Document: P2050R0
Date: 2020-01-13
Audience: SG7
Authors: Andrew Sutton
(asutton@lock3software.com)
Wyatt Childers
(wchilders@lock3software.com)

Tweaks to the design of source code fragments

Introduction

This paper refines features presented in our previous paper P1717 (Compile-time Metaprogramming in C++). Specifically, we are rethinking two aspects of source code fragments, which form the "values" used for source code injection. These are

- the explicit use of an unquote operator for references to local variables
- changing the type of fragments from unique, anonymous class types to `meta::info`.

These two changes make source fragments easier to use, especially with standard containers.

The following table shows the changes proposed in this paper.

P1717	**P2050**

`constexpr {`
`  for (int i = 0; i < 10; ++i) {`
`  -> __fragment struct X {`
`  int unqualid("val_", i);`
`  };`
`  }`
`}`
	`constexpr {`
	`  for (int i = 0; i < 10; ++i) {`
	`  -> fragment struct X {`
	`  int unqualid("val_", %{i});`
	`  };`
	`  }`
	`}`

`<not expressible>`	`vector<meta::info> frags;`
	`for (int i = 0; i < 10; ++i)`
	`  frags.push_back(fragment {`
	`  case %{i}: return %{i} + 10;`
	`  });`

Background

A source code fragment is a piece of code that is intended to be injected. We currently define these as expressions—a kind of source code literal. For example, we can create a fragment of a class like so:

```
constexpr auto frag == fragment class C {
  int x;
  void f() { }
};
```

The source code fragment is the expression beginning with `__fragment` and ending at the closing brace of the class definition. It defines a set of members that can be injected into a class context later:

```
struct X {
  constexpr {
    -> frag;
  }
};
```

The injection statement `-> frag` causes the members of the fragment above to be inserted into the body of class `X`.

There are multiple kinds of fragments. We can declare parts of classes (as above), namespaces, enumerations, parameter lists, and functions (i.e., blocks). Each fragment has its own associated syntax and limitations (e.g., an enumeration fragment cannot be injected into class body).

More complex (and common) uses of fragments require additional require the “parameterization” of fragments with additional values. As a trivial example, consider the ability to create a sequence of member variables:

```
struct X {
  consteval {
    for(int num = 0; num < 10; ++num)
      -> fragment struct {
        int unqualid(“value_” num);
      };
  }
};
```

Within the body of the fragment (in the `unqualid` operator), the identifier `num` designates a placeholder for a value to be supplied during constant expression evaluation. That identifier is “bound” to the variable declared in the `for` loop. When the metaprogram executes, the current value of num is associated with the placeholder in order to yield the following sequence of injected data members.

```
int unqualid(“value_” 0);
int unqualid(“value_” 1);
// …
int unqualid(“value_” 9);
```

The end result is a `struct X` with 10 members named `value_<num>`.

The type of fragment expressions is a unique class type, similar to closure types of lambda expressions. The type is comprised of two parts: the reflection of contents of the fragment (a reflection of the unnamed struct in the example above) and the value corresponding to each placeholder in the definition of the fragment.

Problems

We think there are two significant problems with our current design of fragments:

- The use of implicit placeholders corresponding to local variables (which we have sometimes referred to as “captures”) has proven difficult to explain and frequently leads to confusion, especially since the placeholder and the variable to be substituted are different entities.
- Because each fragment has a distinct class type, we cannot create arrays or vectors of these objects. This seems like a useful feature for metaprograms that compose (or filter) lists of fragments to be injected.

Our redesign of fragments addresses these two issues.

The unquote operator

If you squint, you might notice that `fragment` is actually a kind of “quote” that you find in many languages providing advanced metaprogramming facilities. The content of a quote is typically an unstructured or semi-structured part of a program, e.g., text, tokens, untyped ASTs—it depends on the language. In our design, the operand is a fully analyzed class definition, namespace definition, block, etc.

Languages that provide more obvious quote operators often also provide unquote or escape operators, that allow a programmer to insert computed types or values into the quoted source code. This is not fundamentally different than languages that provide support for string interpolation. Our previous design allowed the same through the use of local variables within fragment bodies. As noted, the implicit nature of the feature easily leads to confusing source code.

In our new design, we have replaced our previous approach of implicitly capturing local variables with a more explicit unquote operator, `%{}`. For example:

```
constexpr {
  auto type = reflexpr(int);
  for (int num = 0; num < 5; ++num) {
    -> fragment struct F {
      typename(%{type}) unqualid("value", %{num});
    };
  }
}
```

The semantics of the unquote operator are not fundamentally changed from our previous design. Instead of implicitly referring to local variables, we now require local values to be explicitly unquoted. The primary benefit of this approach is that it is obvious to readers which values are being added to the injection. It should eliminate some of the user confusion we saw in our earlier experiments.

For now, we limit the syntax inside the quote operator to *id-expressions*. We could relax this restriction and allow for a limited subset of computation within unquote. For example:

```
typename(%{get_type(type)}) unqualid("value", %{num * 2});
```

However, this adds some implementation complexity that we haven't thoroughly explored yet. For now, we restrict the feature to only *id-expressions*.

Fragment values

In Cologne we received feedback, suggesting that every fragment having its own anonymous type was undesirable. Instead, we would like to propose that the type of a fragment expression is instead a reflection i.e. `meta::info`. Among other things, this would allow users to collect fragments in standard containers. For example:

```
consteval std::vector<meta::info> make_members(int n) {
  std::vector<meta::info> fragments;
  for (int i = 0; i < n; ++i) {
    meta::info f = __fragment struct {
      int unqualid(“member_”, %{i});
    };
    fragments.push_back(f);
  }
  return fragments;
}
```

As noted, the type of fragments is `meta::info`. However, the compiler still needs to maintain information about the placeholders required by unquoted terms in the definition. Our approach simply

internalizes this information, making `meta::info` values for fragments handles to data that was previously exposed as a value in its own right.

Conclusions

The changes proposed in this paper address two issues related to the usability of source code fragments. The unquoting operator makes it more obvious that local values are being inserted into injected code. The reformulation of fragment values allows them to be used with more common programming techniques, including the use of standard containers.