String interpolation

Document #:	P3412R3
Date:	2025-12-14
Project:	Programming Language C++
Audience:	SG16, 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 [P3298R1] and [P3398R0] 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, After presentation to SG16 and SG22

— 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.

1.4 R3, This revision

— Removed printf support but keep as a future direction (pending WG14 support).
— Added text about how to handle stringization when the *stringize argument* contain f-literals.
— Clarify that `_Pragma` (and probably Microsoft's `__pragma`) can be allowed in expression-fields.
— Noted that lambdas, gcc *statement-expressions* and Clang *^blocks* inside an expression-field doesn't cause problems detecting the end of the expression-field.
— Clarified that raw literals don't interact with f-literals in any special way except that the reversal of phase 2 (line splicing) must only be done in the parts of the f-literal that are not expression-fields.
— Discuss implications of user written `__format__` calls wrt. concatenation and stringization.
— Add section about `module` related interactions (basically none).
— Some English language corrections.
— Noted that :: can indeed start a format-specifier (for a *range-formatter*). This proposal revision does not solve that problem, this will always be interpreted as one pp-token and never start a format-specifier.

— Added `constexpr` to the declarations of `__format__` as `std::format` is `constexpr` after [P3391R1] was accepted for C++29. However not in library wording as P3391 has not emerged in the latest draft yet.
— Add the different vendor's feedback about implementability.
— Removed possibility to have ternary operators on top level of expression-fields. This allows future uses of `?` as for instance a prefix operator to check for existence of a value. It also simplifies the detection of the end of expression-fields somewhat.
— Add a first version of wording which uses a combination of the two wording strategies discussed in R2.
— Remove the attribute `cpp_string_format` and instead expand the `__format__` argument list if overload resolution would otherwise fail or require calling consteval function in the implicit conversion sequence.
— Remove the debugging feature as this would prevent emitting the preceding *string-literal* pp-token from phase 3 before lexing the *expression-field* contents. While this feature was easy to implement in Clang it would be impossible to implement in a compiler which can not buffer pp-tokens in phase 3.

2 Abstract

This proposal adds string interpolation (so called f-literals) to the C++ language. After phase 6 each f-literal has been transformed 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 character type.

In addition this proposal introduces a special rule to support functions like `std::print` without first calling `std::format`: If overload resolution does not find any viable function or if this requires calling a consteval function in the conversion sequence, and at least one argument is a `__format__` call the arguments of the last `__format__` call are expanded in line and overload resolution is tried again.

This proposal (currently the R0 revision) has been implemented in a Clang fork, which is available on Compiler Explorer. This fork has been used for compilation over 6000 times in the first half-year of 2025. This proposal 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 use string interpolation to format Points. This is in contrast with R0, where [P3298R1] and [P3398R0] were required to make the below examples work as expected and R1 where the f-literals in `std::print`like functions 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()}";      // a is initialized from a std::format call.

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 can't be called with a std::string,

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. In R3 this is changed to a rule that handles failed overload resolution if a `__format__` call is in the argument list.

Some text of this proposal were 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, for instance based on a $ or % prefix. (full list). As `std::format` already uses the Python syntax with {} enclosed arguments 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 that `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 early phases of translation.

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 parentheses. 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 lexer.

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))

// Not allowed:
```

```
f"The answer is {boolVar ? 17 : 42} ";

// *As it is transformed to the erroneous:
__format__("The answer is {: 42}", (boolVar ? 17 ));
```

A consequence of referring to the grammar for *expression* is that 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.

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.

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 tools such as syntax coloring editors may not contain any expression parser. 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. are handled by the normal lexing code.

To detect the end of the expression-field just scan for the first } or : pp-token, skipping over nested parenthesis, curly brace and square bracket pairs as well as nested string and character literals. This rule unfortunately forbids the ternary operator outside of all parentheses as the `:` it implies would be treated as the end of the expression field, but this seems to be a low price to pay to get such a simple rule.

Also note that two adjacent colons are always treated as a scope resolution operator. This prevents setting a format of the elements of a container being formatted, but this can be fixed by allowing a space first in a container format string. This would mean "format the container normally". This allows writing two colons with a space between them to the same effect as a double colon. Allowing this will require a separate proposal.

As `std::format` does not handle the digraph `<%` to start an expression field it seems unnecessary to handle `:>` as a square bracket inside expression-fields. This allows for the fairly common case where a format-specifier starts with `>` (right alignment with space as fill character). It would however be easy to allow the `:>` digraph to end nested square brackets in expression-fields as long as a `:>` character sequence which when interpreted as a digraph is unmatched is reinterpreted as the start of a format-specifier. If normal lexing handles digraphs it may be easier to allow `:>` than not. In this case the f-literal code in phase 3 must be able to determine if the `}` token that followed the expression-field was formed from a digraph so that the next string literal argument being formed can start by the correct character sequence. Allowing digraphs in *expression-fields* is to be seen as an extension so portable code should not use them in extraction fields.

Note that lambdas, which may contain any type of code including for instance goto labels, always contain this code inside matched braces, so any colons will be ignored when detecting the expression-field end. The same goes for *statement-expressions* of gcc and *blocks* of Clang.

Further into the future, if more uses of colons inside expressions are specified, these uses of colons will require the *expression-field* to be enclosed in parenthesis just like the use of the ternary operator does.

7.2 Preprocessor directives in expression-fields

Preprocessor directives inside expression-fields are 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 in an expression-field in an f-literal. Regardless of if the `#if` condition is true or false an unterminated string literal would result. In an implementation that strictly follows the phases of translation enforcing this restriction could be problematic so the proposal is to make this IFNDR.

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. It is however likely that compilers that don't adhere to the phases of translation would have problems implementing this.

7.3 _Pragma in expression-fields

In contrast with directives there seems to be no reason to forbid the use of `_Pragma` in expression-fields. The only allowed contents of a `_Pragma` is a string literal and with this proposal the preprocessor sees this as just any function call with a string literal as argument.

Note that as the argument of `_Pragma` is a string literal, not an expression resulting in a string. Thus f-literals can't be used as the argument of a `_Pragma`.

Microsoft has a similar `__pragma` construct which has a more free form argument list. This is probably not a problem to allow in expression-fields but again, f-literals would not be possible in the argument list of `__pragma` as the arguments are not C++ expressions..

7.4 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 = f"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.

7.5 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 user-defined formatters. 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.

8 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 such tools. It turns out that as all tokens that need to be handled are single characters. 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 in expressions-fields, so handling this in C++ may be just to enable this type of parsing for yet another language. In particular, tools which handle Python f-literals correctly should be able to use same mechanism for C++ with decent results.

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 user-defined functions taking format strings with the corresponding character types.

A raw f-literal, just like a raw literal used as a `std::format` format string today, involves no special handling of braces compared to a non-raw format string. While nested double quotes and newlines do not have to be escaped in a raw literal you still have to double braces to avoid them being treated as the start or end of an expression field in a raw literal. The contents of expression fields in raw f-literals is still lexed as a token sequence which means that comments and excessive whitespace is removed just as in a non-raw f-literal.

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

9.1 The reversal of phase 2

When lexing a raw string literal the current standard text requires the effect of the line splicing of phase 2 to be undone inside the raw string literal. The compiler will need next to magical powers to be able to do this according to the standard as reversal must be done before identifying the d-char sequence that starts and ends the raw literal, while finding the end of the raw string literal requires knowing the d-char sequence. Yet compilers have successfully implemented raw string literals. With raw f-literals this reversal applies to each individual string-literal token that is part of the *f-literal-construct*.

As the only implementation strategies that can be used to reverse the effects of phase 2 for the contents of a raw string literal before identifying the d-char sequence required to know when it ends is to save a copy of the source text before phase 2 and use it instead of the output of phase 2 while lexing the raw string literal or to actually do the reversal while scanning the output of phase 2, operating under the as-if rule.

In the first implementation strategy phase 2 would have to prepare the land by allowing phase 3 to somehow find the corresponding character in phase 2 input stream when it finds the " starting a raw string literal. This seems cumbersome so the second strategy is more likely and in that case raw f-literals don't pose any additional problems, the same reversal while scanning works well.

In some compilers (at least in Clang) the first few phases of translation are performed in one pass, not saving any intermediate program text, which makes the second implementation strategy the only viable option.

10 String literal concatenation

f-literals can be concatenated with other f-literals and to regular literals. Encoding prefixes must be consistent as for concatenation of 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 prevent these characters occurring in a regular literal from being treated as the start or end of an expression field after string concatenation they are doubled when concatenated with f-literals.

```
// The programmer wrote
int x = 15;
int y = 65;

"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. The problem with this is that the literal could be inside a macro used with both regular- and f-literals. Such concatenation of literals with macros expanding to literals is used to send control sequences to terminals.

11 User-defined suffixes

It is unclear how user-defined string literal operator functions would work when applied to an f-literal. The problem is that the preprocessor doesn't know if a certain identifier that follows an 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 where only a string literal is allowed. Now that `std::format` gains a constexpr specifier with the acceptance of [P3391R1] string interpolation works 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 will work there 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 allegedly 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.

14 Modules related aspects

There seems to be no adverse interactions between modules and f-literals. f-literals do expand macros inside expression-fields but this happens when the source code is lexed, not later. This means that a header file which is #`included` works the same as if it is instead imported as a header unit. Even in the case that a f-literal is part of the replacement-list of a macro in a header file there is no difference depending on whether this file is subsequently #included or imported as a header unit as the lexing of the macro source converts the f-literal to a *f-string-construct* before phase 4 when the preprocessor directives are handled.

In module units f-literals are always part of an expression and can not be exported as such. Any macros #`defined` in the module unit's global module partition or in files #`included` from it are expanded in expression-fields as usual. Any f-literals in macro replacement-lists in module units are converted to `__format__` calls when those macros are lexed but their usage can never occur outside the module unit as macros are not exportable.

15 The `__format__` function

One reason that the lexing of an 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 other reason is that a `__format__` call as a function argument is treated specially during function overload resolution, which is explained below.

15.1 Naming

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. This name is reserved for the implementation according to 5.11 [lex.name] but so are many alternatives such as `_format`, `_Format` etc.

An alternative would be to put the function 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.

15.1.1 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. Other than this special name the function is just a regular function, there are no restrictions on argument types. However, the first parameter should be constructible from a string literal for the function to be callable.

In this spelling alternative phase 6 must output an explicit call to the operator function when an f-literal is encountered:

```
f"Value {x}";

// Translates to

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

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

This proposal opts for a named function like `__format__` as it doesn't require any changes to the core language to be able to declare the function.

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

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

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

Regardless of which spelling 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

15.4 Source code `__format__` calls

If the program source code contains a call to `__format__` this is not eligible for concatenation with other string literals or `f-string-keyword` constructs created from f-literals in phase 3. As phase 6 looks for the `f-string-keyword` token source code `__format__` calls are not treated differently than any other function call until overload resolution in Phase 7.

An alternative would be to make `__format__` calls in the source code illegal. This would require additional specification and implementation work which does not seem warranted.

16 Overload resolution special treatment

The question of how f-literals can evaluate to `std::string` except when provided to functions like `std::print` which want to do the formatting themselves, for instance to gain performance, has had different solutions in each revision of this proposal. In R0 the `__format__` function returned a special type annotated according to two supporting proposals to solve this. R1 gave up on allowing *f-literals* to serve both purposes and instead added an accompanying *x-literal* variant to be used when calling functions like std::print. R2 of this proposal introduced a new attribute `[[cpp_format_string(N)]]` used on functions like `std::print` to indicate that if a certain parameter was a `__format__` call it was to be expanded to its arguments before doing function overload resolution.

In R3 this attribute is replaced by expanding a `__format__` call in the argument list of a function call to its arguments if the function overload resolution would otherwise fail, or if the parameter type matched to the `__format__` argument would require invoking a `consteval` constructor or conversion function in the implicit conversion sequence. The last rule is required now that `__Format__` is `constexpr` and could *sometimes* be a valid first argument to `std::print`. While it would be better to keep the `__format__` function call if it is possible to evaluate at compile time this may not be known when overload resolution is performed.

```
std::string a;

a = f"Value {0}";                                  // #0
std::print(f"Value {a}");                          // #1
std::print(f"Wrong value {1}");                    // #2
std::print("String value: {}", f"Some value {3}"); // #3
```

In #0 overload resolution for the `operator=` call succeeds and the type of its source operand in the selected overload is `std::string&&`. The conversion sequence from the `std::string` return type of `__format__` thus doesn't even contain a user defined constructor or conversion function that could be `consteval`, so the `__format__` call is not expanded and the variable is move-assigned from the return value of `__format__`.

In #1 the `std::print` overload which takes a `std::format_string` as its first parameter matches the return type of `__format__` but as the constructor of `std::format_string` that is called in the implicit conversion sequence is `consteval` the `__format__` call is expanded, ensuring optimal performance of `std::print`.

In #2 `__format__` is expanded even though in this particular call site it would be possible to evaluate at compile time, as the conversion sequence for the `format_string` which is the first parameter of `std::print` contains a `consteval` constructor call. This rule is important as some compilers may not be able to deduce whether the `__format__` call can be evaluated at compile time when overload resolution is performed. An alternative is to make it implementation defined whether `__format__` is expanded or not if it can be compile time evaluated.

In #3 `std::print` overload resolution succeeds without expanding the `__format__` call so no special rules are in effect: the second argument to `std::print` is the `std::string` returned by `__format__`.

Here are the steps of this process in #3 for clarity:

```
// 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, but in this case only the *last* f-literal is considered for expansion. The rationale for this is that in the normal formatting case that expansion of a f-literal results in any number of arguments and thus the matching parameter must be a pack, which is only allowed last in a parameter list.

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

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

template<typename... Args>
void print(ostream&, format_string<Args...> fmt, Args&&... args);

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

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

For the call at #4 both the first and last overloads are viable after expansion but in the first overload the first of two `__format__` calls in the argument list would have to be expanded, which is not allowed. So the last `__format__` argument is expanded and overload resolution is retried as if the call site looked like this:

```
// Last __format__ call expanded.
std::print(__format__("File{}.txt", (n++)), "The value is now {}", a);
```

Now only the third `std::print` overload viable, as a `std::string` can be converted to a `std::filesystem::path`, and the unexpanded `__format__` call still prevents the first overload from being viable due to the `consteval` constructor.

If the rule had instead been to expand the first `__format__` call the first overload would have been selected but

then the std::string returned from the second `__format` call would have been ignored by `std::print`. This is always the case as the number of arguments resulting from the expansion of a `__format__` call always matches then number of *expression-fields* in the f-literal.

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 has only one `__format__` call as argument and as it matches the first std::print overload after expansion this overload is called, ignoring the `std::string` that `std::format` returns, so this is meaningless.

It is conceivable but not probable that there exists some format-like function in some code base that takes a mandatory string parameter after the arguments matched to a format string, but to implement this the programmer would have to use TMP to special treat the last element of the parameter pack. Such functions are still supported by f-literals as long as you don't try to create the value of the last parameter from a f-literal.

16.2 What happens when std::format is called with a f-literal?

As the `__format__` function calls `std::format` it is mostly meaningless to wrap a *f-literal* in a call to `std::format`. However, explicitly calling `std::format` can be used as a disambiguator as shown in example #5 above. As std::format has a `std::format_string` as its first parameter just like `std::print` the `__format__` call is expanded so instead of calling `std::format` from `__format__` the explicitly stated `std::format` call is the only one that actually runs.

This 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);         // What we write today

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

When the second LOG macro invocation expands its arguments as `__VA_ARGS__` the `__format__` call is expanded just as in the previous example [This is not entirely true as macro expansion in phase 4 works on the intermediate *f-string-construct* emitted from phase 3 but the effect is the same].

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 the extra arguments. Translation can't' 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 it. 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 original 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
```

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

This is a useful function but it doesn't expand to *f-literals* very neatly as the call site above would have to be formulated as:

```
auto text = tformat("Weight is {weight} kg", weight / 0.454);       // Extra arg used by US translation
```

This said it does seem warranted to preclude extra arguments after an *f-literal* i.e. to only expand it if it is the last argument in the argument list. As there may be other use cases which make more sense we keep this question open for now.

17 printf format strings

The authors were asked to present in the SG22 - C liaison group and gave some thought to what could be done for `printf` style format strings. This section is a possible future direction, it is not intended to be standardized by this proposal as it requires extra liaison with WG14.

This meeting resulted in the R2 approach to handling `std::print` and `printf __format__` call argument expansion where an attribute is used to indicate where a `__format__` call was to be expanded (i.e. if it was the format string). As this approach was abandoned in R3 there is no way that `printf` can get *f-literal* support using the same system.

However, within WG14 it would be possible to pursue format strings of some kind by introducing an attribute or specifier on `printf` like functions and following the system detailed below.

17.1 Converting format strings to `printf` style

Using f-literals with `printf` can be solved by a specifier `c_fmt_string` which indicates that apart from expanding the argument list in line the remaining literal is to be converted to a `printf` style format string by these transformations:

— Replacing % with %%.
— Replacing {{ with { and }} 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 ends with x.
— Replacing {} in the *format-spec* with *.

This transformation is done during parsing, not by the preprocessor, as it does not know which specifier (if any) a function has.

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.

17.2 printf type deduction

When the parameter corresponding to the `c_fmt_string` specifier is a string literal the compiler replaces all `printf` 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-field* extraction in phase 3, which relies on matched braces in the *f-literal* before concatenation.

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 round-trip, 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.

Unfortunately The `std::format` syntax for alignment can't be used in C format strings as `%d^` could be interpreted as a center alignment with fill character `d` or as a `%d` specifier followed by a `^` character to be preserved in the output. However, reversing the order to put the `<`, `>` or `^` first in the format specifier would work unambiguously. If this is introduced the compiler would have to reverse the order if the source code f-literal are to retain full compatibility between `std::format` and `printf`.

18 Implementation experience

There are two implementations of R0, both by Bengt.

18.1 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: 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 instead of the default std::format which is more suitable for experimentation with a regular C++ compiler.

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 an f-literal as a header file name and when the end of an expression-field is inside a macro expansion. This fork encloses all calls in a `::std::make_formatted_string` function call according to R0 of the proposal, but without the supporting proposals, so some dangling problems can be expected.

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 lexed *expression-fields* into a vector of tokens which is passed out of the preprocessor packed up with the std::format prepared string-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 lexed 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 an 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.

18.3 Vendor feedback about implementability

Compiler implementers were contacted to comment about possible implementation problems, mainly when it comes to the possibility of requiring recursive lexing.

18.3.1 Clang

As an implementation is available in a Clang fork this is clearly possible, as described above. According to Matt Godbolt this compiler had 6000 compiles as of June 2025.

18.3.2 EDG

The implementer responsible for the EDG preprocessor responded: *I don't think this would pose any problems for our preprocessor implementation.*

18.3.3 GCC

From a gcc compiler implementer: *I don't see why it wouldn't be implementable.*

This came with a question regarding `__format__` calls in the source code which has been addressed in this revision.

There was also a question regarding raw f-literals which was already discussed in previous revisions.

18.3.4 MSVC

As MSVC has two separate preprocessor implementations, one legacy and one standard compliant the work to implement this new feature in both would be larger, especially as the legacy implementation was started over 40 years ago. While Microsoft may elect to not implement f-literals in the legacy preprocessor this may not be feasible due to customer requirements.

As yet we have not received specific feedback on the foreseen implementation effort, but we did get some other valuable comments which led to some clarifications in the R3 revision of this proposal. One important point was that stringization needs to handle f-literals.

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 [P3398R0] to avoid dangling in the simple case of assigning an auto variable to an f-literal. A bigger disadvantage seems to be that there is no way to implicitly convert the f-literal to a `std::string`, usage is restricted to printing and ostream insertion. To get a `std::string` from a f-literal you would have to write something like `std::format(f"Value: {v}")` which loses some of the gained terseness.

The major drawback of the language feature approach is however that reasonable preprocessor implementations (eg. Clang) does the lexing that's used throughout the compilation process in phase 3 which means that either the preprocessor has to treat f-literals specially and lex the expression-fields just like in this proposal, but then repack this into a structure that macro processing does not see, or the parser must be able to call back into the lexer from phase 7 to perform the lexing of the tokens of the expression-fields. This must be done on a token by token basis if actual parsing is to be used to detect where the expression field ends. The first approach gains nothing when it comes to detecting expression field ends, it just loses the macro handling ability for no apparent gain. The second approach could potentially create severe code structure problems for some reasonable compiler designs. Also note that after the preprocessor the transformation from source code to the execution

character set has been done which could affect the lexing of the tokens in the f-literals. The alternative to not perform phase 5 for f-literals is less problematic but it would require performing the character set conversion after forming the remaining string literal. The implications this approach may have for the ability to preprocess to a file and compile this file later on include at least that viewing the preprocessed file in a text editor may be problematic as it potentially contains a mix of two different character encodings, which may not even use the same character type in the worst case scenario.

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. Presumably the constant evaluation required for this approach could be implemented in a `consteval operator ""f()` function which would then use some future code injection mechanism to replace its call with a new token sequence or parsable string. To implement the different optimal approached for use within `std::print` like function and otherwise this `consteval` function would have to be able to access some kind of compiler context object which if can inquiry to see if it is inside a call to a function with specific properties or not. While such a context object would have other uses this seems like a feature that is very complicated and far into the future. To make this apply to any surrounding function similarly to how it is done in the current proposal would require being able to attempt overload resolution of the surrounding function.

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. This in turn would require the compile time execution engine to have access to the lexer, with the same potential character set and encoding problems noted for the language feature approach above.

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 lex 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 strategy

In R2 two tentative wording strategies were presented *Recursive lexing* and *Literal splitting*. The problem these strategies tried to solve is how to word the description of string interpolation in the phases of translation 3 to 6 (pp-lexing to string literal concatenation). The basic problem is that *while* lexing an f-literal each encounter of a `{` character starts a token sequence to be lexed… while already in the process of lexing the f-literal!

The first strategy, *Recursive lexing*, just restored the source pointer and recursed into the lexing function until a matching `}` pp-token was returned. This required saving up all tokens generated by the recursive calls and creating the output token stream constituting a complete `__format__` call. While similar intermediate storage of pp-tokens is required in phase 4 when expanding macros, this strategy would require a similar functionality in phase 3 too.

The *Literal splitting* strategy was envisioned to overcome this problem by emitting the f-literal part before the first `{` as a specially tagged string literal pp-token so that phase 3 could treat each string literal part between *expression-fields* as a separate pp-token, avoiding the temporary storage of token sequences. A drawback with this strategy was that the string literal tokens had to be tagged with *start, middle* and *end* information which was used by phase 6 to create the `__format__` call which was the result from phase 6.

In R3 a hybrid strategy is used, which avoids recursion in phase 3 as well as the need for tagging string literal parts.

20.1 Phase 3: Lexing

In this strategy phase 3 generates what looks syntactically as a call to a function with unspellable name (i.e. a function name that can't be written out in source code). This name is illustrated as *f-string-keyword* in this proposal. To stress that (in contrast with the `__format__` function call created in phase 6) this is not a real function call we use the term *f-string-construct* in this proposal and in the proposed wording.

A `f-string-construct` is a pp-token sequence describing the contents of the f-literal, with interleaved string literal tokens and expression field pp-tokens. When an f-literal is encountered, phase 3 generates an *f-string-keyword* identifier pp-token followed by a left parenthesis. When a `{` character is encountered in the f-literal a regular string literal pp-token is emitted with the contents of the f-literal so far, followed by a comma and a left parenthesis. Then lexing continues as usual until an unmatched `}` pp-token or a `:` token is encountered, at which point a right parenthesis and a comma are emitted instead of the `}` or `:` token and a new string literal pp-token is created, starting with the `}` or `:` character. Additional such groups of tokens are emitted for each `{` until the closing double quote of the f-literal is encountered, at which time the last string literal token and then the final right parenthesis to end the *f-string-construct* are emitted.

If the f-literal has an encoding prefix this encoding prefix is attached to the first string-literal that results from the lexing.

Some state must be kept by phase 3 although no token sequence needs to be stored and no recursion happens. This state is used to keep track of whether a `}` token is unmatched or part of the expression-field contents.

```
double pi = 3.14159265358979323846;
int ds = 2;
Lf"Pi with {ds} decimals is: {pi:.{ds}}"

// Creates a  f-string-construct:

f-string-keyword(L"Pi with{", (ds), "} decimals is: {", (pi), ":.{", (ds), "}}")
```

In the C++ standard wording each phase of translation is envisioned as running in sequence, producing its output in full before the next phase starts. With such an implementation the only additional state that phase 3 needs to keep relates to keeping track of the nested parentheses, brackets and braces in an expression field to be able to determine if a `}` or `:` finishes the *expression-field* or not. As a minimum phase 3 needs to contain a lifo stack of ints where each int represents the number of nested parentheses (of any kind) and the number of ints represents the number of active *f-literals* being lexed.

20.2 Phase 4: Preprocessing

In phase 4, preprocessing, the *f-string-construct* resulting from a f-literal is handled as any other pp-token sequence. As the construct's name is a keyword token there can't be a macro of this name to expand. However, any macros in the argument list of the *f-string-construct* are expanded as usual.

The lexing in phase 3 ensures that the contents of each expression field has matched parentheses, brackets and braces. This guarantees that in phase 4 the outer parenthesis added around each *expression-field* is matched. However, if expanding macros or replacing macro parameters causes an unmatched parentheses inside an expression field this can cause skipping over parenthesized expression fields in phase 5 and 6 to fail.

In the wording such errors could be IFNDR for simplicity but a quality compiler should detect that the *expression-field* is no longer enclosed in *matched* parentheses after phase 4 as error diagnostics could be very misleading otherwise.

20.2.0.1 The # operator

A special feature of the preprocessor is the prefix `#` operator applied to a macro parameter or `__VA_ARGS__`. This can be used to re-create the entire macro argument list as a quoted string. This happens in phase 4, which means that the pp-token sequence that the macro parameter(s) expand to can contain *f-string-constructs*. The

implementation of the # operator must handle `f-string-constructs` so as to re-create the original f-literal except that comments are removed and whitespace sequences are replaced by single spaces. To do this the *f-string-construct* pp-token sequence is processed by essentially removing the quotes of the string literal arguments except the first and last ones, removing the parentheses around each expression-field pp-token sequence and the commas separating arguments. To find out which right parenthesis ends an expression-field pp-token sequence requires counting matching parentheses.

20.3 Phase 5: literal encoding detection

Phase 5 must be able to detect which regular string literals and *f-string-constructs* are going to be concatenated and then deduce the appropriate character type for the entire literal. The encoding prefix of the first argument of each *f-string-construct* can be used when determining the encoding of the string literal sequence. As part of this process the rest of the *f-string-construct* must be skipped over after detecting its encoding prefix. This amounts to counting matching parentheses.

20.4 Phase 6: String concatenation

In phase 6 *f-string-constructs* are included as elements of string literal sequences to be concatenated. As a part of the concatenation process the string literal arguments of each *f-string-construct* are concatenated to each other. This involves passing over the intervening expression-field contents which requires counting matching parentheses.

If a string-literal concatenation sequence contains at least one *f-string-construct* the result from phase 6 is one `__format__` call with the concatenated string literals as its first argument and all the parenthesized expression fields of all *f-string-constructs* as further arguments. To perform this task phase 6 must maintain more state than today, as it collects more tokens before being able to emit the combined string literal as the first `__format__` argument. This is very similar to how macro expansion usually results in many tokens being stored and then fed one by one into phase 7. In the Clang implementation the same buffering mechanism that is used for macro expansion is reused for buffering the output of string concatenations involving f-literals.

Here is an example of how a source code line that concatenates f-literals and regular string-literals is transformed in the different phases of translation.

```
#include <format>

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

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

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

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

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

// 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.
```

Note:

At #1 the macro definition already contains the token sequence `f-string-keyword(" {",(name), "},")` as emitted by phase 3 when the `#define` directive is processed in phase 4. For function-like macros this means

that macro parameters can be replaced inside the expression-fields of f-literals present in the macro definition:

```
#define FORMAT_NAME(NAME) f"The name is {##NAME}"
FORMAT_NAME(Bjarne)

// After phase 3
#define FORMAT_NAME(NAME) f-string-keyword("The name is {", (##NAME), "}")
FORMAT_NAME(Bjarne)

// After phase 4
f-string-keyword("The name is {", ("Bjarne"), "}")

// AFter phase 6
__format__("The name is {}", ("Bjarne"))

// Program output
The name is Bjarne
```

20.5 Phase 7: Compilation

20.5.1 The new overload resolution step

This step is presented as a new paragraph which defines the new procedure of expanding the last `__format__` call to its arguments in certain circumstances. The idea of re-doing the overload resolution is described in prose.

21 Wording

This chapter presents diff towards N5014.

21.1 Changes in 5.2 [lex.phases]

Adjust the point list containing the overview of the phases of translation:

21.1.1 Adjust point 3

Add the new *f-char-sequence* and *raw-f-char-sequence* non-terminals to the list of non-terminals where *universal-character-names* are not replaced.

21.1.2 Change point 5 to

5. For a sequence of two or more adjacent *string-literal* preprocessing tokens and `f-string-constructs`, a common *encoding-prefix* is determined as specified in (5.13.5). Each such *string-literal* preprocessing token including the third pp-token of each `f-string--construct` is then considered to have that common encoding-prefix.

21.1.3 Change point 6 to

6. Adjacent *string-literal* preprocessing tokens and `f-string-constructs` are concatenated (5.13.5).

21.2 Changes in 5.5 [lex.pptoken]

Add the new keyword *f-string-keyword* to the list at the start of the clause, after *export-keyword*.

21.2.1 Paragraph 1

In the comma separated list of categories of preprocessing tokens add a new description directly before identifiers:

placeholder tokens produced by f-literals (*f-string-keyword*),

21.2.2 Add paragraph before 4

[Editor's note: This addition renumbers the subsequent paragraphs, but the paragraph numbers of N5014 are retained here for clarity. This version of the wording only shows one new paragraph, but it could be subdivided into more. There is no consistency even within [lex.string] as to whether notes and examples have their own paragraph numbers.]

4 The *f-string-keyword* is produced by processing a *f-literal* or *raw-f-literal*. This keyword starts a pp-token sequence called an *f-string-construct*. Each *f-string-construct* is the result of parsing one *f-literal* or *raw-f-literal*. These types of string literals can contain *expression-field*s enclosed in brace characters which consist of pp-tokens created by the lexing procedure of phase 3. To find which pp-token is last in an *expression-field* phase 3 has to count the net sum of left minus right parenthesis, bracket and brace pp-tokens generated for the *expression-field*. As a pp-token in an *expression-field* may itself be an *f-literal* or *raw-f-literal* phase 3 must retain a separate count for each *f-literal* nesting level.

21.2.2.0.1 ::: ednote

Maybe move Note 1 from after paragraph 3 or duplicate it for paragraph 4.

:::

21.2.3 Add paragraphs before 4.1

(4.1) — If the next character begins a sequence of characters that could be the prefix and initial double quote of a *f-literal*, such as `f"`, the next sequence of preprocessing tokens shall be a *f-string-construct* 5.13.5 [lex.string]. The *f-literal* is defined as the shortest sequence of characters that matches the f-string pattern

encoding-prefix$_{opt}$ `f` *f-string*

(4.2) — If the next character begins a sequence of characters that could be the prefix and initial double quote of a *raw-f-literal*, such as `fR"`, the next sequence of preprocessing tokens shall be a *f-string-construct* 5.13.5 [lex.string]. Inside the *raw-f-char-sequences* forming the string literal pp-tokens of the *f-string-construct*, any transformations performed in phase 2 (line splicing) are reverted; this reversion shall apply before any d-char, raw-f-char, or delimiting parenthesis is identified. The raw-f-string literal is defined as the shortest sequence of characters that matches the raw-f-string pattern

encoding-prefix$_{opt}$ `fR` *raw-f-string*

:::

[Editor's note: This wording has the same problem as the current wording for raw-string-literal in the next paragraph: To be able to undo the effects of phase 2 (line splicing) inside the literal you must be able to determine where it ends, but the wording indicates that the reversal is to be done *before* the *d-chars* that determine where the literal ends are identified.

This paradox indicates that in reality the reversal of the splicing has to be done *while* lexing the raw string literal, and this process has equal implementation complexity as doing the reversal of the splicing while lexing the raw-f-char-sequences of a raw-f-string.

According to Jens Maurer there is probably a CWG issue to clarify this.]

21.2.4 Change paragraph 4.1 (now 5.3) to start:

(4.3) — Otherwise, If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal.

21.3 Changes in 5.12 [lex.key]

Add the new keyword `f-string-keyword` to the list at the start of the clause.

21.4 Changes in 5.13.5 [lex.string]

21.4.1 Modify the *string-literal* non-terminal

Add two lines covering f-literals:

string-literal:
encoding-prefix$_{opt}$ " *s-char-sequence*$_{opt}$ "
encoding-prefix$_{opt}$ `R` *raw-string*
encoding-prefix$_{opt}$ `f` *f-string*
encoding-prefix$_{opt}$ `fR` *raw-f-string*

21.4.2 After the non-terminal for *d-char*

Add the following new non-terminals:

f-char:
basic-f-char
escape-sequence
universal-character-name

basic-f-char:
any member of the translation character set except the U+0022 QUOTATION MARK, U+005C REVERSE SOLIDUS, U+007B LEFT CURLY BRACKET, U+007D RIGHT CURLY BRACKET or new-line character

f-char-sequence:
f-char f-char-sequence$_{opt}$
`{{` *f-char-sequence*$_{opt}$
`}}` *f-char-sequence*$_{opt}$

format-specifier:
f-char-sequence
f-char-sequence$_{opt}$ `{` *expression-field* `}` *format-specifier*$_{opt}$

extraction-field:
expression-field
expression-field `:` *format-specifier*$_{opt}$

f-string-contents:
f-char-sequence
f-char-sequence$_{opt}$ `{` *extraction-field* `}` *f-string-contents*$_{opt}$

f-string:
" *f-string-contents*$_{opt}$ "

raw-f-char:
any member of the translation character set except U+007B LEFT CURLY BRACKET, U+007D RIGHT CURLY BRACKET or a u+0029 RIGHT PARENTHESIS followed by the initial *d-char-sequence* (which may be empty) followed by a U+0022 QUOTATION MARK

raw-f-char-sequence:
raw-f-char raw-f-char-sequence$_{opt}$
`{{` *raw-f-char-sequence*$_{opt}$
`}}` *raw-f-char-sequence*$_{opt}$

raw-format-specifier:
    raw-f-char-sequence
    raw-f-char-sequence$_{opt}$ `{` *expression-field* `}` *raw-format-specifier*$_{opt}$

raw-extraction-field:
    expression-field
    expression-field `:` *raw-format-specifier*$_{opt}$

raw-f-string-contents:
    raw-f-char-sequence
    raw-f-char-sequence$_{opt}$ `{` *raw-extraction-field* `}` *raw-f-string-contents*$_{opt}$

raw-f-string:
    `"` *d-char-sequence*$_{opt}$ `(` *raw-f-string-contents*$_{opt}$ `)` *d-char-sequence*$_{opt}$ `"`

21.4.3 After paragraph 5

Add new paragraphs:

[Editor's note: This addition renumbers the subsequent paragraphs, but the paragraph numbers of N5014 are retained here for clarity. This version of the wording only shows one new paragraph, but it could be subdivided into more. There is no consistency even within [lex.string] as to whether notes and examples have their own paragraph numbers.]

6 A *string-literal* that has a `f` in the prefix is a *f-string* or a *raw-f-string*. Lexing a *f-string* or *raw-f-string* results in a pp-token sequence called a *f-string-construct*. A *f-string-construct* starts with the *f-string-keyword* followed by a left parenthesis pp-token. The remainder of the *f-string-construct* is a pp-token sequence formed by lexing according to the *f-string-contents* or *raw-f-string-contents* grammar followed by a right parenthesis pp-token.

(6.1) Lexing according to the *f-string-contents* or *raw-f-string-contents* grammar results in one string-literal pp-token created from each consecutive character sequence outside of *expression-field*s and pp-token sequences created by regular lexing of the characters constituting the *expression-field*s, in lexical order. Each pp-token sequence created by lexing an *expression-field* is prefixed by a comma pp-token and a left parenthesis pp-token and suffixed by a right parenthesis pp-token and a comma pp-token.

(6.2) While lexing the input stream to generate the pp-token sequence of an expression-field, pp-tokens representing nested pairs of parenthesis, bracket and braces are matched and if a right brace or colon character is the next character in the input stream when all such pairs are matched the *expression-field* has ended.

(6.3) Matching of nested pairs of parenthesis, bracket and braces is the process of keeping track of the number of left parenthesis, left bracket and left brace pp-tokens generated minus the number of right parenthesis, right bracket and right brace pp-tokens generated. Whenever this sum is 0 all pairs are matched. If this sum is negative or if parenthesis, bracket and brace pairs are not properly nested the program is ill-formed.

(6.4) Parenthesis, bracket and brace pairs are properly nested if for each right parenthesis, bracket and brace pp-token **R** generated, the closest previous left parenthesis, bracket or brace pp-token **L** that has not already been matched to a right parenthesis, bracket and brace pp-token is of matching kind: If **R** is a right parenthesis pp-token then **L** is a left parenthesis pp-token, if **R** is a right bracket pp-token then **L** is a left bracket pp-token and if **R** is a right brace pp-token then **L** is left brace pp-token.

(6.5) The first *string-literal* pp-token in an *f-string-construct* has the same *encoding-prefix* as the lexed *f-string* or *raw-f-string*, if any.

[*Note:* The *f-string-keyword* has no observable spelling. — *end note*]

[*Example:* The following f-literal:

```
Lf"B {a, b} 2C {c * 2:3.{d}}\n"
```

results in the following pp-token sequence:

```
f-string-keyword(L"B {", (a, b), "} 2C {", (c * 2),
                         ":3.{", (d), "}\n")
```

— *end example*]

[*Note:* When lexing the source code of an *expression-field* phase 3 can emit generated pp-tokens without buffering. The matching of parentheses can be implemented using a expression field state consisting of a count or a stack keeping track of nested parentheses of different kinds which is updated during lexing of the characters of the *expression-field*. — *end note*]

[*Note:* An *expression-field* can contain nested *f-string-literals* or *raw-f-string-literals*. Existing expression field state must be saved before lexing the nested *f-string-literal* or *raw-f-string-literal* and restored afterwards. — *end note*]

[*Note:* In an implementation where a function is called to get the next pp-token internal state can be used to keep track of which of the fixed tokens of a *f-string-construct* to return next. This means that sometimes this function will not consume any input when called, insted it just updates the internal state and returns a fixed token. — *end note*]

[*Note:* The reason for emitting an *f-string-construct* token-sequence resembling a function call from phase 3 is to be able to find the end of the *f-string-construct* without introducing any new pp-token types. Finding the end of a *f-string-construct* is needed by stringization 15.7.3 [cpp.stringize], phase 5 (determining encoding prefix) and phase 6 (string concatenation). — *end note*]

21.4.4 Change paragraph 7 to

The *string-literals* in any sequence of adjacent *string-literals* and the first *string-literal* pp-token of all adjacent *f-string-constructs* shall have at most one unique encoding-prefix among them. The common encoding-prefix of the sequence is that encoding-prefix, if any.

::: note 3

A string-literal's rawness has no effect on the determination of the common encoding-prefix.

:::

21.4.5 In paragraph 8

Change the first sentence to:

In translation phase 6 (5.2), adjacent string-literals and *f-string-constructs* are concatenated.

21.4.5.1 At the end of paragraph 8 add

(8.1) When one or more *f-string-constructs* are part of the sequence being concatenated, the concatenation results in a sequence of pp-tokens constituting a call to a `__format__` function. The first argument of this call is a string literal object resulting from the concatenation of all *string-literals* including the *string-literal* pp-tokens of all *f-string-constructs* not part of any *expression-field* pp-token sequence. Each succeeding argument of the `__format__` function call is formed from the pp-token sequence of the next *expression-field* in the concatenated *f-string-constructs* including its enclosing parentheses.

(8.2) All `{` and `}` characters in *string-literals* that are not from *f-string-constructs* are doubled when concatenated with *string-literals* from *f-string-constructs*.

(8.3) Each *f-string-construct* which has no adjacent *string-literals* or *f-string-constructs* must be converted into a `__format__` function call using the procedure described in 8.1.

[*Example:* Two f-literals are concatenated with one regular string-literal in between them.

```
f"One {2} " "{three}" f" {4}"
```

After phase 3:

```
f-string-keyword("One {", (2), "} ") "{three}" f-string-keyword(" {", (4), "}")
```

After phase 6:

```
__format__("One {} {{three}} {}", (2), (4))
```

Resulting string:

```
One 2 {three} 4
```

— *end example*]

[*Note:* To find out which pp-tokens of a *f-string-construct* are considered *string-literals* and which are part of *expression-field* pp-token sequences the *f-string-construct* pp-token sequence's comma pp-tokens outside of nested parenthesis, bracket and brace pairs can be numbered. The *string-literal*s to concatenate are the last pp-tokens before odd numbered comma pp-tokens plus the next to last pp-token of the *f-string-construct*, while *expression-field* pp-token sequences are all tokens between an odd numbered comma pp-token and the next even numbered comma pp-token. — *end note*]

[*Note:* As the pp-token sequences resulting from the *expression-fields* of an outer *f-string* can contain *string-literals* that can be potentially be concatenated the data structure used to reorganize the pp-token sequence into a `__format__` call must be saved before the concatenation of the inner *string-literals* is performed, and restored afterwards. — *end note*]

21.5 In [over.match.general]

Add a new paragraph after paragraph 2.8 briefly describing the process of redoing the choosing of viable functions if needed [over.match.format]:

(2.9) If there is a `__format__` call in the argument list and the set of viable functions is empty or the conversion sequence from the last `__format__` call contains a constructor or conversion function declared `consteval` for all viable functions the `__format__` function call is replaced by its arguments (12.2.4) and the determination of the set of viable functions is redone.

(2.10) Then the best viable function is selected based on the implicit conversion sequences (12.2.45.2) needed to match each argument to the corresponding parameter of each viable function.

21.6 After 12.2.3 [over.match.viable]

Add a new level three clause **[over.match.format]** as 12.2.4, bumping **[over.match.best]** to 12.2.5

1 If the argument list of a function call contains at least one argument consisting of a call to a global scope function named `__format__` and the set of viable functions is empty or consists only of functions where the implicit conversion sequence from the return type of the last `__format__` call in the argument list to the corresponding parameter type contains a call to a constructor declared `consteval` or call to a user defined conversion function declared `consteval` this `__format__` call is replaced by its arguments and then 12.2.3 [over.match.viable] is performed again, creating a new set of viable functions. The set of viable functions is then passed on to 12.2.4 [over.match.best].

[*Note:* The reason for precluding `consteval` calls in the conversion sequence is that the standard library `__format__` function is declared `constexpr`, which means that it may or may not be callable in a constant evaluation context depending on its arguments. Allowing `consteval` calls would cause errors in later stages of the compilation if the arguments to `__format__` turns out to not be possible to evaluate at compile time. — *end note*]

21.7 In 15.7.3 [cpp.stringize]

Add a paragraph describing how a *f-string-construct* is handled during stringization by the prefix # operator.

::: add

[3] Each occurrence of a *f-string-construct* in the *stringizing argument* is replaced by the original f-literal prefix followed by a string literal containing a character sequence formed by concatenating the *string-literal* pp-tokens not part of any *expression-field* pp-token sequence after special handling according to paragraph 2 and removal of the enclosing double quote characters with the original spelling of the pp-tokens of the pp-token sequences of the next *expression-field*s in the *f-string-construct* excluding their enclosing parentheses. Whitespace is handled as for all other pp-tokens being stringized.

[*Note:* This procedure restores the original spelling of the *f-literal* except that sequences of whitespace between the preprocessing tokens of each *expression-field* becomes a single space character, including before the first and after the last preprocessing token of each *expression-field*. — *end note*]

[Editor's note: An alternative which is maybe easier to implement is to just refer to the original spelling of the *entire* f-literal. In practice this means that the pp-token for the *f-string-keyword* would refer to the original source code character sequence. This is probably simpler to implement but would retain comments and newlines inside the expression-fields, which is not consistent with the removal of comments and repeated whitespace that occurs outside of f-literals.

This is a fringe use case so it is even possible to make it implementation defined whether comments and newlines are removed inside *expression-fields*.]

21.8 In 28.5.1 [format.syn]

After the closing brace of the namespace `std` add function declarations:

```
// 28.5.11 forwarding formatting functions
template<class... Args>
  string __format__(format_string<Args...> fmt, Args&&... args);
template<class... Args>
  wstring __format__(wformat_string<Args...> fmt, Args&&... args);
```

21.9 After 28.5.10 [format.error]

Add a new level three clause 28.5.11 [format.forwarding] describing the `__format__` functions.

```
template<class... Args>
  string __format__(format_string<Args...> fmt, Args&&... args);
```

[1] *Effects:* Equivalent to:

```
return format(fmt, forward<Args>(args)...);
```

```
template<class... Args>
  wstring __format__(wformat_string<Args...> fmt, Args&&... args);
```

[2] *Effects:* Equivalent to:

```
return format(fmt, forward<Args>(args)...);
```

21.10 In Index of library names

Add the name `__format__` to the list of library names.

22 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.

Thanks to Jens Maurer for initial code review before publication of R3.

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

23 References

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

[P3298R1] Bengt Gustafsson. 2024-10-15. Implicit user-defined conversion functions as operator.().
https://wg21.link/p3298r1

[P3391R1] Barry Revzin. 2025-04-15. constexpr std::format.
https://wg21.link/p3391r1

[P3398R0] Bengt Gustafsson. 2024-09-17. User specified type decay.
https://wg21.link/p3398r0