1 Reduce

In `reduce.cc` there is a program given which uses a function template called `reduce`. Reduce is a common operation in functional programming which takes a sequence of values and reduces them to a single value by repeatedly applying some binary operator.

Example: Given the sequence `1,2,3,4,5` and the operator `+` we get `1+2+3+4+5 = 15`.

Reduce should take three parameters:

- a reference to an array of arbitrary size and type,
- an initial value, which should be default initialized if no value is passed by the caller,
- a function pointer to a function which takes two `const` references to some arbitrary type, and return a single value of the same type. If the caller omits this parameter it should default to addition.

  Note: you will have to implement a function template which represents this default behaviour.

`reduce` should first apply the function on the initial value and the first element in the array, then apply the operator on the result and the second element in the array, and so on.

In this exercise you are to implement this reduce function template such that the given code compiles and produces the following output:

```
15
3.1415
120
hello world
```

2 Factorial

In this exercise you are to implement the function template `factorial` such that the program given in `factorial.cc` compiles and produces the following output:

```
1
1
120
2432902008176640000
```

Note that the `factorial` function takes no argument, only a template parameter. This allows the compiler to know a lot of things about your function call that it wouldn't normally be able to know, since the compiler know all values that this function is called with. The compiler might even perform the entire calculation during compile-time.

Restriction: Try to solve this exercise without any if-statements.

Note: The value of 20! does not fit in a regular `int`, use `unsigned long long` instead.

3 Serialization

When dealing with sending and receiving data to and from various sources it is important to keep the data in such a structure that both ends of the communication can understand it. The process of transforming your data to and from this kind of format is called *serialization*.

In the file `serialization.cc` basic facilities for serializing to and from a stream has been implemented in the function templates `write` and `read`. There is also a struct `Product` given, and an `operator<<` which allows the user to print `Product` in a human-readable way to a stream (note however that this `operator<<` should not be used during serialization).

Your job in this exercise is to extend the `read` and `write` functionality so that it works for the `Product` struct, and for `std::string` of arbitrary length.

You should also add overloads of `read` and `write` for the type `std::string`. They should be able to handle any type of string that contains any combination of letters, digits and whitespace.

Hint: Write your strings as `#my string here#` when serializing, that way you can use `ignore` and `getline` to find the string between the two `#`.

The `Product` overload should only use `read` and `write` to serialize its respective data members, you should not have to manually perform any stream operations when serializing `Product`.