Initializing string Members from string_view

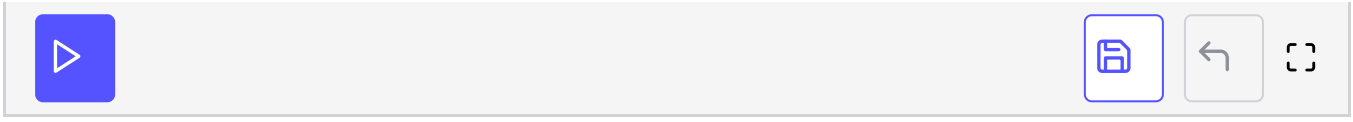
We'll continue our analysis of the example from the last chapter, this time using std::string_view instead of std::string.

WE'LL COVER THE FOLLOWING

Last time, we were left with this code:

```
#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;
}
```



Since the introduction of move semantics in C++11, it's usually better, and safer to pass `string` as a value and then move from it.

For example:

```
class UserName {
  std::string mName;

  public:
  UserName(std::string str) : mName(std::move(str)) { }
};
```

Now we have the following results:

For `std::string`:

- `u1` - one allocation - for the input argument and then one move into the `mName`. It's better than with `const std::string&` where we got two memory allocations in that case. And similar to the string_view approach.
- `u2` - one allocation - we have to copy the value into the argument, and then we can move from it.
- `u3` - no allocations, only two move operations - that's better than with `string_view` and `const string&`!

When you pass `std::string` by value not only is the code simpler, there's also no need to write separate overloads for rvalue references.

See the full code sample:

initializing_from_string_view.cpp

The approach with passing by value is consistent with item 41 - "Consider pass by value for copyable parameters that are cheap to move and always copied" from Effective Modern C++ by Scott Meyers.

However, is std::string cheap to move?

> Although the C++ Standard doesn't specify that, usually, strings are implemented with **Small String Optimisation** (**SSO**) - the string object contains extra space to fit characters without additional memory allocation[^ssonote]. That means that moving a string is the same as copying it. And since the string is short, the copy is also fast.

> *[^ssonote]:* SSO is not standardised and prone to change. Currently, it's 15 characters in MSVC (VS 2017)/GCC (8.1) or 22 characters in Clang (6.0). For multiplatform code, it's not a good idea to assume optimisations based on SSO.

Let's reconsider our example of passing by value when the `string` is short.

```
UserName u1{"John"}; // fits in SSO buffer

std::string s2 {"Marc"}; // fits in SSO buffer
UserName u2 { s2 };

std::string s3 {"Marc"}; // fits in SSO buffer
UserName u3 { std::move(s3) };
```

Remember that each move is the same as copy.

For `const std::string&`:

- `u1` - two copies: one copy from the input string literal into a temporary string argument, then another copy into the member variable.
- `u2` - one copy: the existing string is bound to the reference argument, and then we have one copy into the member variable.
- `u3` - one copy: the `rvalue` reference is bound to the input parameter at no cost, later we have a copy into the member field.

For `std::string_view`:

- `u1` - one copy: no copy for the input parameter, there's only one copy when `mName` is initialized.
- `u2` - one copy: no copy for the input parameter, as `string_view` creation

is fast, and then one copy into the member variable.

- `u3` - one copy: `string_view` is cheaply created, there's one copy from the argument into `mName`.
- Extra risk that `string_view` might point to a deleted string.

For `std::string`:

- u1 - two copies: the input argument is created from a string literal, and then there's copy into `mName`.
- u2 - two copies: one copy into the argument and then there's the second copy into the member.
- u3 - two copies: one copy into the argument (move means copy) and then there's the second copy into the member.

As you see for short strings passing by value might be "slower" when you pass some existing string, simply because you have two copies rather than one.

On the other hand, the compiler might optimise the code better when it sees an object and not reference. What's more, short strings are cheap to copy, so the potential "slowdown" might not be even visible.

All in all, passing by value and then moving from a string argument is the preferred solution. You have a simple code and better performance for larger strings.

As always, if your code needs maximum performance, then you have to measure all of the possible cases.

Other Types & Automation

> The problem discussed in this section can also be extended to other copyable and movable types. If the move operation is cheap, then passing by value might be better than by reference. You can also use automation, like Clang-Tidy, which can detect potential improvements. Clang Tidy has a separate rule for that use case, see clang-tidy - modernize-pass-by-value.

Here's the summary of string passing and initialisation of a string member:

input parameter	`const string&`	`string_view`	`string` and `move`
`const char*`	2 allocations	**1 allocation**	1 allocation + move
`const char*` SSO	2 copies	**1 copy**	2 copies
lvalue	**1 allocation**	**1 allocation**	1 allocation + 1 move
lvalue SSO	**1 copy**	**1 copy**	2 copies
rvalue	1 allocation	1 allocation	**2 moves**
rvalue SSO	**1 copy**	**1 copy**	2 copies

Next, we shall examine the behaviour of `string_view` with non-null terminated strings.