Choosing between string & string_view

We should be clear on why we choose a string_view instead of a string. Let's figure it out with the help of an example.

WE'LL COVER THE FOLLOWING

Since string_view is a potential replacement for const string& when passing in functions, we might consider a case of string member initialization. Is string_view the best candidate here?

For example:

```
class UserName {
  std::string mName;
  public:
  UserName(const std::string& str) : mName(str) { }
};
```

As you can see a constructor is taking `const std::string& str`.

You could potentially replace a constant reference with `string_view`:

```
UserName(std::string_view sv) : mName(sv) { }
```

Let’s compare those alternatives implementations in three cases: creating from a string literal, creating from an l-value and creating from an rvalue reference:

```
#include <iostream>
#include <string>
#include <string_view>

using namespace std;

class UserName
```

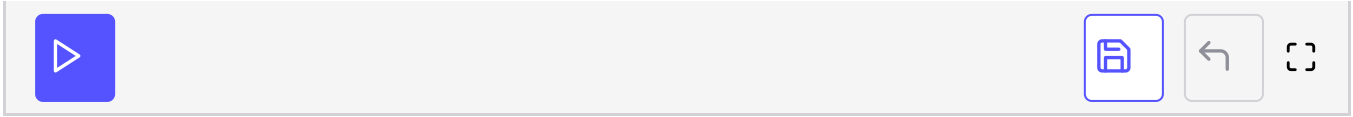
```
{
  std::string mName;
  public:
  UserName(std::string_view sv) : mName(sv) { }
  std::string_view getName(){return mName;}
};
std::string GetString() { return "some string..."; }

int main(){
  // creation from a string literal
  UserName u1{"John With Very Long Name"};
  cout << u1.getName() << endl;

  // creation from l-value:
  std::string s2 {"Marc With Very Long Name"};
  UserName u2 { s2 };
  cout << u2.getName() << endl;
  // use s2 later...

  // from r-value reference
  std::string s3 {"Marc With Very Long Name"};
  UserName u3 { std::move(s3) };
  cout << u3.getName() << endl;
  // third case is also similar to taking a return value:

  UserName u4 { GetString() };
  cout << u4.getName() << endl;
}
```



Now we can analyze two versions of `UserName` constructor - with a string reference or a string_view.

Please note that allocations/creation of `s2` and `s3` are not taken into account, we're only looking at what happens for the constructor call. For `s2` we can also assume it's used later in the code.

Observations

For const std::string&:

- `u1` - two allocations: the first one creates a `temp` string and binds it to the input parameter, and then there's a copy into `mName`.
- `u2` - one allocation: we have a no-cost binding to the reference, and then there's a copy into the member variable.
- `u3` - one allocation: we have a no-cost binding to the reference, and then there's a copy into the member variable.
- You'd have to write a `ctor` taking r-value reference to skip one allocation

for the `u1` case, and also that could skip one copy for the `u3` case (since we could move from r-value reference).

For std::string_view:

- `u1` - one allocation - no copy/allocation for the input parameter, there's only one allocation when `mName` is created.
- `u2` - one allocation - there's a cheap creation of a string_view for the argument, and then there's a copy into the member variable.
- `u3` - one allocation - there's a cheap creation of a string_view for the argument, and then there's a copy into the member variable.
- You'd also have to write a constructor taking r-value reference if you want to save one allocation in the `u3` case, as you could move from r-value reference.

While the `string_view` behaves better when you pass a string literal, it's no better when you use it with existing string, or you move from it.

In the next lesson, we'll examine how things change when we use string_view in our code.