- Solutions

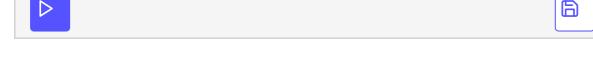
Let's look at the solution reviews of previously solved exercises.

WE'LL COVER THE FOLLOWING Solution Review of Problem Statement 1 Explanation Solution Review of Problem Statement 2 Explanation Solution Review of Problem Statement 3 Explanation

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;</pre>
```

```
};
std::ostream& operator << (std::ostream& strm, const MyInt& myIn){</pre>
               strm << "MyInt(" << myIn.val << ")";</pre>
               return strm;
}
int main(){
               std::cout << std::endl;</pre>
               typedef std::unordered_map<MyInt, int, MyHash, MyEqual> MyUnorderedMap;
               std::cout << "MyUnorderedMap: ";</pre>
               \label{eq:myunorderedMap myMap} $$ MyInt(-2), -2\}, $$ MyInt(-1), -1\}, $$ MyInt(0), 0\}, $$ MyInt(1), 1$$ $$ $$ MyInt(2), 0$$, $$ MyInt(2)
               for(auto m : myMap) std::cout << '{' << m.first << ", " << m.second << "}";</pre>
               std::cout << std::endl;</pre>
               typedef std::map<MyInt, int, MySmaller> MyOrderedMap;
               std::cout << "MyOrderedMap: ";</pre>
               MyOrderedMap myMap2{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};
               for(auto m : myMap2) std::cout << '{' << m.first << ", " << m.second << "}";</pre>
               std::cout << "\n\n";</pre>
}
```



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";
}</pre>
```

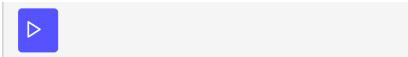
Explanation

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;</pre>
  cout << "is_void<T>::value: " << is_void<T>::value << endl;</pre>
  cout << "is_integral<T>::value: " << is_integral<T>::value << endl;</pre>
  cout << "is_floating_point<T>::value: " << is_floating_point<T>::value << endl;</pre>
  cout << "is_array<T>::value: " << is_array<T>::value << endl;</pre>
  cout << "is_pointer<T>::value: " << is_pointer<T>::value << endl;</pre>
  cout << "is_reference<T>::value: " << is_reference<T>::value << endl;</pre>
  cout << "is_member_object_pointer<T>::value: " << is_member_object_pointer<T>::value << end</pre>
  cout << "is member function pointer<T>::value: " << is member function pointer<T>::value <</pre>
  cout << "is_enum<T>::value: " << is_enum<T>::value << endl;</pre>
  cout << "is_union<T>::value: " << is_union<T>::value << endl;</pre>
  cout << "is_class<T>::value: " << is_class<T>::value << endl;</pre>
  cout << "is_function<T>::value: " << is_function<T>::value << endl;</pre>
  cout << "is lvalue reference<T>::value: " << is lvalue reference<T>::value << endl;</pre>
  cout << "is_rvalue_reference<T>::value: " << is_rvalue_reference<T>::value << endl;</pre>
  cout << endl;</pre>
}
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.