`IndexSequence<I...>`. With this parameter pack in hand, we can then use a simple **pack expansion** expression and braced initialization to populate our `std::array` return value:

```
template <typename T, std::size_t... I, typename F>
constexpr std::array<T, sizeof...(I)> generateArrayImpl(const F& func,
                                                         IndexSequence<I...>)
    // Return the results of calling F(i) for each i in the pack deduced as
    // the template parameter pack I.
{
    return { func(I)... };
}
```

The return statement in `generateArrayImpl` calls `func(I)` for each `I` in the range from 0 to the length of the returned `std::array`. The resulting pack of values is used to list-initialize the return value of the function; see Section 2.1."Braced Init" on page 215.

Finally, our implementation of `generateArray` forwards `func` to `generateArrayImpl`, using `MakeIndexSequence` to generate an object of type `IndexSequence<0,...,N-1>`:

```
template <typename T, std::size_t N, typename F>
constexpr std::array<T, N> generateArray(const F& func)
{
    return generateArrayImpl<T>(func, MakeIndexSequence<N>());
}
```

With these tools in hand and a support function to offset the array index with the year, it is now simple to define an array that is initialized at compile time with an appropriate range of results from calls to `startOfYear`:

```
constexpr std::time_t startOfEpochYear(int epochYear)
{
    return startOfYear(k_EPOCH_YEAR + epochYear);
}

constexpr std::array<std::time_t, k_MAX_YEAR - k_EPOCH_YEAR> k_YEAR_STARTS =
    generateArray<std::time_t, k_MAX_YEAR - k_EPOCH_YEAR>(startOfEpochYear);

static_assert(k_YEAR_STARTS[0]  == startOfYear(1970), "");
static_assert(k_YEAR_STARTS[50] == startOfYear(2020), "");
```

With this table available for our use, the implementation of `yearOfTimestamp` becomes a simple application of `std::upper_bound` to perform a binary search on the sorted array of start-of-year timestamps[19]:

[19] Among other improvements to language and library support for **constexpr** programming, C++20 added **constexpr** to many of the standard algorithms in `<algorithm>`, including `std::upper_bound`, which would make switching this implementation to be **constexpr** also trivial. Implementing a **constexpr** version of most algorithms in C++14 is, however, relatively simple (and in C++11 is still possible), so, given a need, providing **constexpr** versions of functions like this with less support from the Standard Library is straightforward.