Structured binding declarations

Tom Latham

Structured binding declarations

- Introduced in C++17, structured binding declarations allow new names to be bound to existing objects, specifically, to sub-objects or elements of the initialiser, which should either be:
    - a C-style array
    - a tuple-like type (e.g. std::tuple, std::pair, std::array)
    - a struct or a class that has (some) public data members
- While this may sound a bit complicated, in practice it is quite straightforward and helps to make code much more readable
- For full technical info see: https://en.cppreference.com/w/cpp/language/structured_binding

Example usage: improved looping over maps

- Structured bindings offer an improved way of looping over maps

```
std::map<std::string,double> wages;

wages["Jane"] = 24.52;
wages["Pablo"] = 22.86;

...

for ( const auto& elem : wages )
{
      std::cout << elem.first
                << " earns £"
                << elem.second
                << " per hour\n";
}
```

Example usage: improved looping over maps

- Structured bindings offer an improved way of looping over maps
- You can address the key and value of each element using meaningful names
- This makes the code much more understandable

```cpp
std::map<std::string,double> wages;

wages["Jane"] = 24.52;
wages["Pablo"] = 22.86;

...

for ( const auto& [name, wage] : wages )
{
      std::cout << name
                << " earns £"
                << wage
                << " per hour\n";
}
```

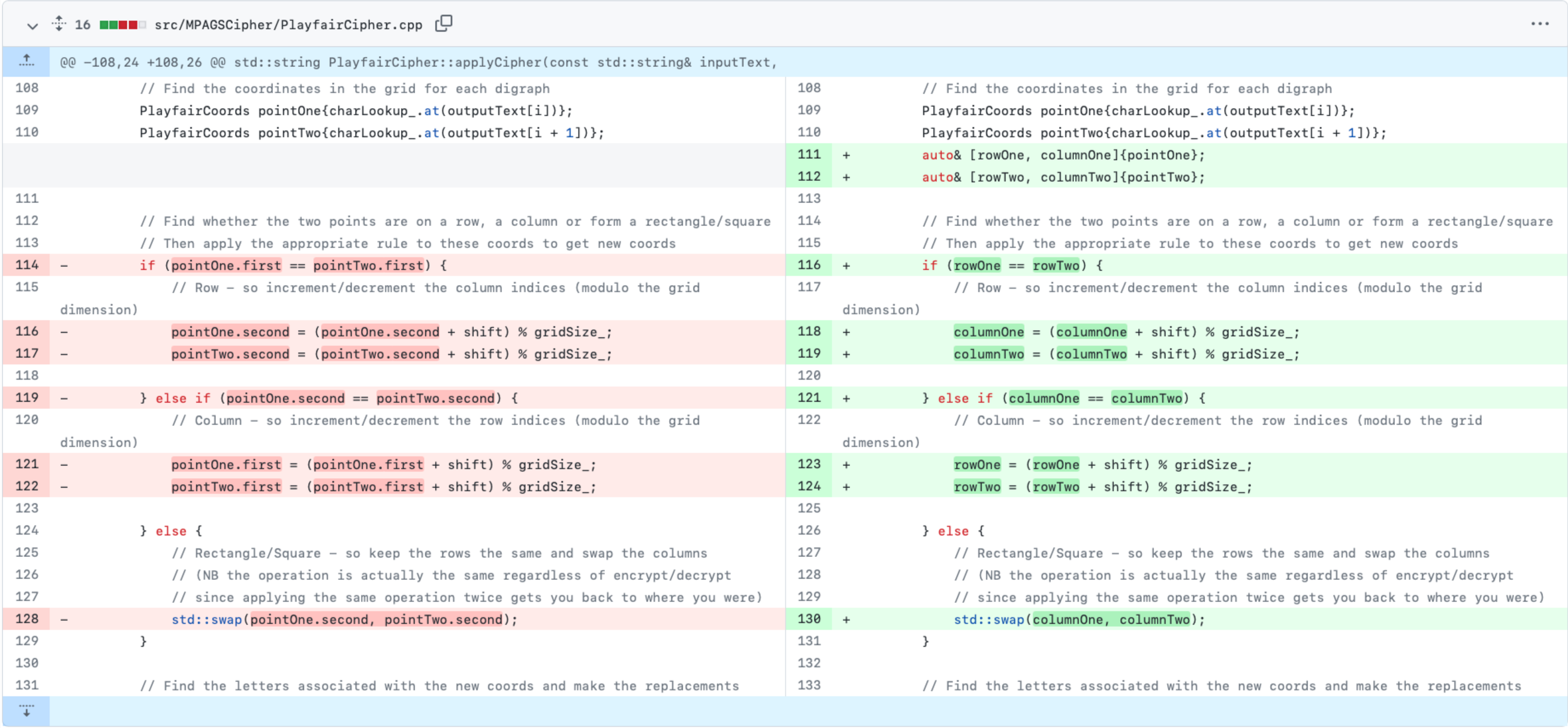
Example usage: improved looping over maps

- NB that we can use qualifiers such as const and the reference symbol

- Without the reference qualifier we would copy each element of the map and the new identifiers would refer to the key and value of the copy

```
std::map<std::string,double> wages;

wages["Jane"] = 24.52;
wages["Pablo"] = 22.86;

...

for ( const auto& [name, wage] : wages )
{
      std::cout << name
                << " earns £"
                << wage
                << " per hour\n";
}
```

Example: improving clarity in PlayfairCipher::applyCipher



16 src/MPAGSCipher/PlayfairCipher.cpp

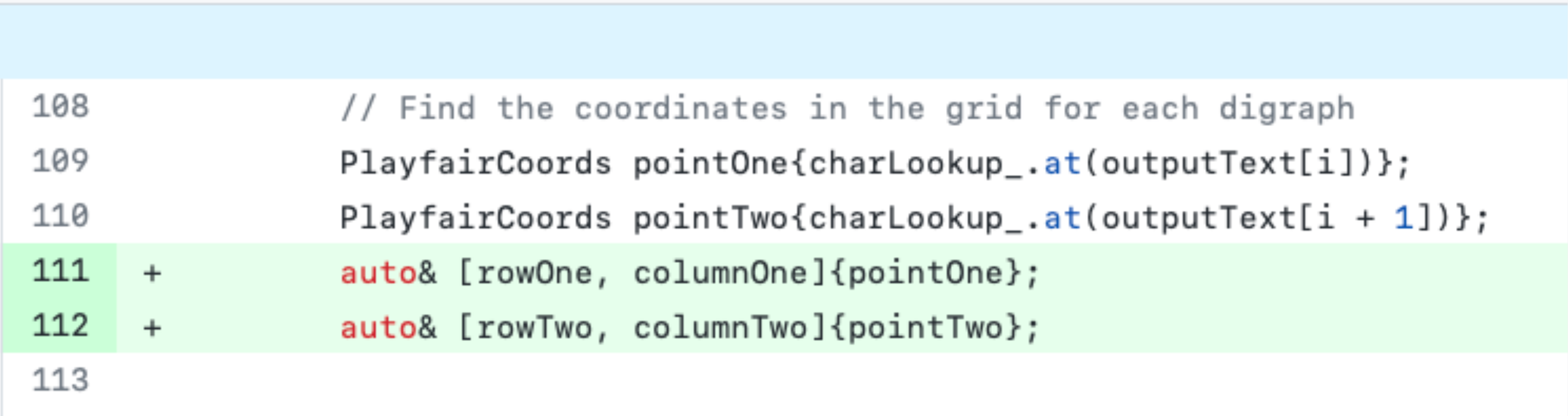
```
@@ -108,24 +108,26 @@ std::string PlayfairCipher::applyCipher(const std::string& inputText,
```

```
          // Find the coordinates in the grid for each digraph
          PlayfairCoords pointOne{charLookup_.at(outputText[i])};
          PlayfairCoords pointTwo{charLookup_.at(outputText[i + 1])};

          // Find whether the two points are on a row, a column or form a rectangle/square
          // Then apply the appropriate rule to these coords to get new coords
-         if (pointOne.first == pointTwo.first) {
              // Row - so increment/decrement the column indices (modulo the grid dimension)
-             pointOne.second = (pointOne.second + shift) % gridSize_;
-             pointTwo.second = (pointTwo.second + shift) % gridSize_;

-         } else if (pointOne.second == pointTwo.second) {
              // Column - so increment/decrement the row indices (modulo the grid dimension)
-             pointOne.first = (pointOne.first + shift) % gridSize_;
-             pointTwo.first = (pointTwo.first + shift) % gridSize_;

          } else {
              // Rectangle/Square - so keep the rows the same and swap the columns
              // (NB the operation is actually the same regardless of encrypt/decrypt
              // since applying the same operation twice gets you back to where you were)
-             std::swap(pointOne.second, pointTwo.second);
          }

          // Find the letters associated with the new coords and make the replacements
```

```
          // Find the coordinates in the grid for each digraph
          PlayfairCoords pointOne{charLookup_.at(outputText[i])};
          PlayfairCoords pointTwo{charLookup_.at(outputText[i + 1])};
+         auto& [rowOne, columnOne]{pointOne};
+         auto& [rowTwo, columnTwo]{pointTwo};

          // Find whether the two points are on a row, a column or form a rectangle/square
          // Then apply the appropriate rule to these coords to get new coords
+         if (rowOne == rowTwo) {
              // Row - so increment/decrement the column indices (modulo the grid dimension)
+             columnOne = (columnOne + shift) % gridSize_;
+             columnTwo = (columnTwo + shift) % gridSize_;

+         } else if (columnOne == columnTwo) {
              // Column - so increment/decrement the row indices (modulo the grid dimension)
+             rowOne = (rowOne + shift) % gridSize_;
+             rowTwo = (rowTwo + shift) % gridSize_;

          } else {
              // Rectangle/Square - so keep the rows the same and swap the columns
              // (NB the operation is actually the same regardless of encrypt/decrypt
              // since applying the same operation twice gets you back to where you were)
+             std::swap(columnOne, columnTwo);
          }

          // Find the letters associated with the new coords and make the replacements
```

Example: improving clarity in PlayfairCipher::applyCipher

Declaration of our structured bindings:

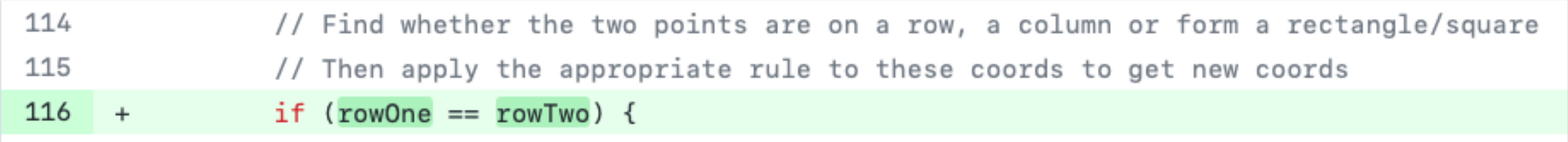


```
            // Find the coordinates in the grid for each digraph
            PlayfairCoords pointOne{charLookup_.at(outputText[i])};
            PlayfairCoords pointTwo{charLookup_.at(outputText[i + 1])};
+           auto& [rowOne, columnOne]{pointOne};
+           auto& [rowTwo, columnTwo]{pointTwo};

```

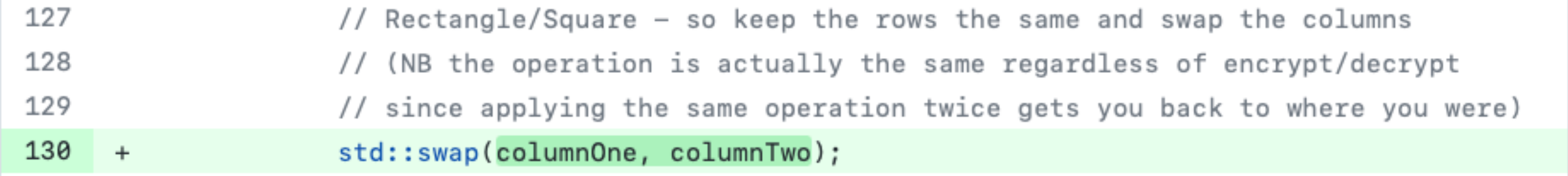
We give meaningful names to the ‘first’ and ‘second’ elements of the pair

Example: improving clarity in PlayfairCipher::applyCipher

Operations immediately become more understandable:



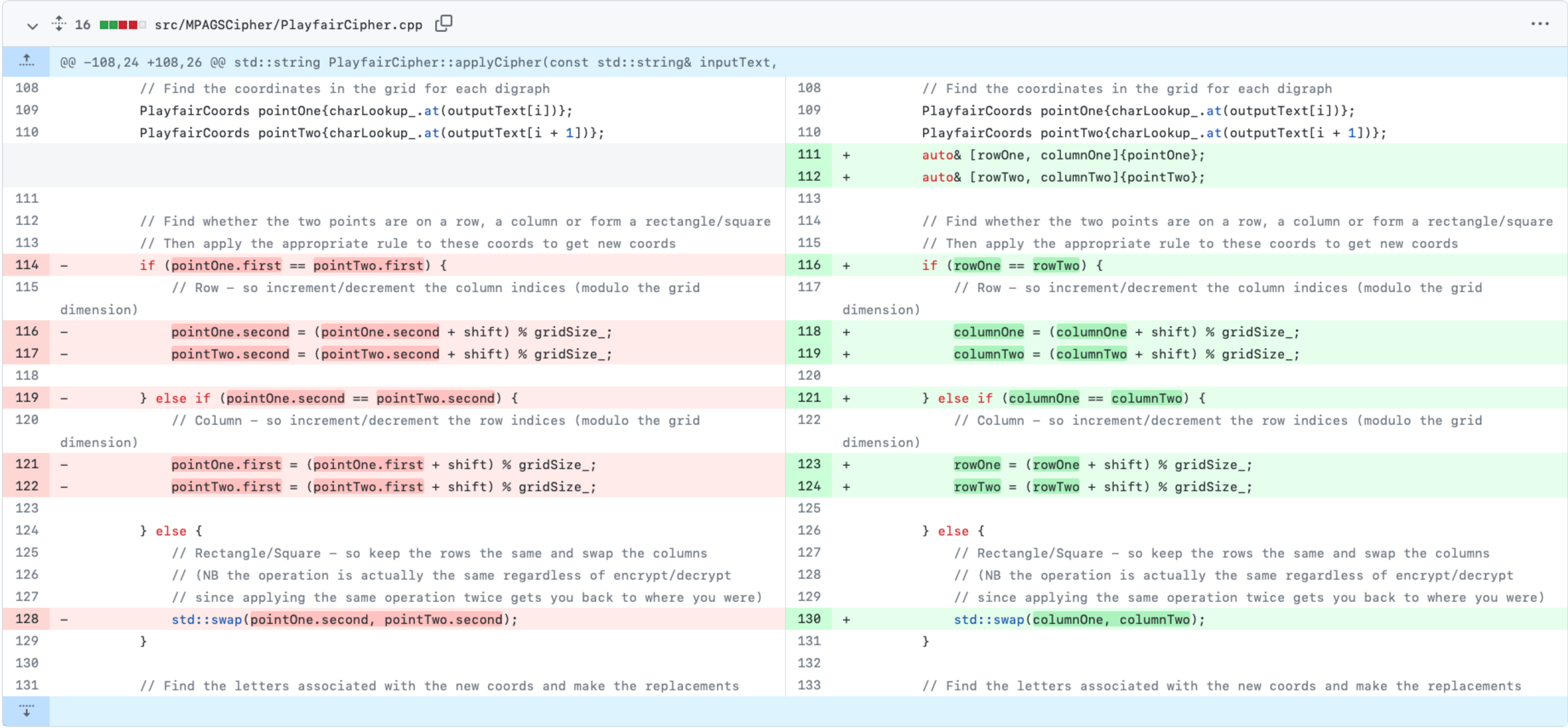
```
            // Find whether the two points are on a row, a column or form a rectangle/square
            // Then apply the appropriate rule to these coords to get new coords
+           if (rowOne == rowTwo) {
```



```
                // Rectangle/Square - so keep the rows the same and swap the columns
                // (NB the operation is actually the same regardless of encrypt/decrypt
                // since applying the same operation twice gets you back to where you were)
+               std::swap(columnOne, columnTwo);
```

And because we used the reference specifier in the declaration, we are acting on the original objects. Point1 and Point2 are updated to the new co-ords.

Example: improving clarity in PlayfairCipher::applyCipher



16 src/MPAGSCipher/PlayfairCipher.cpp

```diff
@@ -108,24 +108,26 @@ std::string PlayfairCipher::applyCipher(const std::string& inputText,
         // Find the coordinates in the grid for each digraph
         PlayfairCoords pointOne{charLookup_.at(outputText[i])};
         PlayfairCoords pointTwo{charLookup_.at(outputText[i + 1])};
+        auto& [rowOne, columnOne]{pointOne};
+        auto& [rowTwo, columnTwo]{pointTwo};

         // Find whether the two points are on a row, a column or form a rectangle/square
         // Then apply the appropriate rule to these coords to get new coords
-        if (pointOne.first == pointTwo.first) {
+        if (rowOne == rowTwo) {
             // Row - so increment/decrement the column indices (modulo the grid
dimension)
-            pointOne.second = (pointOne.second + shift) % gridSize_;
-            pointTwo.second = (pointTwo.second + shift) % gridSize_;
+            columnOne = (columnOne + shift) % gridSize_;
+            columnTwo = (columnTwo + shift) % gridSize_;

-        } else if (pointOne.second == pointTwo.second) {
+        } else if (columnOne == columnTwo) {
             // Column - so increment/decrement the row indices (modulo the grid
dimension)
-            pointOne.first = (pointOne.first + shift) % gridSize_;
-            pointTwo.first = (pointTwo.first + shift) % gridSize_;
+            rowOne = (rowOne + shift) % gridSize_;
+            rowTwo = (rowTwo + shift) % gridSize_;

         } else {
             // Rectangle/Square - so keep the rows the same and swap the columns
             // (NB the operation is actually the same regardless of encrypt/decrypt
             // since applying the same operation twice gets you back to where you were)
-            std::swap(pointOne.second, pointTwo.second);
+            std::swap(columnOne, columnTwo);
         }

         // Find the letters associated with the new coords and make the replacements
```

Specifying the C++ standard to use

- Since structure bindings are only available in C++17 onwards, we need to specify in the build that we want to use that standard when compiling

- To do this we have added `cxx_std_17` to the `target_compile_features` for the MPAGSCipher library

- Alternatively, if we wanted to set it for the entire project we could add:

  ```
  set(CMAKE_CXX_STANDARD 17)
  set(CMAKE_CXX_STANDARD_REQUIRED ON)
  ```

  in the top level CMakeLists.txt file, just before the line:

  ```
  set(CMAKE_CXX_EXTENSIONS OFF)
  ```