```
template <typename T>
typename std::enable_if<std::is_trivially_copyable<T>::value>::type
copyArray(T* dst, const T* src, std::size_t n)
    // Copy src array of size n to dst array by dint of trivial copyability.
{
    std::memcpy(dst, src, n * sizeof *dst);  // Copy all Ts at once quickly.
}
```

The first overload is selected for types that are *not* trivially copyable; each `dst` array element is individually assigned a value from `src`. The second overload is selected *only* for trivially copyable types, providing an optimized assignment from `src` to `dst` via a single call to `std::memcpy`.

We can now use our generic `copyArray` to replace `copyArrayOfRecords` for assigning the value of an array of `FixedCapacityString` objects:

```
void f3()
{
    copyArray(duplicate, original, numStrings);   // generic fast array copy

    for (std::size_t i = 0; i < numStrings; ++i)  // same as in f2 (above)
    {
        assert(original[i] == duplicate[i]);
    }
}
```

The call to `copyArray` in `f3()` (above) invokes the optimized (`memcpy`-based) overload because `FixedCapacityString<30>` is a trivially copyable type. Similar code using `std::string`, being of *non*-trivially copyable type, would choose the unoptimized (element-by-element assignment) overload instead and, hence, would not be an appropriate record type for this use case.

Another potential benefit of trivially copyable types is that they can be safely copied into an array of **`unsigned char`** and inspected — e.g., for debugging purposes — as a “bag of bits,” provided we don’t access any bytes having indeterminate value. When copying an object of trivially copyable type to an **`unsigned char`** array, indeterminate values can come from two sources: (1) padding bytes and (2) any bytes in the object representation that correspond to uninitialized non**`static`** data members; see *Potential Pitfalls — Conflating arbitrary values with indeterminate values* on page 493. Our `FixedCapacityString` template was deliberately engineered to obviate padding bytes, but any unused bytes in `d_buffer` will have indeterminate value. If we want to make the entire footprint of `FixedCapacityString` inspectable as raw bytes, we will need to initialize the entire `d_buffer` in every user-provided constructor, e.g., using `std::memset(d_buffer, 0, N)`. Because only the *default* and *value* constructors are affected, the object remains trivially copyable albeit somewhat less runtime efficient to construct.