Template Argument Deduction for Class Templates

This lesson will make you learn how to skip make_Type functions to construct a template object.

Do you often use `make_Type` functions to construct a templated object (like `std::make_pair`)? With C++17 you can forget about (most of) them and just use a regular constructor.

C++17 has filled a gap in the deduction rules for templates. Now template deduction can occur for standard class templates and not just for functions. That also means that a lot of your code that uses `make_Type` functions can now be removed.

For instance, to create an `std::pair` object, it was usually more comfortable to write:

```
auto myPair = std::make_pair(42, "hello world");
```

Rather than:

```
std::pair<int, std::string> myPair(42, "hello world");
```

```
#include <iostream>
#include <utility>
using namespace std;

int main() {
  auto myPair = std::make_pair(42, "hello world");
  cout << myPair.first << " " << myPair.second << endl;
}
```

Because `std::make_pair` is a template function, the compiler can perform the deduction of function template arguments and there's no need to write:

```
auto myPair = std::make_pair<int, std::string>(42, "hello world");
```

Now, since C++17, the conformant compiler will nicely deduce the template parameter types for class templates too!

> The feature is called *“Class Template Argument Deduction”* or *CTAD* in short.

In our example, you can now write:

```
using namespace std::string_literals;
std::pair myPair{42, "hello world"s};     // deduced automatically!
```

CTAD also works with copy initialisation and when allocating memory through `new()`:

```
auto otherPair = std::pair{42, "Hello"s}; // also deduced
auto ptr = new std::pair{42, "World"s};   // for new
```

CTAD can substantially reduce complex constructions like:

```
#include <iostream>
#include <mutex>
#include <shared_mutex>
using namespace std;

int main(){
  // lock guard:
  std::shared_timed_mutex mut;
  std::lock_guard<std::shared_timed_mutex> lck(mut);

  // array:
  std::array<int, 3> arr {1, 2, 3};
  for(int i = 0; i < 3; i++){
    cout << arr[i] << " ";
  }
}
```

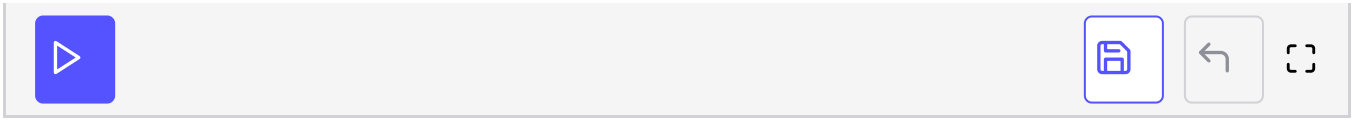
Can now become:

```
#include <iostream>
```

```
#include <mutex>
#include <shared_mutex>
using namespace std;


int main(){
  // lock guard:
  std::shared_timed_mutex mut;
  std::lock_guard lck(mut);

  //array
  std::array arr { 1, 2, 3 };
  for(int i = 0; i < 3; i++){
    cout << arr[i] << " ";
  }
}
```



Note, that partial deduction cannot happen, you have to specify all the template parameters or none:

```
std::tuple t(1, 2, 3); // OK: deduction
std::tuple<int,int,int> t(1, 2, 3); // OK: all arguments are provided
std::tuple<int> t(1, 2, 3); // Error: partial deduction
```

With this feature, a lot of `make_Type` functions might not be needed - especially those that “emulate” template deduction for classes.

Still, there are factory functions that do additional work. For example, `std::make_shared` - it not only creates `shared_ptr`, but also makes sure the control block, and the pointed object is allocated in one memory region:

```
// control block and int might be in different places in memory
    std::shared_ptr<int> p(new int{10});
// the control block and int are in the same contiguous memory section
    auto p2 = std::make_shared<int>(10);
```

How does template argument deduction for classes work? Let’s move to next lesson called the “Deduction Guides” area.