Subscript Operator Templates

The `operator[]()` function is quite straightforward, but we must ensure illegal index values can't be used. For an index value that is out of range, we can throw an exception:

```
template <typename T>
T& Array<T>::operator[](size_t index)
{
  if (index >= size)
    throw std::out_of_range {"Index too large: " + std::to_string(index)};

  return elements[index];
}
```

We could define an exception class to use here, but it's easier to borrow the `out_of_range` class type that's already defined in the `stdexcept` header. This is thrown if you index a `string`, `vector<>`, or `array<>` object with an out-of-range index value, for example, so the usage here is consistent with that. An exception of type `out_of_range` is `thrown` if the value of `index` is not between `0` and `size-1`. An index can already not be less than zero because it is of type `size_t`, which is an *unsigned* integer type, so all we need to check for is that the given `index` is not too large. The argument that is passed to the `out_of_range` constructor is a message that includes the erroneous index value to make tracking down the source of the problem a little easier.

In a first, natural implementation, the `const` version of the subscript operator function would be almost identical to the non-`const` version:

```
template <typename T>
const T& Array<T>::operator[](size_t index) const
{
  if (index >= size)
    throw std::out_of_range {"Index too large: " + std::to_string(index)};

  return elements[index];
}
```

However, introducing such duplicate definitions for the `const` and non-`const` overloads of a member function is considered bad practice. It is a particular instance of what is typically referred to as *code duplication*. Because avoiding code duplication is key in making sure your code remains maintainable, we'll contemplate this a bit more before we continue with class templates.

CODE DUPLICATION

Writing the same or similar code more than once is rarely a good idea. Not only is it a waste of time, such so-called duplicated code is undesirable for a number of reasons—most notably because it undermines the maintainability of your code base. Requirements evolve, new insights are gained, and bugs are discovered. So, more often than not, your code will need to be adjusted several times after it is written. So if you have duplicated code snippets, this means you have to remember to always adjust all individual copies of the same code. Believe us when we say that this is a maintenance nightmare! The principle of avoiding code duplication is also sometimes called the *Don't Repeat Yourself* (DRY) principle.

Even if the duplicated code is just a few lines it is often already worthwhile rethinking it. Consider, for instance, the duplicated `operator[]()` member definitions we wrote for the `Array<>` template. Now imagine that at some point later you want to change the type of exception thrown or change the message passed to the exception. Then you would have to change it in two places. Not only is this tedious, but it would be really easy to forget either one of these duplicates. This unfortunately occurs a lot in practice; changes or bug fixes to duplicated code are made in only some of the duplicates, while other duplicates live on containing the original, now-incorrect version. If you only have each piece of logic in one single place in your code base, this cannot happen!

The good news is that you already know most of the tools you need to battle code duplication. Functions are reusable blocks of computations and algorithms, templates instantiate functions or classes for any number of types, a base class encapsulates all that is common to its derived classes, and so on. All these mechanisms were created precisely to make sure you do not have to repeat yourself!

The traditional approach to eliminate the code duplication between the `const` and non-`const` overloads of a member function is to implement the non-`const` version in terms of its `const` twin. While this sounds simple enough in principle, the resulting code may, nay *will*, seem daunting at first. Prepare yourself. For our `operator[]()` member, for instance, the classical implementation of this idiom looks as follows:

```
template <typename T>
T& Array<T>::operator[](size_t index)
{
  return const_cast<T&>(static_cast<const Array<T>&>(*this)[index]);
}
```

Ouch! We warned you that it would get scary, didn't we? The good news is that C++17 has introduced a little helper function, `std::as_const()`, that makes this code already a bit more bearable:

```
template <typename T>
T& Array<T>::operator[](size_t index)
{
  return const_cast<T&>(std::as_const(*this)[index]);
}
```

That's quite a bit shorter and more readable already. Still, since this is your first encounter with the idiom, let's first rewrite that still nonobvious `return` statement into some smaller steps. That will help us explain what is going on:

```
template <typename T>
T& Array<T>::operator[](size_t index)
{
 Array<T>& nonConstRef = *this;                          // Start from a non-const ref
 const Array<T>& constRef = std::as_const(nonConstRef); // Convert to const ref
 const T& constResult = constRef[index];                // Obtain the const result
 return const_cast<T&>(constResult);                    // Convert to non-const result
}
```

Because this template generates non-`const` member functions, the `this` pointer has a pointer-to-non-`const` type. So in our case, dereferencing the `this` pointer gives us a reference of type `Array<T>&`. The first thing we need to do is add `const` to this type. As of C++17, this can be done using the `std::as_const()` function defined in the `<utility>` header of the Standard Library. Given a value of type `T&`, this function template evaluates to a value of type `const T&`. (If your implementation does not contain this C++17 utility yet, you need to use an equivalent `static_cast<const T&>` as shown earlier.)

Next, we simply call the same function again, with the same set of arguments—an operator function with a single `size_t` argument `index` in our case. The only difference is that this time we call the overloaded function on the reference-to-`const` variable, which means that the `const` overload of the function—`operator[](size_t) const`—gets called. If we hadn't first added `const` to the type of `*this`, we'd simply be calling the same function again, which would trigger infinite recursion.

Because we now call the function on a `const` object, it also means that it typically returns a `const` reference. If it didn't, it would break `const` correctness. What we need, however, is a reference to a non-`const` element. In a final step, we must therefore strip away the `const`ness of the result before returning it from the function. And, as you know, the only way to remove `const` is by using a `const_cast<>`.

Paraphrasing J. R. R. Tolkien, we propose to call this idiom "`const`-and-back-again." You first go from non-`const` to `const` (using `std::as_const`) and then back again to non-`const` (using a `const_cast<>`). Note that this idiom is one of the few cases where it is actually recommended to use a `const_cast<>`. In general, casting away `const`ness is considered bad practice. But eliminating code duplication using the `const`-and-back-again idiom is a widely accepted exception to this rule:

■ **Tip** Use the `const`-and-back-again idiom to avoid code duplication between the `const` and non-`const` overloads of a member function. In general, it works by implementing the non-`const` overload of a member in terms of its `const` counterpart using the following pattern:

```
ReturnType Class::Function(Arguments)
{
  return const_cast<ReturnType>(std::as_const(*this).Function(Arguments));
}
```

The Assignment Operator Template

There's more than one possibility for how the assignment operator works. The operands must be of the same `Array<T>` type with the same `T`, but this does not prevent the `size` members from having different values. You could implement the assignment operator so that the left operand retains the same value for its `elements` member whenever possible. That is, if the right operand has fewer elements than the left operand, you could just copy sufficient elements from the right operand to fill parts of the array for the left operand. You could then either leave the excess elements at their original values or set them to the value produced by the default `T` constructor.

To keep it simple, however, we'll just make the left operand allocate a new `elements` array always, even if the previous array would be large enough already to fit a copy of the elements of the right operand. To implement this, the assignment operator function must release any memory allocated in the destination object and then do what the copy constructor did. To make sure the assignment operator does not `delete[]` its own memory, it must first check that the objects are not identical. Here's the definition:

```
template <typename T>
Array<T>& Array<T>::operator=(const Array& rhs)
{
  if (&rhs != this)                          // If lhs != rhs...
  {                                          // ...do the assignment...
    delete[] elements;                       // Release any free store memory

    size = rhs.size;                         // Copy the members of rhs into lhs
    elements = new T[size];
    for (size_t i {}; i < size; ++i)
      elements[i] = rhs.elements[i];
  }
  return *this;                              // ... return lhs
}
```

Remember, checking to make sure that the left operand is not identical to the right is essential; otherwise, you'd free the memory for the `elements` member of the object pointed to by `this` and then attempt to copy it to itself when it no longer exists! Every assignment operator of this form must start with such a safety check. When the operands are different, you release any free store memory owned by the left operand before creating a copy of the right operand.

EXCEPTION SAFETY

The assignment operator for our `Array<>` class template will work perfectly in the nominal case. But what if something goes wrong? What if an error occurs during its execution and an exception is thrown? Can you perhaps locate the two places in the function's code where this might happen? Try to do so before reading on.

The two potential sources of exceptions inside our function's body are annotated in the following code snippet:

```
template <typename T>
Array<T>& Array<T>::operator=(const Array& rhs)
{
  if (&rhs != this)
  {
    delete[] elements;

    size = rhs.size;
    elements = new T[size];             // may throw std::bad_alloc
    for (size_t i {}; i < size; ++i)
      elements[i] = rhs.elements[i];    // may throw any exception (depends on type T)
  }
  return *this;
}
```

The first is operator `new[]`. In the previous chapter, you learned that it throws a `std::bad_alloc` exception if free store memory cannot be allocated for some reason. While unlikely, especially on today's computers, this can certainly happen. Perhaps `rhs` is a very large array that doesn't fit twice in the available memory.

■ **Note** Free store memory allocation is a rare occurrence these days because physical memory is large and because virtual memory is very large. So, checking for or considering `bad_alloc` is omitted in most code. Nevertheless, given that in this case we are implementing a class template whose sole responsibility is managing an array of elements, properly handling memory allocation failures does seem appropriate here.

The second potential source of exceptions is the `elements[i] = rhs.elements[i]` assignment expression. Since the `Array<T>` template can be used with any type `T`, it might just be instantiated for a type `T` whose assignment operator throws an exception if the assignment fails. One likely candidate already is again a `std::bad_alloc`. As witnessed by our own assignment operator, an assignment often involves memory allocation. But in general this could be any exception type. It all depends on the definition of the assignment operator of the type `T`.

■ **Tip** As a rule, you should assume that *any* function or operator you call might throw an exception and consequently consider how your code should behave if and when this occurs. The only exceptions to this rule are functions annotated with the `noexcept` keyword and most destructors, as these are generally implicitly `noexcept`.

Once you have identified all potential sources of exceptions, you must analyze what would happen if exceptions are in fact thrown there. It would again be good practice for you to do so now, before reading on. Ask yourself, what exactly would happen to the `Array<>` object if an exception occurs in either of these two locations?

If the `new[]` operator in our example fails to allocate new memory, the `elements` pointer of the `Array<>` object becomes what is known as a *dangling pointer*—a pointer to memory that has been reclaimed. The reason is that right before the failing `new[]`, `delete[]` was already applied on `elements`. This means that even if the caller catches the `bad_alloc` exception, the `Array<>` object has become unusable. Worse, actually, its destructor is almost certainly going to cause a fatal crash because it'll again apply `delete[]` on the now-dangling `elements` pointer.

Note that assigning `nullptr` to `elements` after the `delete[]` like we recommended earlier would in this case only be a small patch on the wound. As none of the other `Array<>` member functions—for instance, `operator[]`—checks for `nullptr`, it would again only be a matter of time before a fatal crash occurs.

If one of the individual assignments performed inside the `for` loop fails, we are only slightly better off. Supposing the culprit exception is eventually caught, you are left with an `Array<>` object where only the first some `elements` have been assigned a correct new value, while the rest is still default-constructed. And there is no way of knowing how many have succeeded.

When you call a member function that modifies an object's state, you typically want one of two things to happen. Ideally, of course, the function fully succeeds and brings the object into its desired new state. As soon as any error prevents a complete success, however, what you really do not want is to be left with an object in some unpredictable halfway state. Leaving a function's work half-finished mostly means that the object becomes unusable. Once anything goes wrong, you instead prefer the object to remain or revert to its initial state. For our assignment operator, this means that if the assignment fails to allocate and assign all elements, the end result should be that the `Array<>` object still points to the same `elements` array as it did prior to the assignment attempt.

Like we said in Chapter 15, writing code that is correct might be only half of the work. Making sure that it behaves reliably and robustly when faced with unexpected errors can be at least as hard. Of course, proper error handling always starts from a cautious attitude. That is, always be on the lookout for possible sources of errors, and make sure you understand what the consequences would be of such errors. Luckily, once you have located and analyzed the problem areas—and you'll get better at spotting these over time—there exist standard techniques to make your code behave correctly after an error. Let's see how this might be done for our example.

The programming pattern that can be used to guarantee the desired all-or-nothing behavior for our assignment operator is called the *copy-and-swap idiom*. The idea is simple. If you have to modify the state of one or more objects and any of the steps required for this modification may throw, then you should follow this simple recipe:

1. Create a *copy* of the objects.
2. Modify this copy instead of the original objects. The latter still remain untouched!
3. If all modifications succeed, replace—or *swap*—the originals with the copies.

However, if anything goes wrong either during the copy or any of the modification steps, simply abandon the copied, half-modified objects and let the entire operation fail. The original objects then remain as they were.

While this idiom can be applied to virtually any code, it is often used within a member function. For an assignment operator, the application of this idiom often looks like this:

```
template <typename T>
Array<T>& Array<T>::operator=(const Array& rhs)
{
  if (this != &rhs)
  {
    Array<T> copy{rhs};        // Copy...        (could go wrong and throw an exception)
    swap(copy);                // ... and swap!  (noexcept)
  }
  return *this;
}
```

The main thing to note is that we have rewritten the assignment in terms of the copy constructor. The self-assignment test is now no longer strictly required, but there is no harm in adding it either. Still, if you omit it, you can rewrite the copy assignment operator to make it even shorter:

```
template <typename T>
Array<T>& Array<T>::operator=(Array rhsCopy) // Copy...  (could throw an exception)
{
  swap(rhsCopy);              // ... and swap!  (noexcept)
  return *this;
}
```

Copy construction now occurs when the operator's right side is passed to the function by value.

In a way, this is actually a degenerate instance of the copy-and-swap idiom. In general, the state of the copied object may need any number of modifications between the copy and the swap stages of the idiom. Of course, these modifications are then always applied to the `copy`, never directly to the original object (`*this`, in our case). If either the copy step itself or any of the additional modification steps that may follow throw an exception, the stack-allocated `copy` object is automatically reclaimed, and the original object (`*this`) remains unchanged.

Once you are done with updating `copy`, you swap its member variables with those of the original object. The copy-and-swap idiom hinges on the assumption that this final step, the swapping, can be done without any risk for exceptions. That is, it must not be possible that an exception occurs with some members already swapped and others not. Luckily, implementing a `noexcept` swap function is almost always trivial.

By convention, the function to swap the contents of two objects is called `swap()` and is implemented as a nonmember function in the same namespace as the class whose objects it is swapping. (We know, in our `Array<>` template that it is a member function as well. Be patient, though, we're getting to that!) The Standard Library `<utility>` header also offers the `std::swap<>()` function template that can be used to swap values or objects of any copyable data type. For now, you can think of this template as if it was implemented like this:[1]

```
template <typename T>
void swap(T& one, T& other) noexcept
{
  T copy(one);
  one = other;
  other = copy;
}
```

Applying this template to `Array<>` objects would not be particularly efficient. All the `elements` of the objects being swapped would be copied several times. Besides, we could never use it to swap `*this` and `copy` in our copy assignment operator—do you see why?[2] We'll therefore create our own, more effective `swap()` function for `Array<>` objects. Similar specializations of `std::swap<>()` exist for many Standard Library types.

[1]The actual `swap<>()` template is different in two aspects. First, it moves the objects if possible using move semantics. You'll learn all about move semantics in the next chapter. Second, it is only conditionally `noexcept`. Concretely, it is `noexcept` if its arguments can be moved without exceptions. Conditional `noexcept` specifications are a more advanced language feature we do not cover in this book.

[2]The reason we cannot use the `std::swap()` from within our copy assignment operator is that `std::swap()` in turn would use the copy assignment operator. In other words, calling `std::swap()` here would result in infinite recursion!

Because the member variables of `Array<>` are private, one option is to define `swap()` as a friend function. Here, we'll take a slightly different approach, one that is also followed by standard container templates such as `std::vector<>`. The idea is to first add an extra member function `swap()` to `Array<>` as follows:

```
template <typename T>
void Array<T>::swap(Array& other) noexcept
{
  std::swap(elements, other.elements);   // Swap two pointers (not their contents!)
  std::swap(size, other.size);           // Swap both sizes
}
```

You then use that to implement the conventional nonmember `swap()` function:

```
template <typename T>
void swap(Array<T>& one, Array<T>& other) noexcept
{
  one.swap(other);     // Forward to public member function
}
```

You can find the full source code of the improved `Array<>` template in `Ex16_01A`.

■ **Tip** Implement the assignment operator in terms of the copy constructor and a `noexcept swap()` function. This basic instance of the copy-and-swap idiom will ensure the desired all-or-nothing behavior for your assignment operators. While `swap()` can be added as a member function, convention dictates that making objects swappable involves defining a nonmember `swap()` function. Following this convention also ensures that the `swap()` function gets used by various algorithms of the Standard Library.

The copy-and-swap idiom can be used to make any nontrivial state modification exception safe, either inside other member functions or simply in the middle of any code. It comes in many variations, but the idea is always the same. First copy the object you want to change, then perform any number (zero or more) of risky steps onto that copy, and only once they all succeed commit the changes by swapping the state of the copy and the actual target object.

Instantiating a Class Template

The compiler instantiates a class template as a result of a definition of an object that has a type produced by the template. Here's an example:

```
Array<int> data {40};
```

When this statement is compiled, two things happen. The definition for the `Array<int>` class is created so that the type is identified, and the constructor definition is generated because it must be called to create the object. This is, all that the compiler needs to create the `data` object, so this is the only code that it provides from the templates at this point.