## The Basics

Let's begin our study on std::string\_view with a simple use case.

## WE'LL COVER THE FOLLOWING ^Introductory ExampleSolution

## Introductory Example #

Let's try a little experiment:

How many string copies are created in the below example?

```
// string function:
std::string StartFromWordStr(const std::string& strArg, const std::string& word) {
  return strArg.substr(strArg.find(word)); // substr creates a new string
}
int main() {
  // call:
  std::string str {"Hello Amazing Programming Environment" };
  auto subStr = StartFromWordStr(str, "Programming Environment");
  std::cout << subStr << '\n';
}</pre>
```

Can you count them all?

The answer is 3 or 5 depending on the compiler, but usually, it should be 3.

- The first one is for str.
- The second one is for the second argument in <a href="StartFromWordStr">StartFromWordStr</a> the argument is <a href="const">const</a> string& so since we pass <a href="const">const</a> char\* it will create a

new string.

- The third one comes from substr which returns a new string.
- Then we might also have another copy or two as the object is returned from the function. But usually, the compiler can optimize and elide the copies (especially since C++17 when Copy Elision became mandatory in that case).
- If the string is short, then there might be no heap allocation as Small String Optimisation.

Small String Optimisation is not defined in the C++ Standard, but it's a common optimisation across popular compilers. Currently, it's 15 characters in MSVC (VS 2017)/GCC (8.1) or 22 characters in Clang (6.0)

The above example is simplistic. However, you might imagine a production code where string manipulations happen very often. In that scenario, it's even hard to count all the temporaries that the compiler creates.

## Solution #

A much better pattern to solve the problem with temporary copies is to use std::string\_view. As the name suggests, instead of using the original string,
you'll only get a non-owning view of it. Most of the time it will be a pointer to
the contiguous character sequence and the length. You can pass it around and
use most of the conventional string operations.

Views work well with string operations like substring - substr. In a typical case, each substring operation creates another, smaller copy of the string.
With string\_view, substr will only map a different portion of the original buffer, without additional memory usage, or dynamic allocation.

Here's the updated version of our code that accepts string\_view:

```
std::string_view StartFromWord(std::string_view str, std::string_view word)
{
   return str.substr(str.find(word)); // substr creates only a new view
}
int main() {
   // call:
   std::string str {"Hello Amazing Programming Environment"};
```

```
std::cout << subView << '\n';
}</pre>
```







[]

In the above case, we have only one allocation - just for the main string - str.

None of the string\_view operations invokes copy or extra memory allocation for a new string. Of course, string\_view is copied - but since it's only a pointer and a length, it's much more efficient than the copy of the whole string.

**One warning**: while this example shows the optimisation capability of string views, please read on to see the risks and assumptions with that code! Or maybe you can spot a few now?

Ok, so when you should use string\_view. We'll find out next.