String interpolation

Document #:	P3412R2
Date:	2025-05-18
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Bengt Gustafsson <bengt.gustafsson@beamways.com> Victor Zverovich <victor.zverovich@gmail.com>

Contents

1 Revision history

1.1 R0, Presented to EWGI in Wroclaw

First revision.

1.2 R1, Presented to EWGI in Hagenberg

— Remove the basic_formatted_string struct that R0 used to be able to unify f- and x- literals into just f-literals. This avoids relying on the P3298 and P3398 proposals.

— Remove the print overloads added by R0, instead let programmers use x-literals when printing while f-literals produce a std::string or std::wstring directly.

— Change f-literals to generate a function call to `__FORMAT__` instead of `std::make_formatted_string` or `std::format` to allow uses which do not rely on the formatting functionality of the standard library. Also a small discussion about other possible names.

1.3 R2, This revision

— Add a *non-ignorable* attribute indicating which parameter of a function is a format string to avoid the need for x-literals.
— Rename the `__FORMAT__` function to`__format__` to make it clearer that it is not a macro.

— Change macro expansion to happen after expression extraction. This allows separation of the different steps in the f-literal handling into the phases of translation, and simultaneously makes applications such as syntax coloring editors more robust.
— Definitively exclude user defined suffixes from working together with f-literals.
— Add a chapter about printf style formatting.
— Added some text about possible wording strategies for the preprocessor phases.

2 Abstract

This proposal adds string interpolation (so called f-literals) to the C++ language. Each f-literal is transformed by the preprocessor to a token sequence constituting a call to a function tentatively named `__format__` with the expression fields extracted into a function argument list suitable for consumption by `std::format`. If `<format>` is included or if the `std` module is imported a `__format__` function which forwards to `std::format` is defined, effectively translating the f-literal to a `std::string` or `std::wstring` depending on the format string's encoding.

In addition this proposal adds an attribute which can be attached to a function declaration, for instance of `std::print` to indicate that a `__format__` function call emitted by the preprocessor is to be removed and replaced by its arguments. A variation of the attribute suitable for printf-like functions is also included, which together with the preprocessor changes could potentially be standardized for C too.

This proposal (currently the R0 revision) has been implemented in a Clang fork, which is available on Compiler Explorer. It has also been implemented as a separate program, which demonstrates the viability of this proposal for tools like syntax coloring editors.

3 Examples

Assuming a Point class which has a formatter we can now use string interpolation to format Points. This is in contrast with R0, where P3298 and P3398 were required to make the below examples work as expected and R1 where the f-literals in print had to be spelled as x-literals. Note however that the combination of f-literal and `cout` is not optimal, and just as in C++23 using std::print is preferable if performance is important.

```
Point getCenter();

std::string a = f"Center is: {getCenter()}";      // No dangling risk.

auto b = f"Center is: {getCenter()}";             // b is std::string as std::format is called

size_t len = f"Center is: {getCenter()}".size(); // Works as the f-literal is a std::string.

std::println(f"Center is: {getCenter()}");        // Works as println has an attribute

std::cout << f"Center is: {getCenter()}";         // Sub-optimal as a temporary std::string is created.
```

4 History

This proposal was initiated by Hadriel Kaplan in October of 2023. Unfortunately Hadriel Kaplan never submitted his proposal officially and after some discussions and setting up an issue tracker for the proposal Hadriel Kaplan has not been possible to contact via e-mail and stopped posting on the issues in this tracker or refining his proposal.

The proposal presented here uses the same basic idea of letting the preprocessor extract the expressions out of the format string and place them as an argument list after the remaining literal. In R2 a novel attribute based approach is used to avoid having to separate f- and x-literals.

Some parts of this proposal was taken from Hadriel Kaplan's original draft, in some instances with modifications.

Before this there was a proposal [P1819R0] which used another approach applied after preprocessing.

5 Motivation

Before this proposal:

```
int calculate(int);

std::string stringify(std::string_view prefix, int bits) {
    return std::format("{}:{}: got {} for {:#06x}", prefix, errno, calculate(bits), bits);
}

void display(std::string_view prefix, int bits) {
    std::print("{}:{}: got {} for {:#06x}", prefix, errno, calculate(bits), bits);
}
```

After this proposal:

```
int calculate(int);

std::string stringify(std::string_view prefix, int bits) {
    return f"{prefix}-{errno}: got {calculate(bits)} for {bits:#06x}";
}

void display(std::string_view prefix, int bits) {
    std::print(f"{prefix}-{errno}: got {calculate(bits)} for {bits:#06x}");
}
```

C++ f-literals are based on the same idea as python f-strings. They are wildly popular in modern python; maybe even more popular than the python `str.format()` function that the C++ `std::format()` was based on.

Many other programming languages also offer string interpolation, and some use identical syntax, although there are other spellings, some based on a $ or % prefix. (full list). As std::format already uses the Python syntax with {} it seems logical to continue on this path as there is no consensus among languages anyway.

The main benefit of f-literals is that it is far easier to see the argument usage locations, and that it is less verbose. For example in the code snippets above, in the second example it is easier to see that "`prefix`" goes before `errno`, and "`bits`" is displayed in hex. Here `errno` is used as an example of a macro that is often not known to be a macro. It would be surprising if `errno` and other macros were not allowed in f-literals, which is motivation for implementing string interpolation in the preprocessor.

IDEs and syntax highlighters can support f-literals as well, displaying the embedded expressions in a different color:

```
    f"{prefix}-{errno}: got {calculate(bits)} for {bits:#06x}"
```

6 Terminology

The different parts of an f-literal have specific names to avoid confusion. This is best illustrated by an example, see below.

```
       f"The result is { get_result() :{width}.3}"
//     ^~~~~~~~~~ f-string-literal ~~~~~~~~~~~~~^
//                    or f-literal

       f"The result is { get_result() :{width}.3}"
//                     ^~~~~~~~~~~~~~~~~~~~~~~~^
```

```
//                                  |
//                            extraction-field

       f"The result is { get_result() :{width}.3}"
//                       ^~~~~~~~~~~~^ ^~~~~~~^
//                                   |         |
//                 expression-field    format-specifier

       f"The result is { get_result() :{width}.3}"
//                                       ^~~~^
//                                         |
//                                nested-expression-field
```

When the f-literal is passed along to the rest of the compiler a regular string literal token is formed, not containing the characters of the expression-fields. Such a string-literal token is called a *remaining-literal*.

7 Expression-field contents

The contents of an expression-field is a full *expression*. The grammar for *expression* includes the comma operator so when the expression is extracted by the preprocessor and placed after the literal each extracted expression is enclosed in a parenthesis. This prevents an extracted expression from being interpreted as multiple arguments to the `__format__` function call that the f-literal results in. Allowing a full *expression* instead of only an *assignment-expression* as in a regular function argument is needed to avoid causing errors due to commas in template argument lists, which can't be easily differentiated from other commas by the preprocessor.

This is illustrated by the following examples:

```
f"Weird, but OK: {1 < 2, 2 > 1}"

// Transformed to:
__format__("Weird, but OK: {}", (1 < 2, 2 > 1))

int values[] = {3, 7, 1, 19, 2 };
f"Reversed: {std::set<int, std::greater<>>(values, values + 5)}"

// Transformed to:
__format__("Reversed: {}", (std::set<int, std::greater<>>(values, values + 5))
```

The main complication of allowing an *expression* (or *assignment-expression*) in an expression-field is that an expression can contain a colon, while a colon is also used to end an expression-field if there is a format-specifier.

Less problematic consequence of referring to the grammar for *expression* is that thereby nested string literals and comments using both `//` and `/* */` are allowed. Newlines are also allowed in expression-fields even if the surrounding literal is not raw. An *expression* may contain lambdas which means that there may occur other types of colons including labels and base class list introducers in lambda bodies.

It seems complicated on the standardization level to define a new *almost-expression* which has some more or less arbitrary rules limiting its contents, and it definitely increases the cognitive load on programmers to have to remember those rules. If the rules would involve escaping quotes of nested string literals with backslashes the readability is also hampered. Allowing full expressions also significantly simplifies the task if tools like `clang-tidy` would get fixup functions to change `std::format` calls to f-literals: Whatever is in the argument expressions is allowed inside the expression-field and can be copied in character by character, even including newlines and comments.

For comparison, Python has supported string interpolation for many years but in 2022 the definition of expression fields was changed to a full Python expression, including nested string literals with the same quote kind as the

containing f-literal (Python allows enclosing strings in either single our double quotes, and previously nested string literals had to use the opposite quote kind compared to the enclosing f-literal). This change was proposed in PEP-701 which was incorporated into Python 3.12.

7.1 Detecting where the expression ends

Detecting the end of an *expression* is easy if done while actually parsing. But calling the parser while lexing a string-literal token could be problematic, and other tools such as syntax coloring editors don't contain a full parser, so a full parser can't be relied on.

However, it turns out that it is not very hard to implement a partial parser inside the lexer just to determine where an *expression* ends, assuming that it is possible to restart lexing from the character following the `{` that starts the expression-field. Restarting the lexing implies that nested comments, newlines, multi-character tokens etc. is handled by the normal lexing code.

Due to the fact that an expression-field must be followed by either a `}`or a `:` there are a number of rules to follow. Firstly we have to skip over nested curly brace pairs to see which `}` is the first one after the expression, and secondly we must apply some rules to be able to discern if a colon starts a *format-specifier* or not when outside any nested curly brace pair:

- Scope resolution operators. These are double colons followed by an identifier or the `operator` keyword. If there is something else after a colon-colon token the first colon must be the start of a *format-specifier* and the second a colon used as the fill character. The standard format-specifier's fill character syntax requires a `<`, `^` or `>` character after the fill character, none of which can't start an identifier. While it is possible that a user-defined formatter allows a leading colon followed by one or more letters this seems unlikely. If such a formatter exists its users will have to change the format specifier syntax to for instance allow a space between the colons in order to be able to use f-literals for their formatting. As colon as fill character is extremely rare this rule can be simplified to: *double colons can never start a format specifier in a f-literal.*
- The colon of ternary operators must not be mistaken for the start of a *format-specifier*. This can be handled by just counting the number of `?` tokens and ignoring as many colons. An alternative, used in the Clang implementation, is to recurse to the *expression-field* handler for each `?` encountered., basically following the C++ grammar.
- The digraph `:>` could be handled either by not supporting digraphs, in which case it would immediately be lexed as a colon followed by a `>` which thus means a `format-specifier` starting by a right-alignment specification. As `std::format` does not support using the digraphs `<%` and `%>` to enclose *extraction-fields* instead of braces we may assume that anyone able to type a f-literal can also type a `]` and does not have to resort to the problematic `:>` digraph. An alternative, which is used in the Clang implementation, is to do a special test if an unmatched `]` token is encountered: If it was formed from the digraph sequence break it up into the separate `:` and `>` to form the expected start of the *format-specifier*. This allows using :> as a `]` substitute as long as it is balanced within the *expression-field*.

The current implementations both use a somewhat more complex parser where nested parenthesis and square bracket pairs are also skipped over. This improves error handling by detecting mismatched parentheses in expression-fields immediately and aids in the handling of :> digraphs. With the currently proposed syntax for reflection splicing using `[:` and `:]` ignoring colons inside matched square brackets becomes necessary.

Further into the future, if more uses of colons inside expressions are specified, the implementation of f-literal lexing may have to be updated, and such new uses of colons would have to be denied if it would mean that it is impossible to detect the end of an expression-field. Thus specifying a full *expression* as allowed in expression-fields is future proof. That is to say, the rules above are strictly not needed to be stated explicitly in the standard, it is enough to refer to the grammar for *expression* and the rules follow from this, including any future modifications, and leaving it to implementations to figure out how to find the expression field end. With both this proposal and the two reference implementations as guidance this should not be hard to do. To aid implementers wording detailing the rules above could also be added to the description of the preprocessor, but leaving this out would make it easier to synchronize the preprocessor specification between C and C++.

One problem with a language agnostic preprocessor is that C doesn't have the scope resolution operator, so

in principle `::x` for some identifier x can start a format specifier, although no valid printf format specifier can start this way, so the difference between using a preprocessor that understands C++ or not is just which error message you get.

7.2 Preprocessor directives in expression-fields

Preprocessor directives inside expression-fields is not allowed. It does not make much sense to allow preprocessor directives inside an expression in the first place and it may make much harm if for instance an #else is placed inside an expression inside a f-literal. Regardless of if the #if condition is true or false an unterminated string literal would result. As allowing preprocessor directives is under the control of the preprocessor implementing f-literals this limitation should be trivial to enforce.

It could be argued that some preprocessor directives or combinations should be allowed in expression fields such as #pragma and a complete #if to #endif combination. If there turns out to be a good use case for this the restriction on preprocessor directives could be relaxed by a later proposal.

7.3 Error handling

To handle errors inside the expression fields in a good way is somewhat challenging considering that a quote that appears inside an expression-field is the start of a nested string literal while the programmer could have missed the closing brace of an extraction-field with the intent that the quote should end the f-literal. In the simplest case this causes the *nested* string literal to be unterminated, but in cases with more string literals on the same line it may cause the inside/outside of string literals to be inverted.

```
// Here the human reader quickly detects the missing } after x, but the lexer
// will find an unterminated string literal containing a semicolon after the meters
// "identifier".
auto s = "Length: {x meters ";
```

In the Clang implementation a simple recovery mechanism is implemented by re-scanning the f-literal as a regular literal after reporting the error. This avoids follow-up errors as long as there are no string literals in the expression-fields of the f-literal. In more complex cases, just as if you miss a closing quote today, various follow up errors can be expected, especially if there are more quoted strings on the same line.

7.4 Implementation in other tools

Embedding full expressions into string literals means that both that preprocessors and tools like static analyzers and editors doing syntax coloring must be able to find the colon or right brace character that ends the expression-field. Not implementing this can have surprising results in the case of nested string literals, i.e. that the contents of the nested literal is colored as if it was not a literal while the surrounding expression-field is not colored as an expression.

```
std::string value = "Hello,";
f"Value {value + " World"}";
```

Above you can see the mis-coloring provided by the tools that produced this document.

As there may not be much of a lexer available in some tools it is a valid question how much trouble it would be to implement correct syntax coloring in those tools. It turns out that as all tokens that need to be handled are single character. So even without lexer the problem is not really hard. This has been proven by the stand alone implementation of this proposal which works on a character by character basis.

Also, many other languages include both string interpolation and syntax that allows brace pairs and ternary operators in expressions, so handling this in C++ may be just to enable this type of parsing for yet another language.

8 Nested expression-fields

Nested expression-fields inside the format-specifier of an extraction-field are always extracted regardless of if the formatter for the data type can handle this or not. While it seems odd to use the `{` character in a format-specifier for some other purpose than to start a nested expression-field it is allowed for a user-defined formatter. To avoid extraction of the nested expression-field in this case you can quote a curly brace inside a *format-specifier* by doubling it as elsewhere in the f-literal. Note that no *standard* format-specifier allows braces except for dynamic width or precision, not even as fill characters.

9 Encoding and raw literal prefixes

The f prefix can be combined with the R and L prefixes. Theoretically it can also be combined with with the u, U and u8 prefixes, but as `std::format` is only available for char and wchar_t this does not currently work, except for a user defined function taking a unicode format string. Another proposal to would be needed to address this limitation of std::format.

The order of encoding, formatting and raw prefixes is fixed so that any encoding prefix comes first, then the f-literal prefix and finally the raw literal prefix.

10 String literal concatenation

f-literals can be concatenated to other f-literals and to regular literals. Encoding prefixes must be consistent as for regular literals. When a sequence of string literals contains at least one f-literal the result of preprocessing is one `__format__` call containing the resulting literal and all the extracted expression fields.

10.1 Quoting of non-f literal contents during concatenation

A problem arises when concatenating f-literals with regular literals containing `{` or `}` characters. To avoid these characters occurring in a regular literal from being treated as the start or end of an expression field after preprocessing string concatenation doubles braces in regular literals when concatenated to f-literals.

```
// The programmer wrote
"Start point {" f"{x}, {y}" "}"

// The preprocessor output
__format__("Start point {{{}, {}}}", (x), (y));

// Program output
Start point {15, 65}
```

An alternative would be to ignore this very fringe issue and require programmers to double braces in string literals that are concatenated with f-literals, as it is such an edge case. The problem with this is that the literal could be inside a macro used with both regular- and f-literals.

11 User defined suffixes

It is unclear how user defined string literal operator functions would work when applied to a f-literal. The problem is that the preprocessor doesn't know if a certain identifier that follows a f-literal is a user defined literal suffix or not. Currently C++ does not define any infix operators consisting of an identifier, but there are a couple of proposals to introduce such operators: `in`, `as` and `match` all fall in this category. To allow for such operators this proposal does not handle user defined suffixes to f-literals, and thus such identifiers are left after the resulting `__format__` function call.

12 Contexts where string interpolation works

With the risk of stating the obvious: String interpolation only works in contexts where calling a C++ function call is allowed. This excludes uses in the preprocessor such as `#include` filenames and uses in `static_assert` and the `deprecated` attribute where only a string literal is allowed. If `std::format` gains a constexpr specifier it is the intent of this proposal to allow string interpolation in places where this would allow `std::format` to be used, such as in non-type template arguments and to initiate constexpr variables. If contexts like `static_assert` and the `deprecated` attribute get the ability to handle a constant expression of character string (or string_view) type string interpolation should work there too.

In fact, by the transformation in the preprocessor of the f-literal to a call to `__format__` other parts of the compiler will handle the different contexts where this is or isn't allowed in different standard versions, as well as errors related to trying to format non-constant expression-fields when f-literals are placed in constexpr contexts.

It is assumed that any later proposal that makes `std::format` constexpr will also add constexpr specifiers appropriately on the standard library's implementation of `__format__` too.

13 Code breakage risk

In keeping with current rules macros named as any valid prefix sequence are not expanded when the prefix sequence is directly followed by a double quote. This means that if there is a parameterless macro called f that can produce a valid program when placed directly before a double quote introducing string interpolation is a breaking change. The same could be said about Unicode and raw literal prefixes when these were introduced, and apparently a few C code bases were broken when the unicode prefixes were added.

Due to the combinations of prefixes the macros that are no longer expanded if followed by a " character are:

```
f, fR, Lf, LfR, uf, ufR, Uf, UfR, u8f, u8fR
```

None of these seem like a very likely candidate for a macro name, and even if such macros exist the likelihood of them being reasonable to place before a string literal without space between is low.

Depending on the contents of the macro this breakage may be silent or loud, but if the macro did something meaningful there should most often be errors flagged when the macro contents disappears and furthermore the data type will most likely change causing further errors. One macro that may cause problems is a replacement for the current `s` suffix that can be written as

```
#define f std::string() +
```

With such a macro (with one of the names listed above) some problems can be foreseen. It could be that some committee member knows of similar breakage happening when the prefixes already added after C++03 were introduced:

```
R, U, UR, u, uR, u8, u8R
```

If the committee at large does not know of such cases it seems unlikely that the new prefixes would cause many problems due to this.

14 Debugging feature

Python has a neat debugging feature which allows printing variables easer: If the expression ends in a = the text of the expression is considered part of the remaining-literal:

```
f"{x=}";

translates to

__format__("x={}", x);
```

The only syntactical problem with this occurs if the expression ends with `&MyClass::operator=` where the = would be treated as the trailing = unless the previous token is `operator`. It is proposed that the token sequence `operator=` at the end of a expression-field should be treated as an error. This simple logic does not reduce programmer expressibility as you can't format a member function pointer anyway, and you can't even explicitly cast it to void* to be able to print the member function address.

15 The `__format__` function

The reason that the lexing of a f-literal results in a call to a function called `__format__` is to allow for code bases that don't use the standard library to still do formatting using their own facilities. The name `__format__` is tentative but the name finally selected must be something that is obscure enough to be used as an unqualified function name without clashing.

An alternative would be to put this name in a special namespace, but then the namespace name would have to be obscure enough instead. A further alternative would be to place it in a sub-namespace of std. In this case we don't need an obscure name as everything is inside the std:: namespace anyway. This has the ideological problem that a code base that doesn't use the standard library has to declare a std namespace itself to be able to put the implementation of the `__format__` function there.

15.1 An alternative spelling

It would be possible to use a special spelling for the `__format__` function to indicate that it is really special. One such spelling that would be rather logical is `operator f""()` which is consistent with how *postfix* literal operators are declared today. Note however the difference that the `f` here must be exactly that letter, we're not supporting any other prefixes as all other needs are covered by x-literals. Other than this special name the function is just a regular function, there are no restrictions on argument types other than that the first argument should be constructible from a string literal for the function to be callable.

In this variation the lexer must output an explicit call to the operator function when a f-literal is encountered:

```
f"Value {x}";

// Translates to

operator f""("Value {}", (x));
```

Note that the lexer will have to make sure to *not* start treating the quotes after the `f` as the start of a f-literal if preceded by the operator keyword. This is needed to allow the user to declare the function and to write explicit calls to the operator as can be done with all other operators.

This proposal opts for a named function like `__format__` as it doesn't require any other changes to the core language to generate the function calls.

15.2 The standard implementation of `__format__`

The standard library implementation of `__format__` is located in the `<format>` header and just perfectly forwards to std::format. The different character types must be handled by separate overloads due to the consteval constructor of `std::basic_format_string`.

```
template<typename... Args>
std::string __format__(std::format_string<Args...> lit, Args&&... args) {
    return std::format(std::move(lit), std::forward<Args>(args)...);
}

template<typename... Args>
std::string __format__(std::wformat_string<Args...> lit, Args&&... args) {
```

```
    return std::format(std::move(lit), std::forward<Args>(args)...);
}
```

With the alternative spelling this instead becomes:

```
template<typename... Args>
std::string operator f""(std::format_string<Args...> lit, Args&&... args) {
    return std::format(std::move(lit), std::forward<Args>(args)...);
}

template<typename... Args>
std::string operator f""(std::wformat_string<Args...> lit, Args&&... args) {
    return std::format(std::move(lit), std::forward<Args>(args)...);
}
```

Regardless of which syntax is used this proposal puts the functions in the global namespace.

15.3 Overloading `__format__`

If a code base defines the `__format__` function exactly as the default implementation above it would not be possible to include the `<format>` header or do `import std;` as two definitions of the same function template would then be available. This can be solved by adding a constraint `requires(true)` to the user defined function to make it always be selected instead of the one provided by `<format>`. This allows std::format (declared in `<format>` to be called from the user defined `__format__` function.

```
template<typename... Args> auto
__format__(std::format_string<Args...> lit, Args&&... args) requires(true) {
    return "my: " + std::format(std::move(lit), std::forward<Args>(args)...);
}


// Usage

int main()
{
    std::cout << __format__("Value: {}", 5);
}
```

Compiler Explorer: Link

16 Solving the f- versus x-literal issue with an attribute

R2 of this proposal introduces a new attribute `cpp_format_string`(N) which can be set on a function declaration to indicate that if a call site refers to this function name and has an f-literal as its N::th argument the `__format__` call created by the preprocessor is to be expanded in line before calling the function. This replaces the x-literals introduced in R1 of this proposal, reducing cognitive load and learning curve, while not relying on other proposals as R0 did.

Note: This attribute is not ignorable. The authors think that requiring *all* attributes to be ignorable was maybe not such a good idea.

To be feasible this attribute based system needs to have rules for how to handle overload sets like the one of std::print shown below.

```
// Declarations in <print>
template<typename... Args>
void [[cpp_fmt_string(1)]] print(format_string<Args...> fmt, Args&&... args);
```

```
template<typename... Args>
void [[cpp_fmt_string(2)]] print(ostream&, format_string<Args...> fmt, Args&&... args);

// In user code
std::print(f"Value {1}");                            // #1
std::print(std::cerr, f"Wrong value {2}");          // #2
std::print("String value: {}", f"Some value {3}");  // #3
```

To correctly handle this we need a couple of rules which rely on knowing which arguments are `__format__` calls. Lets start with the case that we have one `__format__` call as on lines #1 to #3 above. In these cases the question is whether to expand the arguments of the `__format__ call` or not.

To resolve this the compiler separates the overload set into two *partitions*, containing the overloads that match the position of the `__format__` call and the overloads that don't, respectively. Overload resolution is then performed separately for the two partitions, and if this succeeds in only one partition this is the result. If both partitions succeed in the overload resolution the call is ambiguous.

In example #1 where the f-literal is in position one the first overload is matched to the expanded contents of the `__format__` call and this succeeds. The second partition containing the second overload is matched towards the argument list where `__format__` is not expanded. This doesn't match as there are too few arguments.

In #2 the first overload is rejected as you can't use cerr as a format string. The second overload, with the `__format__` clause expanded, succeeds as intended.

In #3 the first overload is in the overload partition without expansion and the second overload is in the expansion partition. But as a string literal is not implicitly convertible to an ostream the second partition does not produce a viable function to call, so the the first `std::print` overload will be called after first calling std::format from the `__format__` constituting the second argument.

```
// Example repeated
std::print("String value: {}", f"Some value {3}");  // #3

// No expansion occurs.
std::print("String value: {}", __format__("Some value {}", 3));

// The inline __format__ function calls std::format
std::print("String value: {}", std::format("Some value {}", 3));

// Output
String value: Some value 3
```

16.1 Multiple f-literals in the argument list

It should be allowed to have more than one f-literal in an argument list. In this case only the *last* f-literal is considered for expansion, so the overload set is partitioned into two as described above and the overload selection rule is the same. The rationale for this simple rule is that there is actually no good use case for having arguments after an expanded f-literal at all, while there are use cases for having non-expanded f-literals before an expanded f-literal.

For the following example lets add a `std::print` overload which takes a `std::filesystem::path` and prints to a file of that name:

```
// Declarations in <print>
template<typename... Args>
void [[cpp_fmt_string]] print(format_string<Args...> fmt, Args&&... args);

template<typename... Args>
```

```
void [[cpp_fmt_string]] print(ostream&, format_string<Args...> fmt, Args&&... args);

template<typename... Args>
void [[cpp_fmt_string]] print(filesystem::path&, format_string<Args...> fmt, Args&&... args);

// User code:
std::print(f"File{n++}.txt", f"The value is now {value}");  // #4
```

The call at #4 subdivides the overload set into one partition with the two overloads with the format string in the second position and the first overload constitutes the partition without expansion.

In the partition where the second f-literal is expanded the third overload is viable as a std::string can be converted to a std::filesystem::path.

In th partition without expansion the overload resolution fails as a std::string can't be used as a format string.

In case you actually want this to provide the second f-literal as an *extra* parameter to the first print overload you would now have to add a call to std::format to make sure the f-literal is converted to a string and thus prevent it from being expanded in the std::print call.

```
std::print(f"File{n++}.txt", std::format(f"The value is now {value}"));  // #5
```

In #5 the std::print call again has only one `__format__` call as argument and as it matches the first std::print overload's `cpp_fmt_string` attribute putting the first overload in its own partition, while the other overloads are in the non-expanding partition. The first partition's overload resolution succeeds (with an extra, ignored parameter) while the second and third overloads fails as a `std::string` can't used as a format string.

16.2 Should std::format have the `cpp_fmt_string` attribute?

The answer is somewhat surprisingly yes. While you would not typically write something like `std::format(f"Value: {3}")` it can be used as a disambiguator as shown in example #5 above. An `cpp_fmt_stringattribute` on std::format also solves issues related to for instance logging macros like this one:

```
void log(const std::string& entry);

#define LOG(...)                \
if (logging_enabled)            \
    log(std::format(__VA_ARGS__))

LOG("Value is now: ", value);           // As we write today

LOG(f"Value is now: {value}");          // As we want to write tomorrow.
```

As is easily seen the last LOG call would fail if std::format didn't have the `cpp_format_string` attribute. The alternative would be to remove the std::format call in the macro, but then users are forced to use f-literals for their logging or having two different macros.

16.3 Should arguments after the expanded f-literal be made illegal?

The only reason to allow more arguments after a format string than the format string will format is to allow for translation with the translated format string referring to arguments by numbers. Translation can never work with std::format as the format string must be a literal known at compile time, which it can't be if translation has been done before calling std::format. However, with the help of std::vformat it would be easy to create a special `tformat` function that does translation after compile time checking the format string:

```
template<typename... Args>
std::string tformat(std::format_string<Args...> lit, Args&&... args) {
    return std::vformat(translate(lit.get()), std::make_format_args(std::forward<Args>(args)...));
```

```
}

// Call site
auto text = tformat("Weight is {} kg", 12.3, 12.3 / 0.454);     // Second arg used by US translation
```

While this has some use cases it becomes less logical with f-literals as the extra arguments must be placed after the f-literal so it seems more useful to make extra arguments after an expanded f-literal a compile time error and let those use cases continue using a std::vformat based system similar to tformat.

To enforce this limitation we can just introduce a new class `strict_format_string` which does not allow that sizeof...(Args) is larger than the number of expression fields in the format string. This is then used in the implementation of `__format__` rather than just forwarding to std::format.

```
template<typename... Args>
std::string __format__(std::strict_format_string<Args...> lit, Args&&... args) {
    return std::vformat(lit.get(), std::make_format_args(std::forward<Args>(args)...));
}
```

This formulation prevents errors like this:

```
std::print(f"Value {x + y}", " is too large");    // #1

// Emitted by the preprocessor:
std::print(__format__("Value {}", (x + y)), " is too large");

// Expands to this before overload resolution as std::print has a cpp_fmt_string attribute.
std::print("Value {}", (x + y), " is too large"); // #2
```

It is easy to think that both string literals are going to be used in #1 but inspecting what the compiler expands this to in #2 it becomes obvious that the second string literal will be ignored. Writing code like #1 should instead cause a compile time error.

17 Creating printf format strings

The authors were asked to present in SG22 - C liaison and gave some thought to what could be done for printf format strings. Using f-literals with printf can be solved by an attribute `c_fmt_string` which indicates that apart from expanding the argument list in line the remaining literal is to be converted to a C-style format string by these transformations:

- Replacing % with %%.
- Replacing {} with %v.
- Replacing {:format_spec} with %format_spec if format_spec ends with a letter except x.
- Replacing {:format_spec} with %format_specv if format_spec does not end with a letter or x.
- Replacing {} in the *format spec* with *.

The reason for placing %v in the format string is to separate the format string conversion from type deduction, which is a useful feature in itself.

17.1 printf type deduction

When the parameter corresponding to the `c_fmt_string` attribute is a string literal the compiler replaces all C-style format specifiers in the literal which end with a `v` with the appropriate formatting letter for the type of the corresponding argument. This functionality is required for f-literals as there are some types like intmax_t which are typedefs to unspecified primitive types, and thus don't have a portable format specifier. In contrast with a regular printf format string it is not possible to use PRIdMAX or similar macros in format specifiers of f-literals as string literal concatenation occurs after expression extraction.

As a bonus this feature allows regular calls of printf to use %v for all types. This provides a type-safe way to specify the format string without using string interpolation as such. Note that while printf allows format strings which are not literals, such format strings may not contain %v as printf has no way to replace those v's with something else at runtime.

Although printf does not support fill characters and alignment this could be added to bring printf formatting to an equal standing with std::format except for the fact that printf can't differentiate between the "empty" float format and the g format. A separate letter could be added for this behavior though, such as r for roundtrip, and if desired r could be made the default way to format float values, to make std::format and printf work with the exact same set of f-literal format specifiers (for standard types) with exactly the same resulting formatting.

With this you can now write:

```
size_t x = 42;
float y = 7;
int w = 5;
printf(f"Percentage: {y + 3.14} %, {x * 2:{w}x"});

// The compiler emits what is effectively:
printf("Percentage: %r %%, %*zx", (y + 3.14), (x * 2), (w));

// Program output
Percentage: 10.14 %,     42
```

Note that this maintains the preprocessor independent of the language being compiled, the massaging of the format string for printf style format strings occurs in the core language compiler.

18 Implementation experience

There are two implementations of R0, both by Bengt.

18.1 A stand alone implementation

extract_fx is a stand alone pre-preprocessor which performs the new preprocessor tasks and produces an intermediate file that can be compiled using an unmodified C++ compiler. As this pre-preprocessor does not do macro expansion it can't support macros expanding to string literals that are to be concatenated with f-literals. All other uses of macros (including in expression-fields) are however supported by passing them on to the C++ compiler's preprocessor.

This implementation mostly works character by character but skips comments and and regular string literals, avoiding translating f-literals in commented out code or inside regular literals. Inside f-literals extract_fx handles all the special cases noted above, except digraphs.

This implementation can be seen as a reference implementation for syntax-coloring editors and similar tools which need to know where the expression-fields are but don't need to actually do the conversion to a function call.

Implementing extract_fx took about 30 hours including some lexing tasks that would normally be ready-made in a tool or editor, such as comment handling.

Note: This implementation does not support x-literals but a command line switch can be used to set the name of the function to enclose the extracted expression fields in, which can be used to get the `__format__` name.

18.2 Clang implementation

There is also a Clang fork which supports this proposal here, in the branch *f-literals*. This implementation is complete but lacks some error checks for such things as trying to use a f-literal as a header file name and when

the end of an expression-field is inside a macro expansion. This fork does not currently support x-literals and encloses all calls in a `::std::make_formatted_string` function call.

The Clang implementation relies on recursing into the lexer from inside lexing of the f-literal itself. This turned out to be trivial in the Clang preprocessor but could pose challenges in other implementations. With this implementation strategy the handling of comments, nested string literals and macros in *expression-fields* just works, as well as appointing the correct *code location* for each token. The only thing that was problematic was that string literal concatenation is performed inside the parser in Clang rather than in the preprocessor. To solve this f-literals collect their tokenized *expression-fields* into a vector of tokens which is passed out of the preprocessor packed up with the remaining-literal as a special kind of string literal token. In the parsing of *primary-expression* the string literal is detected and new code is used to unpack the token sequence and reformat it as a `make_formatted_string` function call. This code is also responsible for the concatenation of f-literals and moving all their tokenized expression-fields to after all the remaining-literals. Writing this code was surprisingly simple.

The Clang implementation took about 50 hours, bearing in mind that the extract_fx implementation was fresh in mind but also that the implementer had little previous experience with "Clang hacking" and none in the preprocessor parts.

Here is an example of the two step procedure used in the Clang implementation to first create a sequence of special string-literal tokens containing the remaining-literal and token sequence for each f-literal and then handing in the parser to build the basic_formatted_string constructor call.

```
// Original expression:
f"Values {a} and " f"{b:.{c}}"

// The lexer passes two special string-literal tokens to the parser:
// "Values {} and " with the token sequence ,(a) and
// "{:.{}}" with the token sequence ,(b),(c).

// The Parser, when doing string literal concatenation, finds that at least one
// of the literals is a f-literal and reorganizes the tokens, grabbing the stored
// strings and token sequences to form:
::std::make_formatted_string("Values {} and " "{:.{}}", (a), (b), (c))

// This token sequence is then reinjected back into the lexer and
// ParseCastExpression is called to parse it.
```

18.2.1 Lessons learned

A point of hindsight is that with more experience with the Clang preprocessor implementation it may have been possible to avoid all changes in the parser and doing everything in the lexer. The drawback with this approach would have been that when seeing a non-f literal the lexer must continue lexing to see if more string literals follow, and if at least one f-literal exists in the sequence of string literal tokens the rewrite to a `__format__` function call can be made directly during lexing. An advantage of this is that running Clang just for preprocessing would work without additional coding, but a drawback is that for concatenated literals without any f-literal there is a small performance overhead as the literal sequence must be injected back into the preprocessor which involves additional heap allocations. As only a small fraction of string literals involve concatenation this should not a be a significant issue.

19 Alternatives

A few other approaches to get string interpolation into C++ have been proposed, which are discussed here.

19.1 Language feature

A language feature that is applied strictly after preprocessing was proposed in [P1819R0] but as the string literal is then not touched by the preprocessor it can't contain macros and nested string literals have to be escaped. This approach would still need [P3398] to avoid dangling in the simple case of assigning an auto variable to a f-literal. A bigger disadvantage seems to be that, at least according to the proposal, there is no way to implicitly convert the f-literal to a std::string, usage is restricted to printing and ostream insertion.

19.2 Reflection

There has been ideas floated that reflection could solve this problem. As there are no concrete proposal texts that we are aware of we can only point out a few drawbacks that seem inevitable with such an approach.

Firstly the problem with macros already being handled when reflection can see the literal is the same as with the language feature approach, as well as the need to escape nested string literals. Secondly there seems to be no inherent way that the leading f can be handled by reflection. A mechanism where a certain identifier can be connected to some kind of reflection mechanism would be needed. The closest approximation would be something like `std::f("...")` which is not the level of ergonomy we aim at for string interpolation.

Furthermore, when analyzing the string literal, a new mechanism to convert each extracted string to the reflection of an expression is needed. Currently it however seems that *token sequence* based code injection is more likely to be standardized than string based code injection so to support reflection based string interpolation would require additional support that can convert a string_view to a token sequence.

As a final remark reflection based string interpolation would be relying on compile time code execution for each f-literal which would add to compile times. The code to tokenize the string literal to find the end of expressions involves considerably more computations than the current string literal validation done by the basic_format_string constructor.

20 Wording

No wording yet, although some investigation into the phases of translation have been made, which so far has led to the reordering of extraction and macro replacement to make f-literals fit with the definition of the phases of translation. There are two wording strategies discussed below, we call them *recursive lexing* and *literal splitting*. The recursive lexing strategy is easy to formulate but could be harder to implement in some existing preprocessors. The literal splitting strategy avoids recursion in phase 3 at the expense of a somewhat more complicated string concatenation procedure in phase 6.

Note that these strategies relate to standard wording, implementations may diverge from either of these and both strategies produce the same output from phase 6. One strategy will eventually be selected for wording.

20.1 Recursive lexing

This strategy recurses into the lexer in the midst of lexing a f-literal token when an extraction field start occurs. This strategy was used successfully in the Clang implementation, although some adjustments of the current character pointer used during lexing were needed.

20.1.1 Phase 3: Lexing

The actual extraction of the expression-fields occurs in phase 3, when pp-tokens are created from characters. When a f-literal is lexed a `__format__` function call is created by extracting all expressions according to the rules described above.

To make this work the lexer detects the leading f of a f-literal and uses separate code to create the complete `__format__` call from the contents of the f-literal. This code has to look for left braces and recurse to get the tokens of each expression field while already inside the lexing of a f-literal token. This requirement caused some concern that it may be infeasible or require vast changes in some preprocessor implementations.

As the complete `__format__` call is created in phase 3, left and right braces of an extraction field must be in the same literal as macro expansion as well as string concatenation occurs in later phases.

20.1.2 Phase 4: Macro expansion

Macro occurs after in phase 4. There is nothing special that happens in this phase, the `__format__` function call emitted by phase 3 for each f-literal is treated like any function call.

If expanded macros contain mismatched parentheses this could cause the later phases to not find the end of a `__format__` call correctly. In a real compiler this will probably be implemented in a way that finds such problems already in the preprocessor but it is not something the standard has to require, the standard in general doesn't specify how or when a ill-formed program is diagnosed.

20.1.3 Phase 6: String literal concatenation

String literal concatenation in phase 6 is updated to allow concatenation of f-literals, both with each other and with non-f literals. When a string literal or `__format__` call is encountered in phase 6 the next pp-token is inspected and if it is also a string literal or `__format__` call concatenation occurs. If such a sequence contains at least one `__format__`call the result of the concatenation is a `__format__` call containing first the concatenation of all the actual string literals and then the extracted expressions of all the `__format__` calls being concatenated.

The concatenation of the actual string literals works according to the current rules.

This definition allows the continued use of macros expanding to string literals which are used to generate control sequences for terminals etc.

Here is an example which shows different types of string literals being concatenated, some of which are expanded from macros and some of which are f-literals. Note that the expansion steps shown below are for illustrative purposes only, a preprocessor/compiler is free to take other steps or just one step as long as the result on the line #3 is the same.

```
#include <format>

#define LITERAL " lucky one."
#define FLITERAL f" {name},"   // #1
const char* name = "John Doe";

L"{Hello" FLITERAL fR"abc( you{LITERAL}}})abc";     // #2. Source code.

// Phase 3 creates __format__ calls for f-literals and handles R literals to get:
L"{Hello" FLITERAL __format__(" you{}}}", (LITERAL))

// Macro expansion is then done as usual, resulting in:
L"{Hello" __format__(" {},", (name)) __format__("you{}}}", (" lucky one."));

// String concatenation then transforms this further to:
__format__(L"{{Hello {}, you{}}}", (name), (" lucky one."));    // #3. After preprocessor

// The __format__ function then calls std::format at runtime to get:
L"{Hello John Doe, you lucky one.}"    // #4. The result of the f-literal.
```

Side note: The code block above is formatted as Python which makes the f-literals colored correctly (but the initial #defines are treated as comments and the C++ comments aren't). This indicates that tools that support Python source code coloring should have limited problems with coloring C++ f-literals.

When lexing the macro definition at #1 the f-literal is lexed by phase 3 so the macro defintion effectively reads:

```
#define FLITERAL __format__(" {},", (name))
```

One of the literals at #2 has an encoding prefix, one is a macro and one has f-prefix. During string literal concatenation he encoding prefix extends to all the literals as usual, while the raw prefix only applies to the immediately following literal. The extracted expressions from all the f-literals are moved after the concatenated literals to allow format to operate correctly when called from the `__format__` implementation in `<format>`.

A complication with this definition is that phase 6 has to be able to find the end of a `__format__` call created by phase 3. To do this it has to count matching parenthesis pairs to be able to detect which right parenthesis ends the `__format__` call. In a real compiler this information would probably be conveyed from the phase 3 code as extra information.

20.2 Literal splitting

In this strategy phase 3 subdivides a f-literal into interleaved string literal parts and extracted expressions. This relieves phase 3 from having to support recursing back into the lexer from within a f-literal (which is in itself a pp-token being lexed). The drawback of this is that the string literals in question must be tagged with *start*, *middle* and *end* to allow the string literal concatenation in phase 6 to construct a `__format__` function call from them. While this is rather simple some kind of stack is needed when nested f-literals are encountered.

20.2.1 Phase 3: Lexing

When phase 3 encounters a f-literal it scans for the first `{` as before and when found it emits the first part of the string literal directly, tagging it as a *start* literal. Further tokens are then lexed as usual until a `}` or `:` token which ends the expression field is encountered, as described above. At this point the lexer starts a new string literal (as if a " character had been encountered) and starts scanning for the next `{` character or the `"` that ends the literal. When a double quote is encountered an *end* literal is emitted while if a { character causes a *middle* literal to be emitted and a new scan for the end of an expression field is started.

When the expression field ends with a `:` token a format specifier starts. Format specifiers can also have expression fields so there is not much special happening, except that the lexer can optionally detect a syntax error if a colon ends an expression field nested in a format specifier.

To be noted is that while this arrangement avoids recursion in phase 3 there is still some state that needs to be kept by an implementation to be able to detect the end of an expression field. When a f-literal is encountered during the scan for the end of an expression field this information has to be saved in a stack and reinitialized. As the problem is recursive in nature this can't be avoided.

20.2.2 Phase 4: Macro expansion

Macro expansion occurs in phase 4. There is nothing special that happens in this phase, the string literal tokens emitted by phase 3 for each f-literal is treated equally regardless of how they are tagged.

If expanded macros contain mismatched parentheses this does not cause phase 6 to not find the end of a `__format__` call correctly. Instead the tokens inside the `__format__`call emitted by phase 6 will not constitute a well formed program and error messages will be emitted by the compiler.

20.2.3 Phase 6: String literal concatenation

With this strategy it is phase 6 that also handles interpreting the *start*, *middle* and *end* literals. Its task is both the concatenation of literals as we do today and to make sure that any expressions between *first* and *last* literals are collected and output as a `__format__` call when the last concatenated literal has been seen. This operation can be described rather concisely:

When a regular or start-tagged string literal token appears in the token stream a new *concatenation operation* starts. Its *resulting literal* is initiated with the string literal contents. If the token is start-tagged the concatenation operation then collects succeeding tokens into a token buffer. All middle-tagged string literals are

instead appended to the resulting literal. This scanning continues until an end-tagged string literal is encountered. When this happens the end-tagged literal is appended to the resulting literal and then the next token is inspected. If it is a regular or start-tagged string literal concatenation continues but for all other tokens the `__format__` call is emitted containing the resulting literal followed by the tokens in the token buffer.

When a regular or start-tagged string literal is encountered when a concatenation operation is active an inner concatenation operation starts, saving the outer concatenation operation's state. When the inner concatenation operation ends its result is appended to the outer operation's token buffer and the outer operation's state is restored.

When the outermost concatenation operation ends its resulting `__format__` call is appended to the result of phase 6.

The example is now treated like this in the preprocessor:

```
#include <format>

#define LITERAL " lucky one."
#define FLITERAL f" {name},"   // #1
const char* name = "John Doe";

L"{Hello" FLITERAL fR"abc( you{LITERAL}}})abc";     // #2. Source code.

// Phase 3 creates interleaved string literals and expressions, and handles R literals:
L"{Hello" FLITERAL s" you{", (LITERAL), e"}}}";

// Macro expansion is then done as usual, resulting in:
L"{Hello" s" {", (name), e"}," s" you{", (" lucky one."), e"}}}";

// String concatenation then transforms this further to:
__format__(L"{{Hello {}, you{}}}", (name), (" lucky one."));    // #3. After preprocessor

// The __format__ function then calls std::format at runtime to get:
L"{Hello John Doe, you lucky one.}"    // #4. The result of the f-literal.
```

The start and end literals are annotated with s and e prefixes above. As in the recursive strategy the f-literal in the macro definition is expanded in phase 3, in this strategy to:

```
#define FLITERAL s" {", (name), e"},"
```

21 Acknowedgements

Thanks to Hadriel Kaplan who initiated this effort and wrote an insightful draft proposal that was used as a starting point for this proposal and fruitful discussions in the following few months.

Thanks to Robert Kawulak and Oliver Hunt for contributing to the R1 version.

Bengt would like to thank his employer ContextVision AB for sponsoring his attendance at C++ standardization meetings.

22 References

[P1819R0] Vittorio Romeo. 2019-07-20. Interpolated Literals.
https://wg21.link/p1819r0