Example: Working with Different String APIs

Does string_view offer usage with APIs like CString and QString? Let's find out.

An interesting use case for `string_view` is when you use it in code that works with different string implementations.

For example, you might have `CString` from MFC, `const char*` from C-APIs, `QString` from QT, and of course `std::string`.

Rather than creating overloads for different string types, you might leverage `string_view`!

For example:

```
void Process(std::string_view sv) { }
```

If you want to use `Process` with different string implementations, then all you have to do is to create a string view from your type. Most of the string types should easily allow that.

For example:

```
// MFC Strings:
CString cstr;
Process(std::string_view{cstr.GetString(), cstr.GetLength()});

// QT Strings:
QString qstr;
Process(std::string_view{qstr.toLatin1().constData()};

// Your implementation:
MySuperString myStr;
// MySuperString::GetData() - returns char*
// MySuperString::Length() - returns length of a string
Process(std::string_view{myStr.GetData(), myStr.Length()});
```

Hypothetically, `Process()` could be implemented as `Process(const char*, int len)`, but with `string_view` the code is more explicit and simpler.

Additionally, you have all the available methods of `string_view`, and such code is more convenient than C-style.

For our final lesson, we'll try string splitting with `string_view`.