Range-based for loop

Dienstag, 14. Dezember 2021 08:41

Links with other topics:
CppCon | The C++ Conference
GitHub - CppCon/CppCon2020: Slides and other materials from CppCon 2020

There is an interesting talk held by **Vittorio Romeo**, where he summarized the acceptance of C++ 11/14 in engineer community, and his personal experience teaching the C++.
https://www.youtube.com/watch?v=7Mfbpbyq6fs&list=PLHTh1InhhwT6vjwMy3RG5Tnahw0G9qIx6

He also talked about the categorization of the features in terms of the security impact on the code, and the pitfalls that are not so obvious (required additional training - deeper understanding in order to be used correctly)

- **Safe** (*override, static_assert, default, …*)
- **Conditionally safe** (*auto, range-based for loop, …*)
- **Unsafe** (*external template, …*)

Regarding the conditional safe feature, **range-based loop**, which is widely used in code base.
The range-based for loop has form:

```
for (const auto& value : values)
{
    // body
}
```

is internally implemented as **in-place (scope) macro expansion**

```
{// begin of scope

   auto&& rv = values;

   auto first = std::begin(values);

   auto last = std::end(values);

   for (; first != last; ++first) // internal for loop

  {

      const auto& value = *first;

      // body of the for loop

      ...

   }

}// end of scope
```

`@note` C++ Insights (cppinsights.io) is very helpful tool to see how your code is internally represented

Problem
Let's assume we have some class A:

```
class A
{
  public:

        using values_type = std::vector<int>;

        A(std::initializer_list<int> list) noexcept : m_values{list}{}
```

```
        // const values_type& getValues() const { return m_values; } //[1] - Undefined Behavior
        values_type getValues() const { return m_values; } //[2] OK

  private:

        values_type m_values;

};
```

And now we write something like

```
for (const auto& value : A({1, 2, 3}).getValues())
{

    // body

}
```

will result in **undefined behavior**
Example: https://godbolt.org/z/Er5rxqzP5

Explanation:
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2012r0.pdf

If we go back to the way how the range-based for loop is implemented

```
auto&& rv = <range expression>;
```

or, in our case [1]

```
const values_type& getValues() const { return m_values; } //[1] UB
auto&& rv = A().getValues();
```

We have a universal reference on **lvalue reference on a member of a temporary object A**.
Returning the **reference** on a temporary object A::m_value doesn't apply the rule of temporary lifetime extension– rv dangling, refers on the A::m_value which ceased to exist (in memory): undefined behavior

But, if we return the vector by the value [2], we have a universal reference **on a temporary object** - A::**values_type**

```
values_type getValues() const { return m_values; } //[2] OK
auto&& rv = A().getValues();
```

Here we can apply the rule: the reference bound to the temporary object, will extend the lifetime of the temporary object – so that they have the same life time: until the reference goes out of scope (in this case, range-based for loop scope).

Be aware of this, and if necessary, do the initialization before the loop.
In **C++20** you can also do this inside the loop:

```
for (auto a = A({1,2,3}); const auto& val : a.getValues())
{
  //body
}
```

In addition to that, there was interesting *C++ now* talk held by **Arno Schödl** on pitfalls related with rvalues.
For me, the most interesting part was on temporary lifetime extension, and the negative connotations of that, especially in context of generic functions that internally use rvalue as a const lvalue reference and return the same using decltype deduction rules, although it's actually a value that is going to "fade".

https://www.youtube.com/watch?v=sb7cj-3l1Kc&list=PL_AKIMJc4roXvFWuYzTL7Xe7j4qukOXPq

The code example, that illustrates the usages:
https://godbolt.org/z/5sqnPxfcz