1 Print all the things!

This exercise covers *function templates*, possibly *variadic templates*, *SFINAE*, *containers*, *iterators*, `std::tuple`, possibly `std::integer_sequence` and *traits*.

In this exercise we intend to use templates, SFINAE and traits to construct a general print function that can print as much as possible.

The user should be able to simply do

```
print(std::cout, data);
```

to print whatever `data` is.

Your job is to make sure that this works for all containers, strings, `std::pair`, `std::tuple` and all data types that has a `operator<<` defined for streams.

`print` should also work recursively, so if we have a container of containers it should be able to print each individual container inside the container as well.

Of course it should be possible to extend this with other types, such as `std::variant`, `std::optional` and so on (but this is *optional* (pun intended)).

All containers and c-arrays (except `std::string`) should be handled with one case. Strings (both `std::string` and c-strings) should be handled separately with normal function overloads (no templates required) because otherwise it will print each individual character, which is not what we want for strings.

Handling `std::tuple` is a bit weirder. To access the data inside a `std::tuple` we must use `std::get`. There are two versions of `std::get`, one that takes indices and one that takes types. Since we can have duplicates of types it would be preferable to use the index version.

Example:

```
std::tuple<int, double, int> t {1, 2.3, 4};
cout << std::get<0>(t) << endl
     << std::get<1>(t) << endl
     << std::get<2>(t) << endl;
```

Will print `1 2.3 4`.

There are many ways to iterate through each element of `std::tuple`, all of which are acceptable for this exercise. However, for most cases you will need to find how many elements there are in a tuple in order to know how many elements you should iterate through. For this you can use `std::tuple_size` which is a trait class that returns the size of the tuple.

Example:

```
std::tuple<int, double, int> t{1, 2.3, 4};
cout << std::tuple_size<decltype(t)>::value << std::endl;
```

will print `3`.

`std::pair` can be handled in two ways: either you create a specific function overload which specifically handles `std::pair` *OR* you note that `std::get` and `std::tuple_size` works for both `std::tuple`, `std::pair` and `std::array`.

This means that you can create a general case which handles all three of these. However there is one problem if you choose that solution; `std::array` is handled by both the container case and the tuple case, so there will be an ambiguity. The easiest solution for this is to simply create a function overload for `std::array` specifically and handle it separately.

One possible way to iterate through tuples is with the help of `std::integer_sequence`, do note that there are easier ways to do this using variadic templates, but using the `std::integer_sequence` is how the STL does it. Implementing the tuple case with `std::integer_sequence` is a challenge for those who want to learn a bit more about C++ and the STL. Look at cppreference for more information about it.

Hint: To detect if a type is a container, you can check if it has iterators. Recall that C-array also have iterators, but only if we think more generally.