Matt Warner

1 Temporaries in for loops

Are we allowed to use a temporary in a range based for loop? Consider the following code:

```
std::vector<int> create_range() {
    return {1,2,3,4,5};
}
int main() {
    for (auto const& value : create_range()) {
        std::cout << value << ' ';
    }
}
```

Is the temporary object returned by `create_range()` kept alive during the for loop?

The answer is yes, and the following code prints:

```
1 2 3 4 5
```

But if we make anything more on the temporary, even something as simple as returning a reference to it:

```
std::vector<int> create_range() {
    return {1,2,3,4,5};
}
// Note: Const lvalues can be given rvalues
const std::vector<int>& f(const std::vector<int> &v) {
    return v;
}
int main() {
    for (auto const& value : f(create_range()))A {
        std::cout << value << ' ';
    }
}
```

Then the code falls into undefined behaviour. On a certain implementation, the output is this:

```
0 0 3 4 5
```

Temporaries are usually destroyed on the end of a statement, so how we transform them on the line of code should not influence the moment they're destroyed.

1.1 The code of a range based for loop

When we write a range based for loop, the compiler expands it into several lines of less nice looking code.

For example, the following loop:

```
for (const auto& value : myRange) {
    // code using value
}
```

Gets expanded into this:

```
{
    auto&& range = myRange;
    auto begin = begin(range);
    auto end = end(range);
    for (; begin != end; ++begin) {
        auto const& value = *begin;
        // code using value
    }

}
```

1.2 Using temporary objects

Lets go back to our initial example using temporaries:

```
1  std::vector<int> create_range() {
2      return {1,2,3,4,5};
3  }
4  int main () {
5      for (const auto& value : create_range()) {
6          std::cout << value << ' ';
7      }
8  }
```

Here is what the expanded for loop looks like in this case:

```
1  {
2      auto&& range = create_range();
3      auto begin = begin(range);
4      auto end = end(range);
5      for (; begin != end; ++begin) {
6          const auto& value = *begin;
7          // code using value
8      }
9  }
```

As we can see, the temporary is not created on the line of the `for`, unlike what the syntax of the ranged based for loop could have been suggesting.

How can the above code work? What prevents the temporary from being destroyed at the end of the statement it is created on, on line 2 in the above code?

This is one of the properties of `auto&&`. Like `const&`, a reference declared with `auto&&` keeps a temporary object alive until that reference itself gets out of scope. This is why the temporary object returned by `create_range()` is still alive and valid when reaching the statements using its values inside of the for loop.

1.3 Transformations of temporary objects

New let's go back to the initial example that was undefined behaviour:

```
std::vector<int> create_range() {
    return {1,2,3,4,5};
}
const std::vector<int>& f(const std::vector<int>& v) {
    return v;
}
int main() {
    for (const auto& value : f(create_range())) {
        std::cout << value << ' ';
    }
}
```

Lets expand the loop again:

```
auto&& range = f(create_range());
auto begin = begin(range);
auto end = end(range);
for (; begin != end; ++begin) {
    const auto& value = *begin;
    // code using value
}
```

Can you see what's wrong with this code now?

Unlike in the previous case, `auto&&` doesn't bind on the expression `create_range()`. It binds on the reference to that object returned by `f`. And that is not enough to keep the temporary object alive.

It is interesting to note that range is declared with an `auto&&` binding to a `const&` which is defined (in the implementation of `f`) to be equal to a `const&` on the temporary. So we have a chain of `auto&&` and `const&` which, individually, can keep a temporary alive. But if we don't have a simple expression with one of them biding directly on the temporary, they do not keep it alive.

1.4 How to fix the code

If you have to use `f` to make a transformation on your temporary, then you can store the result of this transformation in a seperate object, defined on a seperate line:

```
auto transformed_range = f(create_range());
for (auto const& value : transformed_range) {
    std::cout << value << ' ';
}
```

This is less nice because it adds code without adding meaning, and it generates a copy of the transformed range. But in the case of a transformation, `f` can return by value, which can enable **return value optimisations** or move semantics if the type is moveable. But still, the code gets less concise.