16.1 — Introduction to containers and arrays

The variable scalability challenge

Consider a scenario where we want to record the test scores for 30 students and calculate the average score for the class. To do so, we'll need 30 variables. We could define those like this:

```
// allocate 30 integer variables (each with a different name)
int testScore1 {};
int testScore2 {};
int testScore3 {};
// ...
int testScore30 {};
```

That's a lot of variables to define! And to calculate the average score for the class, we'd need to do something like this:

```
int average { (testScore1 + testScore2 + testScore3 + testScore4 + testScore5
     + testScore6 + testScore7 + testScore8 + testScore9 + testScore10
     + testScore11 + testScore12 + testScore13 + testScore14 + testScore15
     + testScore16 + testScore17 + testScore18 + testScore19 + testScore20
     + testScore21 + testScore22 + testScore23 + testScore24 + testScore25
     + testScore26 + testScore27 + testScore28 + testScore29 + testScore30)
     / 30; };
```

This is not only a lot of typing, it's also very repetitive (and it wouldn't be that hard to typo one of the numbers and not notice). And if we want to do anything with each of these values (like print them to the screen), we'd have to enter each of these variables names all over again.

Now let's say we need to modify our program to accommodate another student who was just added to the class. We'll have to scan the entire codebase and manually add in `testScore31` wherever relevant. Any time we modify existing code, we risk introducing new bugs. For example, it wouldn't be that hard to forget to update the divisor in the calculation of the average from `30` to `31`!

And this is with only 30 variables. Think about the case where we have hundreds or thousands of objects. When we need more than a few objects of the same type, defining individual variables simply doesn't scale.

We could put our data inside a struct:

```
struct testScores
{
// allocate 30 integer variables (each with a different name)
int score1 {};
int score2 {};
int score3 {};
// ...
int score30 {};
}
```

While this provides some additional organization for our scores (and allows us to pass them to functions more easily), it doesn't solve the core problem: we still need to define and access each test score object individually.

As you may have guessed, C++ has solutions to the above challenges. In this chapter, we'll introduce one such solution. And in the following chapters, we'll explore some other variants of that solution.

Containers

When you go to the grocery store to buy a dozen eggs, you (probably) aren't selecting 12 eggs individually and putting them in your cart (you aren't, right?). Instead, you're likely to select a single carton of eggs. The carton is a type of container, holding some predefined number of eggs (likely 6, 12, or 24). Now consider breakfast cereal, containing many little pieces of cereal. You definitely wouldn't want to store all these pieces in your pantry individually! Cereal often comes in a box, which is another container. We use containers all the time in real life because they make it easy to manage a collection of items.

Containers also exist in programming, to make it easier to create and manage (potentially large) collections of objects. In general programming, a **container** is a data type that provides storage for a collection of unnamed objects (called **elements**).

Key insight

We typically use containers when we need to work with a set of related values.

As it turns out, you've already been using one container type: strings! A string container provides storage for a collection of characters, which can then be output as text:

```
#include <iostream>
#include <string>

int main()
{
    std::string name{ "Alex" }; // strings are a container for characters
    std::cout << name; // output our string as a sequence of characters

    return 0;
}
```

The elements of a container are unnamed

While the container object itself typically has a name (otherwise how would we use it?), the elements of a container are unnamed. This is so that we can put as many elements in our container as we desire, without having to give each element a unique name! This lack of named elements is important, and is what distinguishes containers from other types of data structures. It is why plain structs (those that are just a collection of data members, like our `testScores` struct above) typically aren't considered containers -- their data members require unique names.

In the example above, our string container has a name (`name`), but the characters inside the container (`'A'`, `'l'`, `'e'`, `'x'`) do not.

But if the elements themselves are unnamed, how do we access them? Each container provides one or more methods to access its elements -- but exactly how depends on the type of container. We'll see our first example of this in the next lesson.

Key insight

The elements of a container do not have their own names, so that the container can have as many elements as we want without having to give each element a unique name.

Each container provides some method to access these elements, but how depends on the specific type of container.

The length of a container

In programming, the number of elements in a container is often called it's **length** (or sometimes **count**).

In lesson 5.9 -- Introduction to std::string, we showed how we could use the `length` member function of `std::string` to get the number of character elements in the string container:

```
#include <iostream>
#include <string>

int main()
{
    std::string name{ "Alex" };
    std::cout << name << " has " << name.length() << " characters\n";

    return 0;
}
```

This prints:

```
Alex has 4 characters
```

In C++, the term **size** is also commonly used for the number of elements in a container. This is an unfortunate choice of nomenclature, as the term “size” can also refer to the number of bytes of memory used by an object (as returned by the `sizeof` operator).

We’ll prefer the term “length” when referring to the number of elements in a container, and use the term “size” to refer to amount of storage required by an object.

Container operations

Back to our carton of eggs for a moment. What can you do with such a carton? Well, first you can acquire a carton of eggs. You can open the carton of eggs and select an egg, and then do whatever you want with that egg. You can remove an existing egg from the carton, or add a new egg to an empty space. You can also count the number of eggs in the carton.

Similarly, containers typically implement a significant subset of the following operations:

- Create a container (e.g. empty, with storage for some initial number of elements, from a list of values).
- Access to elements (e.g. get first element, get last element, get any element).
- Insert and remove elements.
- Get the number of elements in the container.

Containers may also provide other operations (or variations on the above) that assist in managing the collection of elements.

Modern programming languages typically provide a variety of different container types. These container types differ in terms of which operations they actually support, and how performant those operations are. For example, one container type might provide fast access to any element in the container, but not support insertion or removal of elements. Another container type might provide fast insertion and removal of elements, but only allow access to elements in sequential order.

Every container has a set of strengths and limitations. Picking the right container type for the task you are trying to solve can have a huge impact on both code maintainability and overall performance. We will discuss this topic further in a future lesson.

Element types

In most programming languages (including C++), containers are **homogenous**, meaning the elements of a container are required to have the same type.

Some containers use a preset element type (e.g. a string typically has `char` elements), but more often the element type can be set by the user of the container. In C++, containers are typically implemented as class templates, so that the user can provide the desired element type as a template type argument. We'll see an example of this next lesson.

This makes containers flexible, as we do not need to create a new container class for each element type we wish to hold. Instead, we just instantiate the class template with the desired element type, and we're ready to go.

As an aside…

The opposite of a homogenous container is a **heterogenous** container, which allows elements to be different types. Heterogeneous containers are typically supported by scripting languages (such as Python).

Containers in C++

The **Containers library** is a part of the C++ standard library that contains various class types that implement some common types of containers. A class type that implements a container is sometimes called a **container class**. The full list of containers in the Containers library is documented here.

In C++, the definition of "container" is narrower than the general programming definition. Only the class types in the Containers library are considered to be containers in C++. We will use the term "container" when talking about containers in general, and "container class" when talking specifically about the container class types that are part of the Containers library.

For advanced readers

The following types are containers under the general programming definition, but are not considered to be containers by the C++ standard:

- C-style arrays
- `std::string`
- `std::vector<bool>`

To be a container in C++, the container must implement all of the requirements listed here. Note that these requirements include the implementation of certain member functions -- this implies that C++ containers must be class types! The types listed above do not implement all of these requirements.

However, because `std::string` and `std::vector<bool>` implement most of the requirements, they behave like containers in most circumstances. As a result, they are sometimes called “pseudo-containers”.

Of the provided container classes, `std::vector` and `std::array` see by far the most use, and will be where we focus the bulk of our attention. The other containers classes are typically only used in more specialized situations.

Introduction to arrays

An **array** is a container data type that stores a sequence of values **contiguously** (meaning each element is placed in an adjacent memory location, with no gaps). Arrays allow fast, direct access to any element. They are conceptually simple and easy to use, making them the first choice when we need to create and work with a set of related values.

C++ contains three primary array types: (C-style) arrays, the `std::vector` container class, and the `std::array` container class.

(C-style) arrays were inherited from the C language. For backwards compatibility, these arrays are defined as part of the core C++ language (much like the fundamental data types). The C++ standard calls these “arrays”, but in modern C++ these are often called **C arrays** or **C-style arrays** in order to differentiate them from the similarly named `std::array`. C-style arrays are also sometimes called “naked arrays”, “fixed-sized arrays”, “fixed arrays”, or “built-in arrays”. We’ll prefer the term “C-style array”, and use “array” when discussing array types in general. By modern standards, C-style arrays behave strangely and they are dangerous. We will explore why in a future chapter.

To help make arrays safer and easier to use in C++, the `std::vector` container class was introduced in C++03. `std::vector` is the most flexible of the three array types, and has a bunch of useful capabilities that the other array types don’t.

Finally, the `std::array` container class was introduced in C++11 as a direct replacement for C-style arrays. It is more limited than `std::vector`, but can also be more efficient, especially for smaller arrays.

All of these array types are still used in modern C++ in different capacities, so we will cover all three to varying degrees.

Moving forward

In the next lesson, we'll introduce our first container class, `std::vector`, and begin our journey to show how it can efficiently solve the challenge we presented at the top of this lesson. We'll spend a lot of time with `std::vector`, as we'll need to introduce quite a few new concepts, and address some additional challenges along the way.

One nice thing is that all of the container classes have similar interfaces. Thus, once you've learned how to use one container (e.g. `std::vector`), learning the others (e.g. `std::array`) is much simpler. For future containers (e.g. `std::array`), we'll cover the notable differences (and reiterate the most important points).

Author's note

A quick note on terminology:

- We will use `container classes` when we're talking about something that applies to most or all the standard library containers classes.
- We will use `array` when we're talking about something that generally applies to all array types, even ones implemented in other programming languages.

`std::vector` falls into both of these categories, so even though we may be using different terms, it's still applicable to `std::vector`.

Okay, ready?

Let's goooooooooooooooooooooooooooooooooooooooooooooo.