Workshops 40-41

SafeArray, Copy Constructors, Copy assignment operator

These workshops will focus on memory management and safety, starting with implementation of a class `SafeArray`.

`SafeArray` is a template class representing an array of objects of arbitrary type. Memory inside `SafeArray` is dynamically allocated and it should provide basic functions while keeping the safety and correctly handling errors.

Parts of `SafeArray` that have to be implemented.

Fields

The class should have two fields.

1. Pointer to an object of the template type `T`. This pointer will point to the first element of the array.
2. Unsigned integer representing size.

There should be getters for both fields as inline functions providing read-only access to them.

Regular constructors

1. The default constructor.
   It should provide default arguments to the fields of the class, so an object of the class can be created without input arguments.
2. A constructor taking a size variable and a variable of the template type `T`.
   It should initialize an array of `size` elements with the value passed into it as the second argument. If only size is passed, it should use default objects of type `T`, created as `T()`.
3. Constructor with initializer list.
   It should use the size and values of the initializer list to initialize the array.

Constructors must work correctly for all cases, including initialization of an array of size 0. Constructors must never leave any fields uninitialized. Use constructor delegation to achieve this.

Copy constructor

The copy constructor takes as an argument another object of the type `SafeArray` and creates a copy of that object.

Copy assignment operator

The class should have an overloaded `operator=` which takes another object of type `SafeArray` as an input by reference. This operator should replace current values of all fields with ones from the input object.

The straighforward approach - delete current values, assign new values - is unsafe. The better approach is the following:

1. Create a new temporary object as a copy of the input object.
2. Swap the values of the temprary object and the current one (the one that calls the operator).
3. Delete the temporary object. This will be done automatically on exiting the function.

This approach requires implementation of a function `swap` that swaps values of fields of two `SafeArray` objects.

Destructor

The array uses dynamically allocated memory. It must be cleaned when the array is destroyed.

Index functions

The class should provide both `operator[]` and method `at()` to access the data. `operator[]` should not perform checks on the size of the array. Method `at()` should throw an `out_of_range` exception if the index is larger than the size of the array.

Both index functions should have const and non-const versions.

operator<<

`operator<<` for `std::ostream&` objects should be overloaded to allow printing data from `SafeArray` objects.

Resize

The class should have a `resize` function which takes two arguments: size as an unsigned variable and the default value as an object of the template type `T`.

It should delete the previously allocated memory and allocate new memory with the new size. If the new array is larger than the old one, it should fill the new free elements with the defalut value argument. If the defalut value argument isn't provided, the function should use default objects of type `T`.

The function should correctly handle edge cases, not performing unncessesary actions.

Part 2. FastSafeArray

push_back

Add a method `void push_back(T value);` to SafeArray. It should resize the array by adding space for 1 more element, and then write `value` to the new space. You can do it with the method `resize()` in one line.

FastSafeArray

After you finished making `SafeArray` and made sure it works correctly, create a copy of that class, called `FastSafeArray`. Keep both versions so you can compare their speed later.

In `FastSafeArray` implement the following:

Buffer

Add an extra variable to `FastSafeArray`, `size_t _bufferSize`. Split the size logic into two parts. `_size` shows how many values were added to the array, it should be the value an outside user cares about. `_bufferSize` should be the size of allocated space for `_arr`, it handles logic related to memory allocation.

At this point, the sizes should be equal because every time memory is allocated, it is filled with values and vice versa.

Reserving memory (normal version)

Change the behavior of `FastSafeArray`, so that extra space is allocated for potential new values. Allocate space in integer intervals, for example in increments of 20. If your array is empty and you add a single element, use `_arr = new T[20];`. If your array has 99 elements and you resize it to 110 elements, use `_arr = new T[120];`.

This way you don't have to resize the array that often, which should improve speed.

Reserving memory (hard version)

Change the behavior of `FastSafeArray`, so that extra space is allocated for potential new values. Increase allocated space exponentially, for example, in powers of 2. If your array is empty and you add a single element, use `_arr = new T[1];`. If your array has 50 elements and you resize it to 110 elements, use `_arr = new T[128];`.

This way you don't have to resize the array that often, which should improve speed.

Testing

Use 10000 `push_back` operations to test the speed of your arrays.

An example of how it can be done

```
void testSafeArrayTime()
{
    SafeArray<int> sa;

    for (int i = 0; i < 10000; ++i) {
        sa.push_back(0);
    }
}

void testFastSafeArrayTime()
{
    FastSafeArray<int> fsa;
```

```
    for (int i = 0; i < 10000; ++i) {
        fsa.push_back(0);
    }
}


void measureTime(void (*testFunction)())
{
    SafeArray<int> sa;

    std::clock_t c_start = std::clock();
    auto t_start = std::chrono::high_resolution_clock::now();

    testFunction();

    std::clock_t c_end = std::clock();
    auto t_end = std::chrono::high_resolution_clock::now();

    std::cout << std::fixed << std::setprecision(2) << "CPU time used: "
              << 1000.0 * (c_end - c_start) / CLOCKS_PER_SEC << " ms\n"
              << "Wall clock time passed: "
              << std::chrono::duration<double, std::milli>(t_end-
t_start).count()
              << " ms\n";
}

void testTime()
{
    measureTime(testSafeArrayTime);
    measureTime(testFastSafeArrayTime);
}
```