Advanced Programming in C++
Container Design I

This is an exercise on container design, especially in the view of robustness, iterators, and storage management. The container will be developed step-by-step, where each step focuses on some particular feature.

The container

Below is the given definition for the container to be designed, a class template named Container, found on file Container.h.

```
template <typename T>
class Container
{
public:
   // add default constructor to create an empty Container
   // add constructor to create a Container with size n
   // add destructor to do clean up

private:
   T* start_;
   T* finish_;
   T* end_of_storage_;
};
```

The elements stored by a Container are to be kept in a dynamically allocated array of T, located at start_. finish_ is supposed to point past-the-last stored element (related to size), end_of_storage_ is supposed to point past the last storage location (related to capacity).

Robustness

Robustness is related to *exception-safety* and *exception-neutrality*. Exception-safety guarantees are usually defined as follows:

1) *Basic guarantee* – no memory must leak in the presence of an exception.

2) *Strong guarantee* – no memory must leak and the state of the container must be kept consistent, if an exception is thrown during the execution of an operation Normally, the state of the container before the operation was applied should be kept.

3) *Nothrow guarantee* – a function will not emit an exception under any circumstances. One example of this is that std::swap guarantees not to throw, if the assignment used in the implementation of sdt::swap does not throw, i.e. **operator**= of the type to be swapped fulfils the nothrow guarantee.

Exception-neutrality means that if an exception is thrown in a container operation, the exception should be propagated to the caller, if not handled, translated or deliberately absorbed by the operation.

For each step you should do a thorough analysis of the (new) code, especially from an exception-safety and exception-neutrality point of view, and then look at the solution and comments in the README file.

Tracer

For this exercise there is a template class Tracer which can be stored in the containers, for producing trace output showing when contained objects are created/copied/destroyed. Tracing can easily be switched on or off (default is on).

1. Initializing and destroying

The first step is to implement some *constructors*, a *destructor*, and also some functions related to size and capacity. Define member functions separate from the class definition, in a separate file named Container.tcc. A good design principle is to isolate code where one can run into problems for example when allocating memory, in helper functions. Start with adding the following.

- dealing with size implies the need for a suitable size type, which std::size_t (<cstdlib>) is – define a nested type **size_type** to make it available as a Container related type
- **allocate_**(*n*) shall allocate memory for *n* objects of type T, using array **new**; throw std::bad_alloc if n > max_size()
- **deallocate_**(*p*) shall deallocating memory pointed to by *p*, consistent with how allocation is done

Implement the following special member functions:

- default constructor to create an empty Container
- constructor to create a Container with initial size n
- destructor

Other special member functions are left later. Add the following member functions:

- **size**() shall returns the current size, i.e. the current number of stored elements
- **max_size**() shall return the maximum size – let this be (max value of size_type) / sizeof(T)
- **capacity**() shall returns the current capacity, i.e. the largest number of elements possible to store without increasing the current capacity
- **empty**() shall returns **true** if the container is empty, otherwise **false**

Analyze your code and ask yourself: What might throw? Is this function exception-safe? Is it exception-neutral? Check against the given solution and comments for step 1.

2. Copying and assigning

When copying a container, it's common practice to allocate just as much memory as required for the copy, i.e. give the copy a capacity equal to the size of the source – the "shrink to fit" idiom. Add:

- copy constructor
- move constructor
- copy assignment operator
- move assignment operator
- helper function(s) that could be useful for allocating and deallocating storage, and copying elements.

Check against the given solution and also see the comments for step 2.

3. Clearing and swapping

Add the following functions:

- **clear***()* shall clear the container, i.e. reset it to an empty container
- **swap***()* shall exchange the contents of two containers, both member and non-member version.

Having these functions you can simplify the implementation of the copy assignment operator by using the "create a temporary and swap" idiom. The move assignment operator can also take advantage of clear() and swap(). Such modifications will make the code clearer, both regarding readability and semantics (especially for move assignment).

Check against the given solution and comments for step 3.

4. Inserting and removing elements

Now you will need to implement a suitable memory allocation scheme for allocating storage when capacity is exhausted. Apply the typical scheme for std::vector, that is, default capacity is 0, increased to 1 when the first element is inserted, thereafter doubled each time the capacity is exhausted. Add the following functions (there may be some more useful helper functions you can add):

- add a helper function to be called when capacity need to be increased. It shall compute a new, larger capacity, allocate new storage, insert *x* at its proper place, copy the already stored elements to the new storage, and, if all goes well, delete the old memory and set start_, finish_, and end_of_storage_ to proper values. This helper function should be overloaded for copy and move semantics, but the latter is bit of a challenge, so let's not…
- **push_back***(x)* shall insert *x* at the end of the container, after increasing capacity when required. push_back() should be overloaded for both copy and move semantics, but skip the latter, for now at least.
- **back***()* shall return the last element in the container, without removing it. If the container is empty no exception is to be thrown, according to standard container semantics..
- **pop***()* shall remove the last object in the container. If the container is empty no exception is to be thrown, according to standard container semantics.
- also make sure we have maximum functionality also for **const** Containers, go through all operations and take appropriate measures when required.

Another interesting issue is what requirements Container puts on its instantiation type, T? Besides the functionality needed for storing T object in a container, also the required operations should not compromise the exception-safety of Container.

- list requirements Container puts on T

Analyse your code, check with the given solution and see the comments for step 4.

Note: An alternative to back() and pop() would be to have had a pop_back() function combining what back() and pop() does, but that's a classic example of getting into trouble. Any modifier function that removes and returns an object can't be made completely exception-safe, since a (correctly) removed object can be lost when copied into a receiving variable on the caller side.

The goal for step 1-4 was to implement Container to be exception-safe and exception-neutral. It seems as we have accomplished this goal, having only two try/catch in the helper functions for storage management. Fairly clean.

When running tests and tracing initialization and copying of elements, we can notice a lot of unnecessary initialization directly followed by assignments, and initialization of elements that may never be used. This should be fixed, but lets us first introduce iterators, se below.

Check against the given solution and comments for step 4.

5. Types

Containers typically declare a set of nested types. We have already declared one in Step 1, size_type, let's add some more:

- **value_type** as a synonym for T
- **pointer** as a synonym for pointer to value_type, and **const_pointer** as a synonym for pointer to const value_type.
- **reference** as a synonym for reference to value_type, and **const_reference** as a synonym for const reference to value_type.

Replace all occurrences of T, T*, and T& (**const** or not) with these nested types, where appropriate.

6. Iterators

For Container, iterators can simply be defines as pointers, T*, and then use the standard iterator adaptor class reverse_iterator to define reverse iterators. Let’s also add some more types, related to iterators:

- define **iterator** as pointer to value_type, and **const_iterator** as **const** pointer to value_type
- use std::reverse_iterator to define **reverse_iterator** and **const_reverse_iterator**, based on iterator and const_iterator
- define **difference_type** as an alternative name for std::ptrdiff_t (can hold the difference when subtracting two pointers referring to elements in the same container, on our case the difference between two iterators).

Define the following member functions for returning iterators:

- **begin** () shall return an iterator referring to the first element in the container
- **end**() shall return an iterator which is the past-the-end value for the container; if the container is empty, then begin() == end().
- **cbegin** () shall return a const iterator to the first element in the container
- **cend**() shall return a const iterator which is the past-the-end value for the container
- **rbegin** () shall return an iterator which is to be semantically equivalent to reverse_iterator(end())
- **rend**() shall return an iterator which is to be semantically equivalent to reverse_iterator(begin())
- **crbegin** () shall return a const reverse iterator semantically equivalent to rbegin()
- **crend**() shall return a const reverse iterator semantically equivalent to rend()

Check against the given solution and comments for step 6. There may be more sophisticated ways to implement **iterator**.

7. Memory handling

If you have used Trace for testing Container, you should have noticed quite a lot of undesired default initializations, and also realized the possibility of elements being default initialized and destroyed but never used to store any actual elements. To eliminate such, we have to allocate and deallocate memory in a way not invoking the default constructor and destructor for T. (when T is such a type).

- use the global allocation function **operator new** to allocate memory – pass n * **sizeof**(T) to allocate memory for n elements of type T
- use the corresponding global deallocation function **operator delete** to deallocate memory
- use placement **new** to construct a T object in memory
- use explicit destructor call to destroy a T object in memory

This way we separate memory allocation/deallocation from constructing/destroying objects, to be able to construct objects only when required and in a possibly more efficient way.

- add helper function to create and destroy a T object in a given memory location
- redesign functions involved in memory allocation and deallocation
- modify, if required, other functions involved in storage management and object handling

Check against the given solution and comments for step 7.

8. Placement push_back

Placement insert is about inserting an object without having to first create the object separately, and then copy or move it into a container. This requires the ability to supply an arbitrary number of constructor arguments to the placement insert operation, which is facilitated by variadic templates and perfect forwarding, which is facilitated by std::forward().

- add **emplace_back**() as a placement version of push_back() (emplace_back() must be a variadic template function)

9. Initializer list

A container is typically expected to be possible to initialize with values from an initializer list.

- add a constructor taking an std::initializer_list, to initialize a Container from a list of values.

  ```
  Container<string> c{ "one", "two", "three" };
  ```

- add an assignment operator taking an std::initializer_list

  ```
  c = { "one", "two", "three" };
  ```

10. More small exercises

There are more operations which could be added, if you like, for example the following, which are found for std::vector:

- **shrink_to_fit**() to reduce capacity to size()
- **resize**(*sz*) – if *sz* <= size() equivalent of calling pop_back() size() - *sz* times; if size() < sz, appends *sz* - size() default-initialized elements to the sequence
- **resize**(*size*, *c*) – if *sz* <= size() equivalent of calling pop_back() size() - *sz* times; if size() < sz, appends *sz* - size() copies of *c* to the sequence
- **reserve**(*n*) to inform a Container of a planned change in size, so it can manage its storage allocation accordingly
- **front**() returns a reference to the first element
- **back**() returns a reference to the last element
- **data**() returns a pointer to the first element – a pointer such that [data(), data() + size()) is a valid range; for an empty Container, data() == &front()
- **insert**(), in different overloaded versions (a value at a given iterator position, n copies of a value a given iterator position, from a range at a given iterator position, from an initializer list at a given iterator position)
- **erase**(*position*) to erase the element at iterator *position*
- **erase**(*first*, *last*) to erase the elements in the iterator range [*first*, *last*)
- **assign**() in different overloaded versions (from a range, from an initializer list, n copies of a value)
- **operator**[*n*] and at(*n*)
- equality and relational operators (==, !=, <, <=, >, >=)