Advanced Programming in C++
Exercise – Container Design and Robustness, Part I

This is an exercise on design of a container class, especially in the view of robustness. The class will be developed step-by-step, where each step focuses on some particular aspect.

The container

Below is the given definition for the container to be designed, a class template named Container with one template parameter T, the element type.

```
template <typename T>
class Container
{
  public:
    explicit Container(const std::size_t size = 0);
    ~Container();

  private:
    T*          elems_;
    std::size_t capacity_;
    std::size_t size_;
};
```

The elements stored by a Container are to be kept in a dynamically allocated array of T, located at elems_. The capacity is zero by default, increased to 1 the first time, and thereafter increased by two times the current capacity each time the current capacity is exceeded. If if constructor argument (n) is greater than 0, the initial capacity is set to n and thereafter doubled each time the capacity is exceeded. The current number of elements stored is given by size_.

The given constructor may act either as a type converting constructor or as a default constructor, if no argument is given. To avoid implicit conversion, the constructor is declared **explicit**.

Type size_t is defined in <cstdlib>, and defined as some integer type.

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.

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 container objects

The first step is to write the *constructor* and *destructor* declared in the given class definition, and also some functions related to size and capacity. Give the definitions separate from the class definition, in a separate file. There is a given file, Container.h, in the directory given_files found from the web page for this exercise.

- The constructor to be written in a way that is exception-safe (works properly in the presence of exception), and exception-neutral (propagates all exceptions to the caller, without causing integrity problems in a Container object).

When you have written the constructor, analyze the code and ask yourself:

- What might throw?
- Is this constructor exception-safe?
- Is it exception-neutral?

- Write the destructor, and analyze the destructor in the same way as the constructor.

Also add the following member functions related to capacity and size:

- size() – returns the current size, i.e. the current number of stored objects.
- capacity() – returns the current capacity, i.e. the largest number of elements possible to store, without increasing the current capacity if the Container.
- empty() – returns **true** if the Container is empty, otherwise **false**.

Do these functions introduce any robustness problems?

When you are satisfied with your code, and convinced that you have done a thorough analysis of the code from an exception-safety and exception-neutrality point of view, you can check against the given solution, found in directory Container_1, together with some comments in the file README.

2. Copying container objects

From the previous step we have a constructor and a destructor which are both exception-safe and exception-neutral. Now we shall design exception-safe and exception-neutral *copy constructor* and *copy assignment operator*. Add declarations for these in the class definition and define them separately.

When using the copy constructor and copy assignment operator, copying from an exciting Container is involved. One good design principle is to isolate code where we can run into problems making a copy in helper functions. So, you should introduce a helper function, named new_copy, for allocating new memory and copying T objects from a source to the new destination.

Let's also be a bit foreseeing – later on we obviously will have an operation for inserting in a Container, and, in that case, sometimes the need for expanding the capacity of the Container. So, new_copy should be able to make a copy of the same size/capacity but also a copy with a greater size/capacity than the source. If successful, new_copy returns the pointer to the newly allocated, possibly grown, and initialized memory, a new copy.

Analyze new_copy in the same way as we did for the copy constructor.

Implement the copy constructor and copy assignment operator with the help of new_copy, and analyze also these operations concerning exception-safety and exception-neutrality. When making a copy, it is common practice in many cases to allocate just as much memory that's needed for the copy, i.e. give the copy a capacity equal to the size of the source – the "shrink to fit" idiom.

When you are satisfied with your code, and your analysis of the code from an exception-safety and exception-neutrality point of view, you can check against the given solution and comments found in directory Container_2.

3. Swapping the contents of two Containers

From the previous exercise we now have a constructor, also acting as default constructor, and a destructor, a copy constructor and a copy assignment operator which are all exception-safe and exception-neutral.

Now we shall add swap functions for Container, one member and one non-member version, and simplify the implementation of the copy assignment operator by using the “create a temporary and swap” idiom.

When you are satisfied with your code, and your analysis from an exception-safety and exception-neutrality point of view, you can check against the given solution and comments found in Container_3.

4. Inserting and removing objects in a Container

Add the following to Container:

push_back() – member function that take an object of type T as argument, and insert it at the end of the Container.

pop_back() – member function that remove the last object in the container, and return it. If the Container is empty, an exception is thrown.

Analyse these operations regarding exception-safety and exception-neutrality.

When you are satisfied with your code, and convinced that you have done a thorough analysis of the code from an exception-safety and exception-neutrality point of view, you can check against a given solution and comments found in Container_4.

5. Respecify how to access and remove the last object in a Container

In the previous step we found that the specification we made for push_back() had severe drawbacks, concerning both exception-safety and responsibilities. Do the following changes:

- back() – add this member function to return the last element in the container, without removing. If the container is empty, an exception is thrown.
- pop_back() – change this member function to just remove the last element in the container, without returning anything. If the container is empty, an exception is thrown.

Maybe it is high time to also make sure we have maximum functionality also for constant Containers.

Analyse these operations regarding exception-safety and exception-neutrality. A solution and comments are available in Container_5.

6. Summing up step 1-5 and some final fixing

The goal for step 1-5 was to implement Container to be exception-safe and exception-neutral. It seems as we have accomplished this goal, with using only one try/catch in the helper function new_copy().

- There is still one way in which Container falls short of the strong guarantee. Find it and fix it!

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.

- Make a list of the requirements Container puts on T.

When you have looked into these issues, there is a solution and comments available in Container_6.