- Solutions

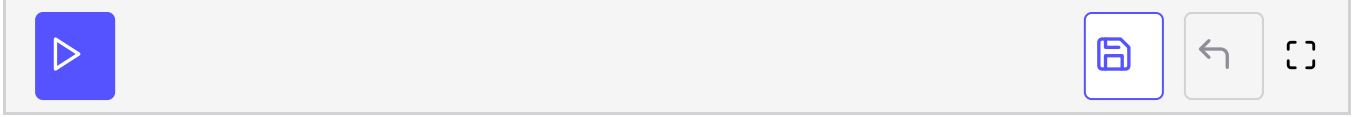
Let's look at the solution reviews of previously solved exercises.

WE'LL COVER THE FOLLOWING

Solution Review of Problem Statement 1

```
// templatePolicyMap.cpp

#include <iostream>
#include <map>
#include <unordered_map>

struct MyInt{
    explicit MyInt(int v):val(v){}
    int val;
};

struct MyHash{
    std::size_t operator()(MyInt m) const {
        std::hash<int> hashVal;
        return hashVal(m.val);
    }
};

struct MyEqual{
    bool operator () (const MyInt& fir, const MyInt& sec) const {
        return fir.val == sec.val;
    }
};

struct MySmaller{
    bool operator () (const MyInt& fir, const MyInt& sec) const {
        return fir.val < sec.val;
```

```
    }

};


std::ostream& operator << (std::ostream& strm, const MyInt& myIn){
    strm << "MyInt(" << myIn.val << ")";
    return strm;
}

int main(){

    std::cout << std::endl;

    typedef std::unordered_map<MyInt, int, MyHash, MyEqual> MyUnorderedMap;

    std::cout << "MyUnorderedMap: ";
    MyUnorderedMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

    for(auto m : myMap) std::cout << '{' << m.first << ", " << m.second << "}";

    std::cout << std::endl;

    typedef std::map<MyInt, int, MySmaller> MyOrderedMap;

    std::cout << "MyOrderedMap: ";
    MyOrderedMap myMap2{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};

    for(auto m : myMap2) std::cout << '{' << m.first << ", " << m.second << "}";

    std::cout << "\n\n";

}
```



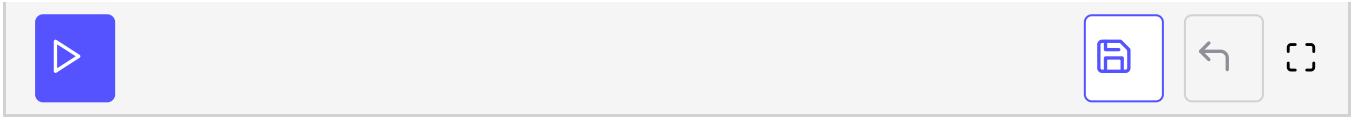
Explanation

In order to use `MyInt` as a key in an `std::map`, `MyInt` has to support an ordering. The class `MySmaller` supports ordering on `MyInt`. The `typedef` in line 50 allows us to use `MyInt` as a key (line 53) in `MyOrderedMap`.

Solution Review of Problem Statement 2

```
// templatePolicyDecreasing.cpp

#include <iostream>
#include <map>


int main(){

    std::cout << std::endl;

    std::map<std::string, int, std::greater<std::string>> myDecreaseMap{{"Grimm", 1}, {"Huber
```

```
    std::cout << "myDecreaseMap: ";
    for(auto m : myDecreaseMap) std::cout << '{' << m.first << ", " << m.second << "}";

    std::cout << "\n\n";

}
```



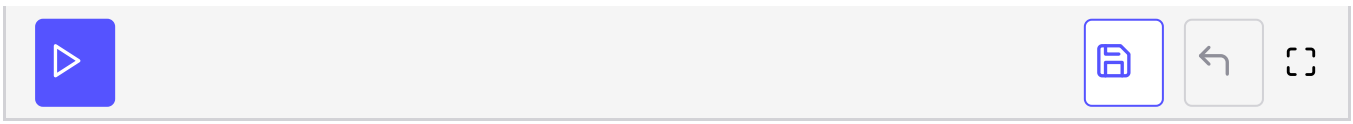
Explanation

In the above example, we have stored in line 11 the keys in increasing order in the `std::map`. In line 14, we have used `for` loop to print the element stored in the `myDecreaseMap`.

Solution Review of Problem Statement 3

```
// TemplateTraitsPrimary.cpp

#include <iostream>
#include <type_traits>

using namespace std;

template <typename T>
void getPrimaryTypeCategory(){

  cout << boolalpha << endl;

  cout << "is_void<T>::value: " << is_void<T>::value << endl;
  cout << "is_integral<T>::value: " << is_integral<T>::value << endl;
  cout << "is_floating_point<T>::value: " << is_floating_point<T>::value << endl;
  cout << "is_array<T>::value: " << is_array<T>::value << endl;
  cout << "is_pointer<T>::value: " << is_pointer<T>::value << endl;
  cout << "is_reference<T>::value: " << is_reference<T>::value << endl;
  cout << "is_member_object_pointer<T>::value: " << is_member_object_pointer<T>::value << end
  cout << "is_member_function_pointer<T>::value: " << is_member_function_pointer<T>::value <<
  cout << "is_enum<T>::value: " << is_enum<T>::value << endl;
  cout << "is_union<T>::value: " << is_union<T>::value << endl;
  cout << "is_class<T>::value: " << is_class<T>::value << endl;
  cout << "is_function<T>::value: " << is_function<T>::value << endl;
  cout << "is_lvalue_reference<T>::value: " << is_lvalue_reference<T>::value << endl;
  cout << "is_rvalue_reference<T>::value: " << is_rvalue_reference<T>::value << endl;

  cout << endl;

}

int main(){

    getPrimaryTypeCategory<void>();
    getPrimaryTypeCategory<short>();
    getPrimaryTypeCategory<double>();
```

```
    getPrimaryTypeCategory<int []>();
    getPrimaryTypeCategory<int*>();
    getPrimaryTypeCategory<int&>();

    struct A{
        int a;
        int f(double){return 2011;}
    };
    getPrimaryTypeCategory<int A::*>();
    getPrimaryTypeCategory<int (A::*)(double)>();
    enum E{
        e= 1,
    };
    getPrimaryTypeCategory<E>();
    union U{
      int u;
    };
    getPrimaryTypeCategory<U>();
    getPrimaryTypeCategory<string>();
    getPrimaryTypeCategory<int * (double)>();
    getPrimaryTypeCategory<int&>();
    getPrimaryTypeCategory<int&&>();

}
```



Explanation

In the example mentioned above, we have defined the function `getPrimaryTypeCategory` in line 9 which takes the built-in type of C++ and returns true in front of the type we have defined in lines (13 – 26). We're checking for each type by passing the type in the function.

Let's move on to tag dispatching in idioms and patterns in the next lesson.