Structured Binding Declarations

This lesson sheds light on the modifications made to tuple functionality.

Do you often work with tuples or pairs?

If not, then you should probably start looking into those handy types. Tuples enable you to bundle data ad-hoc with excellent library support instead of creating small custom types. The language features like structured binding make the code even more expressive and concise.

Consider a function that returns two results in a pair:

```
std::pair<int, bool> InsertElement(int el) { ... }
```

You can write:

```
auto ret = InsertElement(...)
```

And then refer to `ret.first` or `ret.second`. However, referring to values as `.first` or `.second` is also not expressive - you can easily confuse the names, and it's hard to read. Alternatively, you can leverage `std::tie` which will unpack the tuple/pair into local variables:

```
int index { 0 };
bool flag { false };
std::tie(index, flag) = InsertElement(10);
```

Such code might be useful when you work with `std::set::insert` which returns `std::pair`:

```
#include <iostream>
#include <set>
#include <tuple>

int main(){
  std::set<int> mySet;
```

```
  std::set<int>::iterator iter;
  bool inserted;


  std::tie(iter, inserted) = mySet.insert(10);
  if (inserted)
    std::cout << "Value was inserted\n";
}
```

As you see, such a simple pattern - returning several values from a function - requires several lines of code. Fortunately, C++17 makes it much simpler!

With C++17 the code can be more compact:

```
std::set<int> mySet;
auto [iter, inserted] = mySet.insert(10);
```

Now, instead of `pair.first` and `pair.second`, you can use variables with concrete names. In addition, you have one line instead of three, and the code is easier to read.

The code is also safer as `iter` and `inserted` are initialised in the expression.

> Such syntax is called a **structured binding expression**.

In C++17, the syntax for accessing and binding to tuples has taken a more concise form. Find out more in the next lesson.