Lab: Cloning `std::array<T,N>` using Class Templates

Learning Outcomes

- Gain experience in class templates by implementing containers that are independent from types of stored data
- Gain experience in developing software that follows data abstraction and encapsulation principles
- Reading, understanding, and interpreting C++ unit tests written by others

Overview

Polymorphism is a programming technique where a single identifier, symbol, interface or construct is used in code to represent multiple different definitions. At some point the program has to determine which specific definition should be selected. We refer to polymorphism as *static*, if this decision is made at *compile time* by a compiler based on the use context. Static polymorphism has a significant cost of longer compilation times, but may contribute to zero or minimal run-time overhead. In contrast, we refer to polymorphism as *dynamic*, if the decision of which specific definition to use is made by a program at *run time*, based on execution of function calls. Dynamic polymorphism has minimal compile time cost, but may lead to slower run-time execution involving multiple memory access calls to retrieve information from a virtual table that has been generated for a type to identify functions overridden from its base type.

We have already discussed polymorphic techniques. We have covered static polymorphism disguised as function overloading, and operator overloading. Dynamic polymorphism in C++ is mostly related to inheritance and virtual functions and will be covered in future lectures this semester. Polymorphism lets us develop better abstractions: regardless of types involved, polymorphic function calls or expressions may not require syntax changes.

In this exercise we are defining function and class templates, which are another manifestation of *compile time* polymorphism. Function templates and class templates represent entire families of functions and classes in a generic way – they omit specification of some internally referenced types or – as you will learn in this exercise – compile time integral constants.

Task

C++ evaluates the name of a built-in array in an expression as a pointer to its first element. For this reason, an array name will decay to a pointer when passed to a function. Consequently the corresponding function parameter can only be defined as a pointer [to the type of array element] and will be initialized with the address of the array's first element. That function can iterate through the array elements only if the function signature is augmented with an additional parameter initialized with the array size. C++11 introduced the `std::array<T,N>` container, defined in `<array>`, as a fixed-size sequence of `N` elements of a given type `T` where the number of elements is specified at compile time. Thus, an `array` can be allocated with its elements on the stack, in an object, or in static storage. The elements are allocated in the scope where the `array` is defined. An `array` is best understood as a built-in array with its size firmly attached, without implicit, potentially surprising conversions to pointer types, and with a few convenience functions provided.

Why should we use an `array` when a `std::vector<T>` is so much more flexible? Although an array is less flexible, it is simpler. Occasionally, there is a significant performance advantage to be had by directly accessing elements allocated on the stack rather than allocating elements on the free store, accessing them indirectly through the `vector` [which has a *handle* to the array elements], and then deallocating them. Since a `vector` is a handle to the dynamically allocated array that can grow, it needs to store more data while an `array` has zero overhead because it doesn't keep any data other than the array elements. On the other hand, the stack is a limited resource [especially on some embedded systems], and stack overflow is nasty.

Why would we use an `array` when we could use a built-in array? Since built-in arrays decay to a pointer in an expression, they cannot be copied, nor assigned, and nor can they be passed to functions without specifying an additional size parameter. Since an `array` is an object, there are no nasty conversions to pointers. An `array` knows its size, so it is easy to use with standard library algorithms, and it can be copied and assigned. There is no overhead [time or space] involved in using an `array` compared to using a built-in array.

Your task is to implement a class template that abstracts a built-in array - that is, a simpler clone of `std::array<T,N>`. To define a class that encapsulates a built-in array, you need the type of elements to be stored in a built-in array, and a compile time constant representing the built-in array's size. These parameters will not be hardcoded in your implementation. Instead, you must implement a class template, and use template type parameter `T` [for the array element type] and nontype parameter `N` [for the array dimension], respectively. The class template must have the following definition:

```
namespace hlp2 {

template <typename T, size_t N>
class Array {
  // to be implemented by you!!!
}

} // end namespace hlp2
```

You are familiar with *template type parameters* specified by keyword `typename` [from class lectures and previous assessments]. Nontype parameters work in a similar fashion, but have a few significant differences. They represent compile time numeric constants, not types. They can only use integral types [`short`, `int`, `size_t`, `bool`, and so on], enumeration types, pointer and reference types. Floating-point types will be supported only from C++20 onwards.

Instantiations of templates with different template parameters represent different classes. Therefore, types `hlp2::Array<int,4>` and `hlp2::Array<int,3>` are distinct concrete classes that are instantiated from the same class template.

The lecture handout on template nontype parameters provides an introduction to this topic. See the example code from lectures on class templates for a practical example of implementing a class template using both template type and nontype parameters.

Implementation details

You must implement an interface for container `hlp2::Array<T,N>` that consists of:

1. 18 member functions: `begin()`, `begin() const`, `end()`, `end() const`, `cbegin() const`, `cend() const`, `front()`, `front() const`, `back()`, `back() const`, `operator[]()`, `operator[]() const`, `empty() const`, `data()`, `data() const`, `size() const`, `fill()`, and `swap()`.
2. 5 non-member functions: `swap()`, `operator==()`, `operator!=()`, `operator>()`, and `operator<()`.

These functions should have the exact same behavior as the corresponding function of `std::array<T,N>`. This is a good place to study the interface provided by `std::array<T,N>` and many code examples.

While implementing the class template you should pay particular attention to the following:

1. Constructors, copies, assignments, and destructor. Notice that constructors [or a destructor] are *not present* in the list of member functions that you must implement. How are `hlp2::Array<T,N>` initialized, copied, and destroyed? What about memory leaks? And, what about the the Rule of 3? Since `hlp2::Array<T,N>` is a clone of `std::array<T,N>`, we can answer these questions by understanding how `std::array<T,N>` works. Class `std::array<T,N>` has unique semantics compared to other containers such as `std::string` and `std::vector<T>` - it is considered to be a ***templatized aggregate***. Aggregate classes are classes or structures with ***no user-provided, explicit constructors and no private nonstatic data members***. For example:

   ```
   1 struct IntWithComment {
   2   int value;
   3   std::string comment;
   4 };
   ```

   defines an aggregate structure. Objects of type `IntWithComment` can be initialized by an initializer list:

   ```
   1 IntWithComment iwc {1, "a"};
   ```

   Further, aggregate classes can also be templates. For example:

   ```
   1 template <typename T>
   2 struct ValueWithComment {
   3   T value;
   4   std::string comment;
   5 };
   ```

   defines an aggregate parameterized for the type of the value `value` it holds. You can declare objects as for any other class template and still use it as aggregate:

   ```
   1 ValueWithComment<int> vc{12, "initial value"};
   ```

   Likewise an `array<T,N>` can be initialized by an initializer list with up to `N` elements whose types are convertible to `T`:

```
std::array<int,3> a0 {1, 2, 3};
std::array<double,5> a1 {1.1, 2.2, 3.3, 4.4, 5.5};
```

The number of elements in the initializer must be equal to or less than the number of elements specified for the `array`. As usual with a built-in array, if the initializer list provides values for some but not all elements, the remainder are initialized with the appropriate default value:

```
std::array<std::string,5> a2 {"a", "b", "c"};
```

The first 3 elements of `a2` are initialized by the `string` constructor that takes a parameter of type `char const*` while the last 2 elements are initialized by `string`'s default constructor as empty strings.

You can provide an empty initializer list to guarantee all array elements of fundamental types are zero initialized:

```
std::array<int,4> a3 {};         // ok: all elements have value 0
std::array<std::string,4> as3{}; // ok: all elements are empty string
```

Think about what happens when initializers are not provided when defining `array<T,N>`'s:

```
std::array<int,4> a4;         // oops!!! elements have undefined values
std::array<std::string,5> a5; // ok: all elements are empty string
```

The definitions of `a4` and `a5` do not specify initializers. Since class `array<T,N>` does not declare any constructors, the compiler will synthesize a default constructor that does not create an empty container, because the number of elements in the container is always constant according to the second template nontype parameter throughout its lifetime. Instead, we could assume the default constructor does nothing. Prior to the default constructor's execution, each array element is default initialized using `T`'s default constructor. Since built-in types don't declare a default constructor, the 4 elements of `a4` will have unspecified values. On the other hand, each element of `a5` will be default initialized to an empty string by `string`'s default constructor.

Since `array<T,N>` doesn't define copy constructor, copy assignment, and destructor functions, the compiler will synthesize these functions when required:

```
std::array<std::string,5> a2 {"a", "b", "c"};
std::array<std::string,5> a5 {a2}; // use synthesized copy ctor
a2 = a5; // use synthesized copy assignment
```

The synthesized copy constructor for `array<string,5>` cannot directly copy `a2`'s built-in array to the corresponding built-in array member of `a5`. Instead, the synthesized copy constructor uses the copy constructor of the element type [which is `string`] to copy each element of `a2`'s built-in array to the corresponding element of `a5`'s built-in array. Similar to the synthesized copy constructor, the synthesized copy assignment assigns each element of `a5`'s built-in array to the corresponding element of `a2`'s built-in array using the copy assignment member function of the element type [which is `string`].

When a `array<T,N>` object goes out of scope, the compiler will synthesize a default destructor which will do nothing. The object's built-in array will be destroyed as part of the implicit destruction phase that follows the destructor body. This implicit destruction phase will call the `string` destructor for each element of the built-in array:

```
int main() {
  std::array<std::string,5> a2 {"a", "b", "c"};
} // call std::string dtor for each element of a2's built-in array
```

2. All standard library containers such as `std::string` and `std::vector<T>` provide common type definitions. Likewise, your implementation of class template `hlp2::Array<T,N>` must define identifiers `size_type`, `value_type`, `reference`, `const_reference`, `iterator`, `const_iterator`, `pointer`, and `const_pointer` as names for corresponding types.

3. You should not store any "management information" such as size in an `hlp2::Array<T,N>` because the number of elements in the container is always constant throughout its lifetime and is specified by the second template nontype parameter `N`.

```
hlp2::Array<int,3> aa {1, 2, 3};
// if compiler evaluates the expression as true, the statement has
// no effect. Otherwise a compile-time error is issued and the text
// "Incorrect implementation!!!" is included in the diagnostic message.
static_assert(sizeof(aa)==sizeof(int)*3, "Incorrect implementation!!!");
```

4. The element count is not optional:

```
hlp2::Array<int> ia {1, 2, 3}; // error: size not specified
```

5. The element count must be a constant integral expression:

```
void f(int n) {
  hlp2::Array<int,n> ia {1, 2, 3}; // error: size not constant expression
  // ...
}
```

Remember, if the client wants the element count to be variable, she should use `std::vector<T>`.

6. There is no constructor for `hlp2::Array<T,N>` that copies an argument value [as there is for `std::vector<T>`]. Instead, a `fill` method is provided:

```
void f() {
  hlp2::Array<int,8> ia; // uninitialized, so far
  ia.fill(99);           // assign eight copies of 99
  // ...
}
```

7. Because a `hlp2::Array<T,N>` doesn't follow the "handle to elements" model [as do `std::string` and `std::vector<T>`], member function `swap` has to actually swap elements so that swapping two `Array<T,N>`s applies `swap` to `N` pairs of `T`s.

8. When necessary, a `hlp2::Array<T,N>` can be explicitly passed to a C-style function that expects a pointer through member function `data`. For example:

```
void f(int *p, int sz); // C-style interface

void g() {
  hlp2::Array<int,10> a;

  f(a, a.size());        // error: no conversion
  f(&a[0], a.size());    // C-style use
  f(a.data(), a.size()); // C-style use
}
```

9. ***Other than `<cstddef>`, your submission files must not include any other standard library header.***

10. Compiling templates: You now know that definitions of function templates must be available to any translation unit that instantiates these templates. Hence, you declared and defined the function templates in a single .hpp file that was then included in every source file that instantiates these templates. The same is true with definitions of class templates and corresponding member and non-member functions. This week, you'll physically split the definition of class template `hlp2::Array<T,N>` and its implementation: you'll define the class template and declare non-member functions in file array.hpp; you'll provide the definitions of member and non-member functions in file array.tpp; and then you'll include array.tpp in file array.hpp:

```
//-----------------------------------------------------------------------
---
#ifndef ARRAY_HPP_
#define ARRAY_HPP_
//-----------------------------------------------------------------------
---

#include <cstddef> // for size_t
// Don't include <array> - your submission will not be compiled!!!

namespace hlp2 {

// Define class template Array<T,N> and document each
// member and non-member function

// ALL member and non-member functions must be defined in
// separate file array.tpp
#include "array.tpp" // Don't include <array> in array.tpp!!!

} // end namespace hlp2

#endif // end ARRAY_HPP_
```

11. An important detail to remember about templates is that ***code is instantiated only for template member and non-member functions that are called***. That is, member and non-member functions for a class template are instantiated only if the concrete class [instantiated from the class template] calls these functions. This, of course, saves time [during compilation] and space [for the executable generated by the linker] and allows use of class templates only partially. So one way for you to test your implementation is to define class template `hlp2::Array<T,N>` and non-member functions in array.hpp and define one function at a time in array.tpp. Then, write a simple driver that tests the newly defined function. Once

this function is tested, add another function to test. This incremental approach will allow you to easily identify and fix an incorrect function.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file and documentation details

You will be submitting file array.hpp and array.tpp.

Compiling, executing, and testing

Download array-driver.cpp and output files containing the correct output for the 3 unit tests in the driver. Follow the steps from the previous lab to refactor a makefile that can compile, link, and test your program.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. Click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of g++ options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader [you can see the inputs and outputs of the auto grader's tests]. The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
    - The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
    - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.