Templates

Søren Hansen <sha@ase.au.dk>
V1.2

Introduction

In the exercise we will endolge ourselves in templates both the class and their functional countaparts.

Prerequisites

- Have a good grasp what templates pertain
- General experience as a coder

Exercise 1 Implementing class MyArray Template

Exercise 1.1 In the beginning - creating the primary template

In this exercise you are going to implement a class that encapsulates an ordinary C array providing a *container* like interface but without utilizing the usual dynamic allocation strategy. The point being that we want the speed and memory efficiency of a ordinary C array but with the added *sophistication* of C++ encapsulation and thus functionality. The idea is heavily inspired by `std::array`(more on this in a later lecture).

Create the class and implement it using the interface below as a starting point. Note that you must be able to parameterize it by the *type* the array is to be made of as well as the *number* of elements in it.

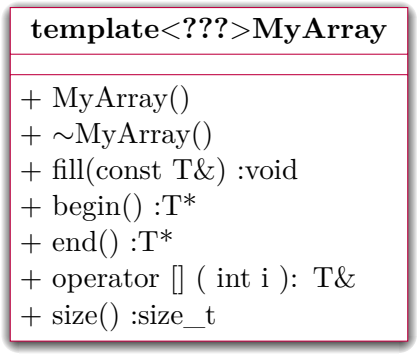

Figure 1.1: The class MyArray interface - simplified

Furthermore the method `begin()` returns a pointer to the first element, whereas `end()` returns one after the last element.

In the interface above *constness* has not been considered... Evaluate the interface and change method(s) to const where appropriate. Likewise add *const* methods you deem necessary. You are going to need them later.

Obviously you should verify that your code actually works as expected. These testing snippets should be included in your hand in.

Exercise 1.2 Finding a specific element in our array - Take 1

In listing 1.1 you will see the signature for the template function you are to implement. It is a free function.

Listing 1.1: Finder template function 'myfind'

```
1 template<typename T>
2 T* myfind(T* first, T* last, const T& v)
```

For testing you could ensure that the array in question is parameterized by an `int` and some number of elements. If that is the case then the following code snippet should find the designated element.

Listing 1.2: Using template function 'myfind'

```
1   my[3] = 3; // Assuming that 'my' has been appropriately allocated based
        on MyArray.
2
3   std::cout << "Looking for '3'? " << (myfind(my.begin(), my.end(), 3) !=
        my.end()? "found" : "sry no") << std::endl;
```

Exercise 1.3 Finding a specific element in our array - Take 2

Our `myfind` template function above does a fair job. However if you change our array in the prior exercise such that it is parameterized by a `double` instead, then the same code will fail...

- Why is this the case, what doesn't the compiler like?
- The code will work if we cast or change the number '3' to what? and why is that?
- Alas the above property is going to irritate users. Therefore alter the interface and implementation of `myfind()` such that it will *behaves*

Hint: Always know your types...

Exercise 1.4 The copy constructor & assignment operator

The build-in copy constructor does a fair job and as such works as desired when two `MyArrays` are parameterized by the exact same types. However this is rather strict, it would thus be desirable that if the two types can implicitly be converted from one to the other then such copy-constrution should work. This obviously extends to the assigments operation as well.

Alter your template class `MyArray` such that the below code will work.

Reflect design-wise on how these 2 operations are to handle the fact that many arrays are not of the same length... In that respect what design choice do you choose and why?

Listing 1.3: Conceptual test code that should work

```
1  MyArray<???> myInt;     // Array based on 'int'
2  MyArray<???> myDouble;  // Array based on 'double'
3
4  // Set some values in 'myInt'
5
6  myDouble = myInt; // From an intuitive perspective this should work,
       since ints can implicitly be converted to doubles.
```

Exercise 2 Partial specialization

Exercise 2.1 Pointers

It is evident from inspecting the code for the primary template `MyArray` that if an instantiation is parameterized by a pointer element e.g. `std::string*`, thus an allocated resource, then these will *not* be deallocated upon `MyArray`'s destruction.

Therefore create a partial specialization that caters for pointers and add a destructor that traverse the array and deletes each and every element using `delete`.

Things to consider in this endeavor:

- The return type for functions `begin()` and `end()`
- The return type for function `operator[](int)` especially, remember code like this must work: `my[2] = new std::string`, assuming it handles `std::string*` elements!
- What do you propose to do with the function `fill()` (that is *implement - then how* or *drop it*) ? - Explanation required!

Exercise 2.2 Finding a specific element in our array of pointers

Again we want to be able ti find a specific element in our array. In that respect we could use the template function `myfind()` which we created earlier, however it would not work as expected, but rather just compare pointers. We therefore need to create an overload of this template function to work with our new partial class template.

The desired signature for this template function is:

Listing 2.1: `myfind()` template function signature

```
1  template<typename T, typename V>
2  T** myfind(T** first, T** last, const V& v)
```

A simple usage scenario could be:

Listing 2.2: Using template function `myfind()` for our new partial class template

```
1     my[5] = new std::string("Hello"); // Assuming that my is a MyArray of
          string pointers
2
3     std::cout << "Looking for 'Hello'? " << (myfind(my.begin(), my.end(),
          std::string("Hello")) != my.end()? "found" : "sry no") << std::endl;
```

Exercise 2.3 Reflection

In this particular design it was chosen that a *partial specialization* would be the choice to handle the deallocation scheme.

If you had had the choice designwise, would you have done the same?

Whether yes or no, discuss pros and cons regarding this solution or this solution as opposed to your preferred solution.

Simply stating one point about another design is not an answer.

Exercise 3 Accumulation - how to do?

In the event that we need an accumulation of all elements in a container, then using a template function is obviously the path to choose but what about the *value type* used in the container? Obviously its easy if we only need to contend with our own type, thus a simple overload would do, however what if the requirement is that it should work with `MyArray<>` as well as say `std::vector<>`?

By design all *containers* do aid us in this respect, namely by defining the type `value_type` like this `typedef T value_type;`. Using this we can instantiate an accumulation variable in our accumulation template function - see listing 3.1 for the implementation which we are going to use.

Listing 3.1: Accumulation template function

```
1  template<typename U>
2  typename U::value_type myAccumalation(const U& u)
3  {
4     typename U::value_type m = typename U::value_type();
5
6     for(auto first : u)
7     {
8        m += first;
9     }
10
11    return m;
12 }
```

In listing 3.1 several things standout:

- What kind of type is `U::value_type`?
- What does this code - `typename U::value_type m = typename U::value_type();` - do?
- The use of the keyword `auto` in a special setting, what is going on?
- What is the specific requirement on `U::value_type`? Inspect its usage in the implementation above.

Verify that `myAccumulation()` works with our `MyArray<>` - any change needed? And verify that it also works with say `std::vector<>`.