Reference Lifetime Extension

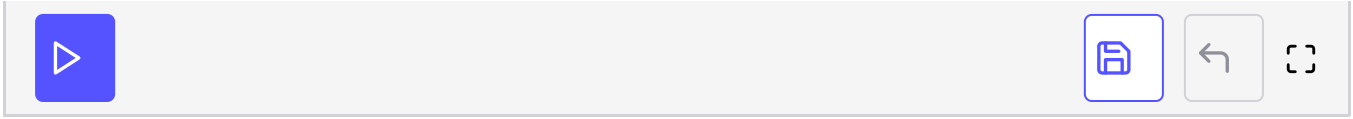
This lesson deals with the lifetime of temporary objects.

What happens in the following case:

```
#include <iostream>
#include <vector>
using namespace std;

std::vector<int> GenerateVec() {
  return std::vector<int>(5, 1);
}

int main() {
  const std::vector<int>& refv = GenerateVec();
  cout << refv.size();
}
```



Is the above code safe?

Yes - the C++ rules say that the lifetime of a temporary object bound to a `const` reference is prolonged to the lifetime of the reference itself.

> Here's a full example quoted from the standard (Draft C++17 - N4687) 15.2 Temporary objects [class.temporary]:

> Example:

```
struct S {
 S();
 S(int);
 friend S operator+(const S&, const S&);
 ~S();
};
```

```
S obj1;


const S& cr = S(16)+S(23);
S obj2;
```

> The expression `S(16) + S(23)` creates three temporaries: a first temporary `T1` to hold the result of the expression `S(16)`, a second temporary `T2` to hold the result of the expression `S(23)`, and a third temporary `T3` to hold the result of the addition of these two expressions. The temporary `T3` is then bound to the reference `cr`. It is unspecified whether `T1` or `T2` is created first. On an implementation where `T1` is created before `T2`, `T2` shall be destroyed before `T1`. The temporaries `T1` and `T2` are bound to the reference parameters of `operator+`; these temporaries are destroyed at the end of the full expression containing the call to `operator+`. The temporary `T3` bound to the reference `cr` is destroyed at the end of `cr`'s lifetime, that is, at the end of the program. In addition, the order in which `T3` is destroyed takes into account the destruction order of other objects with static storage duration. That is, because `obj1` is constructed before `T3`, and `T3` is constructed before `obj2`, `obj2` shall be destroyed before `T3`, and `T3` shall be destroyed before `obj1`.

While it's better not to write such code for all of your variables, it might be a handy feature in cases like:

```
for (auto &elem : GenerateVec()) {
  // ...
}
```

In the above example, `GenerateVec` is bound to a reference (`rvalue` reference for the start of the vector) inside the range-based for loop. Without the extended lifetime support, the code would break.

How does it relate to `string_view`?

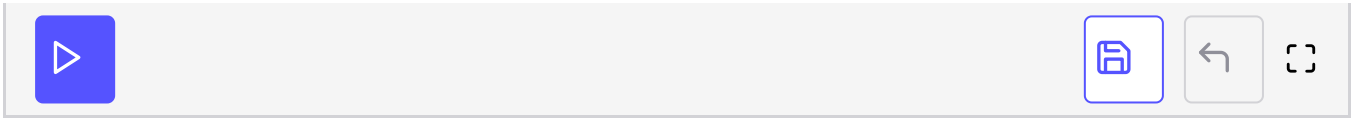
For `string_view` the below code is usually error-prone:

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

```
using namespace std;

std::string func()
{
  std::string s; // build s... return s;
}


int main() {
  std::string_view sv = func();
  // no temp lifetime extension!
}
```



This might be not obvious - `string_view` is also a constant view, so should behave almost like a `const` reference. But according to existing C++ rules, it's not- the compiler immediately destroys the temporary object after the whole expression is done. The lifetime cannot be extended in this case.

`string_view` is just a proxy object, similar to another code:

```
std::vector<int> CreateVector() { ... }
std::string GetString() { return "Hello"; }

auto &x = CreateVector()[10]; // arbitrary element!
auto pStr = GetString().c_str();
```

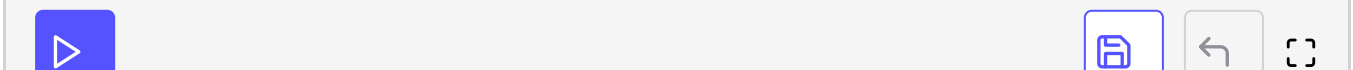
In both cases `x` and `pStr` won't extend the lifetime of the temporary object created in `CreateVector()` or `GetString()`.

You might fix it by:

```
#include <iostream>
using namespace std;

std::string func()
{
  std::string s; // build s... return s;
  return s;
}

int main() {
  auto temp = func();
  std::string_view sv { temp };
  // fine lifetime of temporary is extended through `temp`
}
```



Every time you assign a return value from some function you have to be sure the lifetime of the object is correct.

> There's a proposal to fix the issues with string_views and other types that should have extended reference lifetime semantics: see P0936.

The next lesson talks about choosing between string and string_view.