P3412R1+

String interpolation

Bengt Gustafsson

Telecon – Feb 26, 2025

At a glance

```
int x = 42;

std::string a = f”Value: {x}”;
std::print(x”Value: {x + 3}”);   // Any expression
std::cout << f”Value: {x:>3}”;   // Format specifier ok
std::println(x”Last error: {errno}”);  // Macros ok
```

What happens during lexing (f)

```
int x = 42;

std::string a = f”Value: {x}”;
```

Becomes

```
std::string a = __FORMAT__(”Value: {}”, (x));
```

What happens during lexing (x)

```
int x = 42;

std::print(x”Value: {x + 3}”);
```

Becomes

```
std::print(”Value: {}”, (x + 3));
```

The steps in the compilation process

- The lexer extracts the expressions into $f and $x operators in phase 3.
- Macros are expanded as usual in phase 4.
- Concatenation happens as usual in phase 6, but with handling of $f and $x operators, moving all expressions last.
- Replace $f with __FORMAT__ and remove $x
- Double braces when concatenating string-literals to $x and $f operators.

Translation example

```
#define B b
f”Value: {a}” ” {} ” fR”(\t {B + c})”
After phase 3:
$f(”Value: {}”, (a)) ” {} ” $f(”\t {}”, (B + c))
After phase 4:
$f(”Value: {}”, (a)) ” {} ” $f(”\t {}”, (b + c))
After phase 6:
__FORMAT__(”Value: {} {{}} \t {}”, (a), (b + c))
```

The crux: Finding the end of expression fields

- An expression field is an *expression* in the C++ grammar.
- An expression field ends with : or }
- Both these tokens can occur in expressions.

- To find which } is the end count matched { … } pairs.
- To find which : is the end count matched ? … : pairs.
- Special handling for :: - not ending expression field if followed by an identifier, `operator` or `*`.
- Special handling for :> digraph requires counting […] pairs and assuming the :> is] if unmatched [exist.

Working on the token or character levels

- R2: Macro expansion is done after extracting the expressions => no unbalanced braces in macros allowed.
- This allows simpler tools to find expression ends.
- Recursing within phase 3 handles comments, newlines and nested string literals.
- This was trivial to implement in the Clang preprocessor.

Conflicting goals.

- We want f-literals to be usable whenever std::string is.
- We don’t want to lose the performance gained with `std::print`, compared to `cout << std::format(…)`.
- We didn’t want to resort to both f-literals and x-literals.

In R0 his led to a rather complicated system with a formatted_string class relying on P3298 and P3398.

In R1 we resort to having both f-literals and x-literals, resulting in significant simplification.

Alternatives to f and x for R2

- Use \{ in any string literal to mean it is a f or x literal, use s suffix to get std::format wrapping. *Jonathan Müller.*
- Use f”” when R1 uses x”” and use s suffix to get std::format wrapping. *Barry Revzin*
- Use the new `quotes` to indicate an extraction literal, use s suffix to get std::format wrapping. *Bengt.*

Note: All these require augmented core language to handle an `operator ””s` that can take any argument list.

Unclear if this can cause conflicts if you format one size_t.

Implementation experience

- A reasonably complete Clang implementation is on godbolt. Look for the x86_64 clang version marked P3412 String interpolation. This implements P3412R0, so always use f””!
- A free-standing pre-preprocessor also exists, showing how syntax highlighting can be implemented in tools like editors that don’t do full lexing.

Problem with x-literal misuse.

```
auto v = x”Value {1}, {2}”;
```

Becomes:

```
auto v = ”Value {}, {}”, (1), (2);
```

Same as:

```
int v = 2;
```

But compilers can easily emit a warning.

The Pythonic debug feature.

- This proposal has the debug feature of python f-literals.
- An expression that ends in = is its own label.
- An expression that ends in operator= fails to compile.*

```
int y = 42;
std::print(x”{y=}, {y*2=}”);
```

Output:

```
y=42, y*2=84
```

* It already fails in std::format as a member function pointer can’t be formatted!