freed eventually with `std::free()` instead of calling them individually for every object. The chunks need to be linked in a list so that they can be deallocated in the destructor of the allocator. This should not require any additional allocations so the "link" to the next chunk must be stored within a chunk. Apart from the destructor, no other function should iterate through the entire list as this can lead to bad performance.

The `allocate()` function of the pooled allocator should check if the most recent chunk has enough memory left for the allocation. If it is full, a new chunk must be allocated with `std::malloc()` that is exponentially larger than the old one (e.g. double or 1.5 times its size). As described above, the new chunk must be inserted into the list of all previous chunks.

The `deallocate()` function should first see if the pointer that should be deallocated lies at the end of the most recent chunk. In that case it should remember this so that the memory can be reused. In all other cases, `deallocate()` does nothing. This means that the allocator may waste memory as it does not deal with fragmentation at all. Only in the destructor of the pooled allocator all chunks are freed.

Doubly Linked List: For this exercise we consider a doubly linked list to be a template class with the two type template parameters `T` and `Allocator`. This class is a container for objects of type `T`. To allocate storage it should only use an object of type `Allocator` or any type returned from `Allocator::rebind`. It should support default, copy, and move construction, as well as copy and move assignment so it needs to implement the relevant functions for that. Additionally it should implement the following member functions:

```
size_t size() const
void insert(const T& value)
void erase(const T& value)
```

To provide better compiler error messages to users of this class, the copy constructor, copy assignment operator, and the `insert` function should have a constraint that requires the concept `std::copy_constructible` for `T`. Also, the `erase` function should require the concept `std::equality_comparable` for `T`.

- The default constructor should initialize the list to be empty.
- The copy constructor should construct a new list that copies all elements from the given other list in the same order.
- The move constructor should move all resources (i.e. the list itself and also the allocator) from the other list to the one being constructed. The other list should be empty afterwards.
- The copy assignment operator should first remove all elements from the list. Then, it should assign itself a new, default constructed allocator. Finally, it should copy all elements from the other list in the same order.

- The move assignment operator should first remove all elements from the list and then move all resources from the other list to the one being assigned to.
- `size()` should return the number of elements in the list and should have constant complexity.
- `insert()` should insert the value at the end of the list with constant complexity.
- `erase()` should remove the first element that is equal to the given value from the list. If no such element exists, it should do nothing.

The doubly linked list should make sure that for a given entry the previous and the next entry can be found in constant time. Also, the memory locations of list entries should not change when another entry is inserted or removed.

Generic `find()` function: We define a generic `find()` function to be a function that takes two arguments: A non-const reference to a container and a const reference to a value. This function should find an element of the container that is equal to the given value and return a non-const pointer to it. If no such element exists, it should return a null pointer. The type of the value passed to `find()` does not have to be the same as the type of the elements stored in the container. Both types just need to be comparable which should be checked by adding a constraint to `find()` that uses the concept `std::equality_comparable_with`. To actually be generic, `find()` should have several (templated) overloads so that it can handle many different container and value types.

(a) Implement a simple malloc allocator. This allocator should call `std::malloc()` and `std::free()` to allocate and deallocate memory, respectively. Write an allocator class template that satisfies the requirements mentioned above called `MallocAllocator`.

Put your implementation into the file `MallocAllocator.hpp`.

(b) Implement a pooled allocator as described above. Write a class template called `PooledAllocator`.

Put your implementation into the file `PooledAllocator.hpp`.

(c) Write a class template called `List` that implements a doubly linked list as described above.

Put your implementation into the file `List.hpp`.

The subdirectory for this exercise contains a benchmark program that measures the performance of `List`. It compares the runtime of insertions, lookups, and destruction when used with `MallocAllocator` and with `PooledAllocator`. If your implementation is correct, the insertion and destruction time of a list that uses pooled allocation should be significantly faster.

(d) Implement a generic `find()` function as described above. It should support containers from the standard library that support iteration (e.g. `std::vector`) and your `List` class.

Put your implementation into the file `Find.hpp`.

Hints for the implementation:

- All tasks should be implemented in the `pool` namespace.
- `std::malloc` and `std::free` are defined in the header `<cstdlib>`.
- The concepts mentioned above are defined in `<concepts>`.
- Make sure that the memory allocated in your allocators is aligned correctly for the given type. As described above `std::malloc` returns memory that is suitably aligned for all types.

- You should not use any library classes or functions that dynamically allocate memory (`std::unique_ptr`, `std::vector`, etc.) in your linked list or the allocators.
- If you need an allocator of another type `U` in your linked list, use the following type:
  `using UAllocator = typename Allocator::template rebind<U>;`
- The linked list needs to use *placement new* to initialize objects after allocating storage for them.
- Likewise, you have to manually call the destructor of objects before deallocating their storage.
- When you use placement new, you have to include `<new>`.
- You may need at least two different overloads for `find()`: One that takes any container `C` as first argument and the other that takes a `List<T, Allocator>` for any types `T` and `Allocator` as first argument.
- The concept `std::equality_comparable_with` takes two types as template parameters and is only satisfied if the two types are comparable by using `==` and `!=`.
- To get the type of the values in a container of type `C` from the standard library, you can use the nested type `C::value_type`.
- Function templates can also be friends. You need to make sure not to reuse the template parameters of the surrounding class and include all constraints in the same order for that to work:

  ```
  template <typename T, typename U>
  class MyClass {
      template <typename T2, typename U2, typename X>
      friend void foo(MyClass<T2, U2>&, X&)
      requires std::equality_comparable_with<T2, X>;
  };
  ```