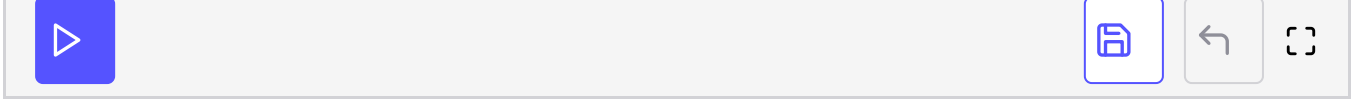
A Separate Structure

The lesson elaborates the use of a separate struct to return multiple values.

The outputs seem to represent related data. That's why it's probably a good idea to wrap them into a `struct` called `SelectionData`.

```
#include <iostream>
#include <tuple>
#include <optional>
#include <variant>
#include <vector>

class ObjSelection
{
  public:
  bool IsValid() const { return true; }
};

struct SelectionData
{
  bool anyCivilUnits { false };
  bool anyCombatUnits { false };
  int numAnimating { 0 };
};

//   you can rewrite the function into
std::pair<bool, SelectionData> CheckSelectionVer3(const ObjSelection &objList)
{
  SelectionData out;
  if (!objList.IsValid())
    return {false, out};
  // scan...
  return {true, out};
}

int main(){
  ObjSelection sel;

  bool anyCivilUnits = false;
  bool anyCombatUnits = false;
  int numAnimating = 0;
  //   the caller site:
  if (auto [ok, selData] = CheckSelectionVer3(sel); ok)
  {
    std::cout << "ok...\n";
  }
}
```



The code uses `std::pair` so we still preserve the success flag, it's not the part of the new `struct`.

The main advantage that we achieved here is the improved code structure and extensibility. If you want to add a new parameter, then extend the structure. Previously - with a list of output parameters in the function declaration - you'd have to update much more code.

But isn't `std::pair<bool, MyType>` similar to the concept of `std::optional`?

So let's see how we can use a struct with `std::optional` in the next lesson.