

String interpolation

Document #: P3412R4
Date: 2026-10-23
Project: Programming Language C++
Audience: SG16, EWG
Reply-to: Bengt Gustafsson
<bengt.gustafsson@beamways.com>
Victor Zverovich
<victor.zverovich@gmail.com>

Contents

1	Abstract	3
2	Examples	3
3	Motivation	4
3.1	Motivation for automatic conversion to strings	5
4	Revision history	5
4.1	R0, Presented to EWGI in Wroclaw	5
4.2	R1, Presented to EWGI in Hagenberg	5
4.3	R2, After presentation to SG16 and SG22	5
4.4	R3, Discussed together with P3951 in London	6
4.5	R4, This revision	6
5	Terminology	6
6	Lexing a f-literal (phase 3)	7
6.1	Expression-field contents	8
6.2	Detecting where an expression field ends	8
6.2.1	Handling format specifiers starting with : or]	8
6.3	No preprocessor directives in expression-fields	9
6.4	__Pragma in expression-fields	9
6.5	Error handling	9
6.6	Nested expression-fields	10
6.7	Encoding- and raw literal prefixes	10
6.8	Open questions regarding lexing of f-literals	10
6.8.1	Does a colon outside balanced parentheses always start a format-specifier?	10
6.8.2	Does the digraph <: start a replacement-field?	11
6.8.3	Does the digraph :> end a replacement field?	11
6.8.4	Is : a good character to start a format-specifier?	11
6.8.5	Are raw f-literals special?	11
7	Preprocessing (phase 4)	12
7.1	Preprocessing to a file	12
8	String literal concatenation (phase 5)	13
8.1	Concatenating all the string literal parts	13
8.2	Quoting of regular literal contents during concatenation	13

9	Conversion to a token sequence (phase 6)	14
10	Parsing and overload resolution (phase 7)	14
10.1	Modifying concepts	15
10.1.1	1. Expansion of argument expressions	15
10.1.2	2. An alternative to P3951 objects	16
10.1.3	3. Expansion without coalescing literal parts.	19
10.1.4	4. f-literal objects without the replacement_field level.	19
10.2	Perfect forwarding of f-string-constructs	19
11	The __format__ function	19
11.1	Naming	20
11.1.1	An alternative spelling	20
11.2	Location of the __format__ function	20
11.3	The standard implementation of __format__	21
11.4	Alternative __format__ functions	21
11.5	Source code __format__ calls	22
12	User-defined literal suffixes	22
13	Contexts where string interpolation works	22
14	Code breakage risk	22
15	Modules related aspects	23
16	Implementation in other tools	23
17	Implementation experience	23
17.1	Stand alone implementation	23
17.2	Clang implementation	24
17.2.1	Lessons learned	25
17.3	Vendor feedback about implementability	25
17.3.1	Clang	25
17.3.2	EDG	25
17.3.3	GCC	25
17.3.4	MSVC	25
18	Alternatives	25
18.1	Language feature	25
18.2	Reflection	26
19	History	26
20	Wording strategy	27
20.1	Phase 3: Lexing	27
20.2	Phase 4: Preprocessing	28
20.2.1	The # operator	28
20.3	Phase 5: Encoding detection and string concatenation	29
20.4	Phase 7: Compilation	30
20.4.1	The new overload resolution step	30
21	Wording	30
21.1	Changes in 5.2 [lex.phases]	30
21.1.1	Adjust point 3	30
21.1.2	Change point 5 to	30

21.2	Changes in 5.5 [lex.pptoken]	30
21.2.1	Paragraph 1	30
21.2.2	Add paragraph before 5	31
21.2.3	Add paragraphs before 5.1	31
21.2.4	Change paragraph 5.1 (now 6.3) to start:	31
21.3	Changes in 5.12 [lex.key]	31
21.4	Changes in 5.13.5 [lex.string]	31
21.4.1	Modify the <i>string-literal</i> non-terminal	31
21.4.2	After the non-terminal for <i>d-char</i>	32
21.4.3	After paragraph 5	33
21.4.4	Change paragraph 7 to	34
21.4.5	In paragraph 8	34
21.5	In [over.match.general]	35
21.6	After 12.2.2 [over.match.funcs]	35
21.6.1	12.2.3 Modifying <i>f-string-construct</i> arguments	35
21.7	In 15.7.3 [cpp.stringize]	36
21.8	In 28.5.1 [format.syn]	36
21.9	After 28.5.10 [format.error]	37
21.10	In Index of library names	37
21.11	... somewhere ...	37
21.12	Modifying concepts	37
21.12.1	Expanding argument expressions	37
21.12.2	Creating a fliteral_object	37
22	Acknowledgements	37
23	References	38

1 Abstract

This proposal adds string interpolation (so called f-literals) to the C++ language. After phase 5 each f-literal has been transformed to a token sequence called a *f-string-construct*. By default this construct is *modified* to a call to a function tentatively named `__format__` with a function argument list compatible with `std::format`. In addition this proposal introduces a way to specify that one of a few alternative modifications to the *f-string-construct* is to be done. These can be used to implement functions like `std::print` that need the same argument expressions as `std::format` and optionally to create objects of a new `std::fliteral_object` class template with similar functionality as the synthesized types of [P3951R2]. These alternate modifications can be selected by constraining parameter types with special built-in *modifying concepts*.

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`. This `__format__` function is declared in an inline namespace to allow other libraries to provide other `__format__` functions without ODR violations or unexpected behavior. To use the standard `__format__` function a `using namespace std::fliterals` directive can be issued or, for convenience, the new `<fliterals>` header can be included.

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 8000 times. This proposal has also been implemented as a separate program, which demonstrates the viability of this proposal for tools like syntax coloring editors.

2 Examples

Assuming a `Point` class which has a formatter we can use string interpolation to format Points. 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.

```
#include <fliterals> // Use the standard implementation of __format__

Point getCenter();

std::string a = f"Center: {getCenter()}"; // a is initialized from a std::format call.

auto b = f"Center: {getCenter()}"; // b is a std::string.

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

std::println(f"Center: {getCenter()}"); // Invokes a new format_string constructor.

std::cout << f"Center: {getCenter()}"; // Suboptimal: A temporary std::string is created.
```

In all cases except the `println` call the *f-string-construct* is modified to a `__format__` function call which forwards to `std::format` formatting to a `std::string`. In the `println` example however, the compiler notices after collecting the candidate functions that one of them has a *modifying concept* matched to the *f-string-construct* resulting from the f-literal in the source code. This causes the associated modification of the *f-string-construct* which in this case removes the *f-string-construct* to reveal the literal and argument expressions expected by `std::println`. The details of this process are described below.

3 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. That Python linters such as `ruff` automatically convert older formatting styles `"Literal" % (args)` and `"literal".format(args)` to f-literals indicate that f-literals are nowadays seen as the preferred syntax.

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}"
```

3.1 Motivation for automatic conversion to strings

In contrast with P3951 this proposal includes a way that f-literals are converted to some kind of string by calling a `__format__` function under certain conditions. The motivation for this is mainly that it is what users expect and that it is convenient.

An additional motivation is that automatic conversion makes implementation of new overloads of “formatting style” functions such as logging functions and exception constructors unnecessary. Case in point is [P4364R0] which proposes a `std::print` style API for compile time message/warning/error generation that would be unnecessary if f-literals were automatically converted to strings. It seems like something of an oxymoron to write [P4364R0] to avoid having to type `std::format` when formatting an error message while in [P3951R2] arguing that typing `std::format` around f-literals in general is not an inconvenience worth trying to avoid.

Note that the reason for not *always* converting a f-literal to a string is to not lose the performance gain that `std::print` has over printing a string produced by `std::format`. This type of performance sensitivity is not present for compile-time error text generation and usually not for logging. And in all the cases where the end result is a string anyway there is no performance gain to be had by enclosing the f-literal in `std::format` as this is what the f-literal results in anyway (especially if we get [P2826R4] Expression Aliases).

4 Revision history

4.1 R0, Presented to EWGI in Wroclaw

First revision.

4.2 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.

4.3 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.

4.4 R3, Discussed together with P3951 in London

- 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 `constexpr` 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.

4.5 R4, This revision

- Changed phase numbering to reflect the changes in a more recent standard draft [N5054].
- Changed terminology from *extraction field* to *replacement field* to match [format.string].
- Add `>` to the set of pp-tokens that ends an expression-field.
- Add a rule the first space after a colon that ends an expression field is removed if the first non-space character is `:` or `]`.
- Added open lexing questions section.
- Added section on what the preprocessed result of a f-literal should be.
- Use a new system for modifying preprocessed *f-string-constructs* depending on which *modifying-concepts* constrain parameters.
- Placed `__format__` in a nested namespace to avoid linker symbol clashes.
- Added section on how code using different declarations of `__format__` can coexist.
- Removed all references to printf formatting. While this is still a possibility it didn't seem to appeal to WG14 representatives in the SG22 group presentation but can of course be found in the previous revisions.

5 Terminology

The different parts of an f-literal have specific names to avoid confusion. This is best illustrated by an example, see below. In R4 the terminology has been harmonized with [format.string], while the term *expression field* is retained with the meaning of one of the expressions of a *replacement field*.

```

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

f"The result is { get_result() :{width}.3}"
//      ~~~~~
//      |
//      replacement-field

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

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

```

Before the f-literal is passed along to the rest of the compiler it is converted to a *f-string-construct* which resembles a function call with the parts of the f-literal subdivided into interleaved string literals and expression fields. Instead of a function name *f-string-constructs* have a special preprocessor keyword **f-string-keyword**.

6 Lexing a f-literal (phase 3)

When the lexer (phase 3) sees the start of a f-literal it converts the f-literal to a *f-string-construct* which is treated as a call to a function with an unspellable name represented by a special pp-token keyword we can write as **f-string-keyword**. A complete f-literal is transformed to a *f-string-construct* by scanning the contents of the f-literal and extracting the replacement-fields as sequences of pp-tokens by lexing as usual. The resulting pp-tokens are interspersed between the literal parts which are treated as separate string literal pp-tokens. Each replacement-field pp-token sequence is enclosed in parenthesis pp-tokens and the literal parts and replacement-fields are separated by commas. Here is an example:

```

// Source code
f"Value of {a} times {b} is {a * b:>5}"

// After lexing. This string represents a pp-token sequence where
// each pp-token is separated by a space:
f-string-keyword ( "Value of {}", ( a ) , "}" times {",", ( b ) , "}" is {",", ( a * b ) , " :>5}" )

```

Note that the lexer does not have to buffer any pp-tokens, they are emitted as they are formed, with some extra (, , and) pp-tokens inserted to make sure that the syntax of a function call is retained.

A consequence of lexing the contents of an *expression-field* as usual 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.

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 or 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.

6.1 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 each extracted expression is enclosed in parentheses. This prevents an extracted expression from being interpreted as multiple arguments in the resulting *f-string-construct*. Allowing a full *expression* instead of only an *assignment-expression* as in a regular function argument is important to avoid errors due to commas in template argument lists, which can't be differentiated from other commas by the lexer.

This problem is illustrated by the following examples:

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

// Lexed to:
f-string-keyword("Weird, but OK: {", (1 < 2, 2 > 1), "}")           // Comma operator!

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

// Lexed to:
f-string-keyword("Reversed: {", (std::set<int, std::greater<>>(values, values + 5), "}")
```

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.

6.2 Detecting where an expression field ends

Detecting the end of an *expression-field* 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 even contain an 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, square bracket pairs and splicer pairs. This rule 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:

```
// Not allowed:
f"The answer is {boolVar ? 17 : 42}";

// As it is lexed to the erroneous:
f-string-keyword("The answer is {", (boolVar ? 17), ":", 42});
```

Note: If [\[P4235R0\]](#) *Removing digraphs* is adopted there will be no `:>` tokens in expression fields.

6.2.1 Handling format specifiers starting with `:` or `]`

As `::` is not among the pp-tokens ending an expression-field double `::` is always treated as the scope resolution operator in this proposal. Likewise, a `:]` pp-token is always treated as the end of a splice. This prevents format specifiers starting with `:` or `]` from being used. Such format specifiers are fairly rare but do exist: A fill character of `:` or `]` and an element format specifier when formatting a range are obvious examples.

To cater for such format specifiers it is proposed that a space first in a *format specifier* is excluded from the succeeding string literal if the next non-space character is a `:` or `]`. This allows a fairly intuitive work-around:

Just add a space after the first colon to avoid it being treated as part of a longer pp-token. The reason for only removing a space if followed by one `:` or `]` is to avoid having to double a leading space which indicates that a positive value is to be formatted with a leading space (matching the `-` of a negative value). The reason to accept additional spaces before a `:` or `]` is to cater for the off-chance that a user defined format specifier starts with a number of spaces followed by `:` or `]`.

```
std::vector<float> v;
f"Floats: {v::>10.3f}" // Try to format each float with three decimals in a 10 wide field.

// This is what you get as :: is treated as a pp-token not ending an expression field
f-string-keyword("Floats: {", (v::>10.3f), ")")

// To fix this add a space to avoid lexing to a :: pp-token:
f"Floats: {v: >10.3f}" // Format each float with three decimals in a 10 wide field.

// The output has the space removed:
f-string-keyword("Floats: {", (v), " >10.3f}")
```

P3951R2 proposes another mechanism where when parenthesizing the expression a succeeding `::` pp-token is treated as the start of a format specifier except if the parenthesis is preceded by the keyword `decltype`. We find this idea less intuitive and it seems somewhat harder to implement due to the special treatment of `decltype`.

6.3 No 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.

6.4 `_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..

6.5 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 a replacement-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.

6.6 Nested expression-fields

Nested expression-fields inside the format-specifier of a replacement-field are always extracted regardless of if the formatter for the data type handles dynamic values 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 just 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.

[P3951R2] has a special mechanism to avoid that the contents of a brace pair inside a format specifier is treated as an expression: Place a colon first after the left brace. This has some level of logic for the example use case: Element formatting for tuples, which can be done in `fmt` by writing nested format strings for each tuple element in individual braces. But for other uses of braces the colon notation does not make sense so this proposal does not include that feature, but relies on doubling braces as the only mechanism.

6.7 Encoding- and raw literal prefixes

The `f` prefix can be combined with the `R` and `L` prefixes. Theoretically it can also be combined 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.

6.8 Open questions regarding lexing of f-literals

There are a number of issues that need to be resolved in SG16 and confirmed by EWG. Those are listed below with the proposed resolutions of this proposal, [P3951R2] and including opinions voiced by CWG representatives who want to explore the design space without taking previous `std::format` format string syntax into account.

6.8.1 Does a colon outside balanced parentheses always start a format-specifier?

According to this proposal replacement-field end is determined by inspecting the pp-token stream. This means that a `::` pp-token *does not* start a format specifier. P3951R1 instead inspected the character stream and thus starts a formatted specifier at the first colon of a `::` pp-token yet to be formed. In P3951R2 `::` is treated as a scope resolution operator as in this proposal.

It would be possible to also prevent the `:` of a *conditional-expression* from ending an expression field, it is just a matter of matching `?` characters with `:` characters outside of nested parentheses. As there are some ideas about for instance prefix `?` operators we don't include this feature for now. It was implemented in the Clang fork for this proposal available on Compiler Explorer though.

6.8.2 Does the digraph <: start a replacement-field?

Neither this proposal nor P3951 treat <: as the start of a replacement field despite this character combination being a digraph for {. CWG representatives have argued that <: should start a replacement field regardless of the fact that std::format does not. Out arguments for not letting <: start a replacement-field include:

- When detecting the { that starts a replacement-field we are scanning characters, not forming pp-tokens, and <: is a pp-token, not a character.
- Few programmers know of digraphs, and fewer still use them.
- Digraphs today exist to be able to compile old source code, but old source code does not contain f-literals. Few if any programmers today work on systems with character sets lacking the { character.
- If <: starts a replacement-field, how do we get this character combination into a f-literal? The only option seems to be to use string literal concatenation between regular and f-literals. The surprise effect of not being able to get these two characters into a f-literal is considerable, especially considering how few programmers know of digraphs.

Note: If [P4235R0] Removing digraphs is adopted interpreting <: as the start of an expression field is not an issue.

6.8.3 Does the digraph :> end a replacement field?

This proposal treats the :> pp-token as ending a replacement field along with the : and } pp-tokens. In all cases the characters of these tokens are treated as the start of the format specifier or end of the replacement-field.

This is more controversial as we are now treating the alternative token :> differently from the pp-token it is an alternative of. We feel that this is a small price to pay to get this relatively common start of a format specifier to work as expected. A more advanced option is to treat :> as a right bracket as long as there is a corresponding left bracket (or <: token) to match.

Note: If [P4235R0] Removing digraphs is adopted there will be no :> tokens in expression fields.

6.8.4 Is : a good character to start a format-specifier?

It is agreed that starting a format specifier with a : is not ideal, as this character have valid uses inside an expression-field as exemplified above. CWG representatives floated the idea to use ; instead as it can't be part of an expression. This proposal maintains that : is the right choice both for its legacy use in std::format and additionally as using ; would make error recovery during compilation harder. Error recovery is easier to implement if skipping tokens outside of balanced parentheses until a ; is seen finds a good point to re-start compilation after an error. With ; occurring in replacement-fields with possible syntax errors the compiler can no longer be certain that the next semicolon ends the statement.

With the rules around : characters and pp-tokens starting with : characters in this version of the proposal gives the issues with using : to start format specifiers satisfactory solutions, so we think that keeping the : as the format specifier start is a good call.

6.8.5 Are raw f-literals special?

In this proposal as well as in P3951 raw f-literals don't treat the { and } characters differently than regular f-literals. Some CWG representatives have argued that because the " ending a raw literal has to be preceded by its *d-char-sequence* a { character should not be treated as the start of a replacement-field unless it is preceded by the same d-char-sequence. This follows the logic that any character in a raw string literal represents itself except when preceded by the d-char-sequence. However, this prevents using an empty d-char-sequence as there would be no way to start a replacement field, whereas today an empty d-char-sequence is most often fine as the character sequence)" is very seldom needed in a raw string literal. This suggestion would also take f-literals further from the syntax of raw format strings for std::format where escaping of braces is still done by doubling. As in most cases the number of replacement fields exceeds the number of braces in the strings retaining the original way to escape braces reduces the effort to get the spelling right.

An exception may be when writing code generators producing code in C-like languages which have quite a lot of braces. One such code generator an author maintains, which generates C++ code using `fmt::format` and raw literals, had 92 replacement-fields and 35 each of literal left and right braces. So even when generating C++ code it may not be better to simplify writing literal braces at the expense of replacement-field starting braces.

7 Preprocessing (phase 4)

The functionality of phase 4 is unchanged and treats each *f-literal-construct* as a pp-token sequence resembling a function call. As the `f-string-keyword` is not possible to spell it can never denote a macro to be expanded, but any macros inside the pp-token sequences of the expression-fields (including nested expression-fields) are expanded.

7.1 Preprocessing to a file

One valid way to do C++ compilation is to first pre-process to a file and then pass this file to the compiler. In reality the second of these compilations passes the already preprocessed file through the first four phases of translation again although phase 4 will have nothing to do as there are no `#include` or `#define` directives to process and no macro expansions to do.

To make this work with f-literals the `f` prefix must be retained in the preprocessed output as it has special meaning in phase 5 and later. This means that the contents of the expression fields after macro expansion must be replaced back into the brace pairs of the surrounding string literals inside the *f-string-construct*. This is basically what the `stringize` operator does, but with the addition to remove the extra parenthesis pairs and commas between the pp-token sequences for each expression-field. Example:

```
// Example: glibc definition of errno
#define errno (*__errno_location())

// Original source
f"The error code is {errno}"

// After phase 4 when preprocessing to file.
f-string-keyword("The error code is {", ((*__errno_location()), "}");

// Preprocessed output file contents.
f"The error code is {(*__errno_location())}"

// After phase 4 when actually compiling. (Same as above.)
f-string-keyword("The error code is {", ((*__errno_location()), "}");
```

This causes a problem if the source code for expression-fields is to be made available to users via some kind of compile time functionality. [P3951R2] recognizes this and introduces a syntax where the preprocessed output contains the source code both before and after preprocessing, separated by a `;` character. We feel that this complexity is not warranted for a very peripheral feature, especially considering that the source code before preprocessing can contain newlines, comments and nested string literals (that can be raw) complicating the generation of the string literal containing the original source code that P3951 puts after the `;`.

Making the *preprocessed* source code available to users would be much easier and could be a good compromise. This can be motivated by saying that *you get the source code of this compilation*, that there was some previous source code given to some preprocessor at some earlier time is not considered. This may or not be what users wish for, but it is a reasonable definition. Note that the `source_text_of` function of [P4380R0] Token Sequence Injection has a worse problem with preserving pre-macro expansion source text when doing separate preprocessing. I don't see how this can be fixed, so for consistency we should go with the *source code of this compilation* model for f-literals also.

We would happily re-introduce the Python debug feature included in P3951 which was included in R0 of this

proposal but later dropped due to wording problems related to keeping the phases of translation separated. For this feature the preprocessed output file naturally has the source code before preprocessing preserved before the equals sign:

```
// Example: glibc definition of errno
#define errno (*__errno_location())

// Original source
f"{errno=}"

// After phase 4 when preprocessing to file.
f-string-keyword("errno={", ((*__errno_location()), "}");

// Possible resulting string
"errno=11"
```

Note that preserving the format specifier of a replacement-field requires adding a space after the leading colon if one was removed in phase 3. Whether this is required can easily be detected during creation of the preprocessed output without extra information provided from phase 3.

8 String literal concatenation (phase 5)

f-literals can be concatenated to 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 *f-string-construct* containing the literals and the extracted expression fields of all f-literals being concatenated. Regular literals preceding a f-literal are concatenated to the first string-literal of the succeeding *f-string-construct* and regular literals succeeding any *f-string-construct* of the sequence are concatenated to the last string-literal of the preceding *f-string-construct*.

```
// Source code (using control codes to get a bold output)
"\033[1m" f"Important error: {errno}" "\033[0m"

// After phase 3
"\033[1m" f-string-keyword("Important error: {", (errno), "}") "\033[0m"

// After phase 4 (with glibc errno)
"\033[1m" f-string-keyword("Important error: {", ((*__errno_location()), "}") "\033[0m"

// After phase 5
f-string-keyword("\033[1mImportant error: {", ((*__errno_location()), "}") "\033[0m")
```

8.1 Concatenating all the string literal parts

At some point during compilation all the string literals that are embedded in the concatenated *f-string-construct* as the odd numbered arguments may have to be concatenated and placed as one first argument. Such concatenation has to be done if a `__format__` function call is generated and before expanding argument expressions, but with other modifying concepts (see below) it may not.

8.2 Quoting of regular literal contents during concatenation

A problem arises when concatenating f-literals with regular string literals containing `{` or `}` characters. To prevent these characters from being treated as the start or end of an expression field after string concatenation they are doubled in phase 5 when concatenated with f-literals.

```

// Source code
int x = 15;
int y = 65;

"Start point { " f"{x}, {y}" " }"

// The phase 5 output
f-string-keyword("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. This is suggested by P3951. The problem with that 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 traditionally used to send control sequences to terminals. Also the resulting error output could be very confusing as the string literal parsing of `std::basic_format_string` would get a very strange string. For the above case:

```
"Start point { {}, {} }"
```

The first brace will send the format string parsing out on a hike and as the programmer does not see this string it can be hard to understand what is going on.

9 Conversion to a token sequence (phase 6)

In phase 6 each *f-string-construct* is preserved, converting each pp-token it contains to a token for phase 7 to parse, including the `f-string-keyword` which is converted to an equally unspellable `f-string-keyword` token.

The *f-string-construct* is modified into a regular token sequence after determining which way to do this during overload resolution. However, as the *f-string-construct* now consists of a token sequence mimicking a function call it can be parsed to an AST representing a function call by just assuming that the `f-string-keyword` token is an identifier designating a function.

10 Parsing and overload resolution (phase 7)

When parsing the token stream from phase 6 the `f-string-keyword` is parsed as an identifier, so each *f-string-construct* is parsed as a function call. The return value of a function call is, unless it is discarded, always an argument of another function or constructor call. This is true for all types of expressions and initializers. Thus the return values of *f-string-constructs* are arguments of outer function or constructor calls and therefore in the position of arguments in overload resolution.

During overload resolution each *f-string-construct* must be *modified* to something that is valid in an argument list. This modification occurs as an intermediate step between forming the candidate function set and selecting viable functions. The default modification is to replace the `f-string-keyword` token with the regular identifier `__format__`, but if certain conditions are met by candidate functions *f-string-constructs* may be modified in other ways.

This example constructs a `std::filesystem::path` from a f-literal. The default modification `__format__` call is performed as no `std::filesystem::path` constructor is constrained by a *modifying concept*. Thus the `std::filesystem::path` constructor from `std::string` is called as (the standard implementation of `__format__` returns a `std::string`).

```

int fno = 1;
std::filesystem::path nextFile = f"File{fno++}.txt";

```

```
// After modification this results in:
std::filesystem::path nextFile = __format__("File{}.txt", (fno++));
```

The main reason for including different modifications than the default is to preserve the performance advantage of `std::print` compared to first creating a temporary `std::string` by calling `__format__` and then writing it on an `ostream`. Preserving this performance and retaining the convenience of getting the string returned by `__format__` by default has been a goal of this proposal from the start and has had different solutions in each revision. Previous versions included only one way to modify *f-string-constructs* apart from the default, while this version shows the possibility to provide different modifications, thereby providing the additional features that [P3951R2] offers with its synthesized template string objects without the overhead of synthesizing a new type for each and every f-literal.

10.1 Modifying concepts

Candidate function parameters constrained by predefined *modifying concepts* select specific ways to modify *f-string-constructs*. Modifying concepts have the same usage syntax as other concepts but apart from constraining the arguments to *f-string-constructs* they control in what way the *f-string-concept* is modified before viability checking. Using concept syntax for this purpose offers the right level of control while avoiding introducing any new syntax. In some cases it may be of interest to allow other expressions in addition to *f-string-constructs* to match a parameter. This is accomplished using a disjunction with another constraint. Such disjunction does not affect how the modification is done for arguments that are *f-string-constructs*, but for other argument types no modification occurs as they are not modelling the modifying concept even if they do model the overall constraint.

For each *f-string-construct* in the argument list the *corresponding parameters* of all candidate functions are inspected. If any of the corresponding parameters is constrained by a modifying concept this determines the modification to do. If the corresponding parameter of a candidate function is not constrained by a modifying concept the converting constructors of its type are inspected and if any of them is constrained by a modifying concept this determines the modification to do. This is consistent with how constraints generally work when there is a user defined conversion in an implicit conversion sequence. If the different corresponding parameters for different candidate functions are constrained by different modifying concepts the call is ill-formed. If no corresponding parameter is constrained by a modifying concept the default modification (replacing the **f-string-keyword** with `__format__`) is performed.

If the argument list contains multiple *f-string-constructs* the modification of *f-string-constructs* commences left to right, which means that for a modification that expands the argument list the parameter index to match is changed before the corresponding parameters of succeeding *f-string-construct* arguments are determined.

When the modifications of all *f-string-constructs* in the argument list have been performed the viability selection is done.

Which modifying concepts and corresponding modifications to standardize remains to be discussed. It is possible to only include the first one but by including more offers some additional functionality, as usual at the cost of additional complexity. It is possible to add more modifying concepts by later proposals but it may not be possible to add for instance `std::print` overloads for them at a later time as this would result in conflicting modifications which is not allowed by the rules above.

10.1.1 1. Expansion of argument expressions

The first modifying concept modifies a *f-string-construct* by concatenating all its string literals to the first one and removing the **f-string-keyword** and the parentheses surrounding its argument list, thus expanding the concatenated string literal and the argument expressions in line with any other arguments. This modifying concept is tentatively called `fliteral_expand_arguments`. This concept is used on the constructor parameter of a new `std::basic_format_string` constructor to indicate that functions that take a `basic_format_string` are to be called with the expanded argument expressions:

```

template<typename T> concept fliteral_expand_arguments = /* unspecified */;

template<typename CharT, typename... Args> class basic_format_string {
    template<typename T>
        consteval basic_format_string(const T& s);           // Existing ctor

    template<fliteral_expand_arguments T>
        consteval basic_format_string(const T& s);           // New ctor
};

```

Note that while the modifying concept is used on the `basic_format_string` constructor while the argument list after modification is used for the call to a function such as `std::print`, passing only the concatenated string literal to the `basic_format_string` constructor.

As an example consider a call to `std::print` with a f-literal as argument.

```

std::print(f"Value: {v}");

// From the preprocessor this results in:
std::print(f-string-keyword("Value: {", (v), "}"));

// As the print function's first parameter is a std::basic_format_string<char> the compiler
// sees that there is a constructor with a parameter constrained by the
// fliteral_expand_arguments concept and thus concatenates string literals and expands
// arguments before viable function selection.
std::print("Value: {}", (v));

```

10.1.2 2. An alternative to P3951 objects

Another possible modifying concept is used to encapsulate all the information about the f-literal as well as the current values of all the expression-fields as an object. In contrast with P3951 these objects are instances of specializations of a class template in the standard library. This modifying concept could be included if the additional P3951 functionality of enumerating the replacement-fields with their format specifiers and the individual parts of the string literal that they are interspersed between is viewed as essential. We call this concept `create_fliteral_object` and it modifies the *f-literal-construct* into a constructor call of a specialization of the type `std::fliteral_object`. With some additional gymnastics in the wording it would also be possible to provide the source code of each expression field as a string. As discussed above we think that this should be the source code of the invocation of the compiler, even if preprocessing was done separately before that.

The `std::fliteral_object` contains an array of string literals and a tuple of `replacement_field` objects. Each `replacement_field` object holds the value to format, its format specifier including the leading `{` and a tuple of values of the nested-expression-fields of that replacement-field. The compiler specializes `std::fliteral_object` for the particular types of the expression-fields in an unspecified way and initializes the object from the expressions of the expression-fields.

It is likely that for this modification it is convenient to not concatenate the string literal parts as it helps the compiler filling in the different string parts of the `fliteral_object` type. Note that by presenting the different strings as `string_view` objects the backing store for all of them can be just one char array in constant memory, but this is an implementation detail.

Here is a possible implementation of the struct representing one replacement field:

```

template<typename CharT, typename Arg, typename... Nested>
struct replacement_field {
    using value_type = Arg;

    // The constructor takes a basic_format_string for this argument, and checks it.

```

```

replacement_field(std::basic_format_string<CharT, Arg, Nested...> str, Arg&& value,
                  Nested&&... ns) : format_specifier(str.get()),
                  argument(std::forward<Arg>(value)), nested(std::forward<Nested>(ns)...) {}

std::basic_string_view<CharT> format_specifier;
std::basic_string_view<char> source_code; // requires extra information from phase 3.
Arg argument;
std::tuple<Nested...> nested;
};

```

When the compiler specializes this type for a replacement field the Arg and Nested types are value types for rvalue expressions and const reference types for lvalue expressions just like in P3951.

The complete f-literal is represented by a specialization of `fliteral_object` as shown below.

```

template<typename CharT, typename... Fields>
struct fliteral_object {
    static const int N = sizeof...(Fields);
    std::basic_string_view<CharT> strings[sizeof...(Fields) + 1];
    std::tuple<Fields...> fields;
};

```

These types will need to be elaborated to minimize runtime cost and storage requirements. For instance it may be advantageous to use `constexpr` functions to access the strings, and the tuple protocol may be implemented for the entire objects. Also the names are not good, but at this point we're merely showing possibilities.

Thanks to the existing rules for constrained declarations it is easy to create a variable of a `fliteral_object` type, return `fliteral_object` specializations from functions or even specify that a constant template argument must be a f-literal and is to be treated as a `fliteral_object`:

```

std::create_fliteral_object auto myFliterlObject = f"V: {x}";

std::create_fliteral_object auto myFunction(int x) { return f"V: {x}"; } // Dangling as in P3951.

template<std::create_fliteral_object auto ctp_string> class X;

```

We feel that this system delivers convenience by default while thanks to the use of concepts more expert friendly uses are still possible.

10.1.2.1 Basic usage of `fliteral_objects`

The `fliteral_object` structure allows straight-forward implementations of the examples in P3951, such as `highlight_print`:

```

// Some VT100 control sequences
#define GREEN_BOLD "\033[1;32m"
#define RESET "\033[0m"

template<typename Str>
void highlight_print(Str s)
{
    template for (constexpr int I : std::views::iota(0, Str::N)) {
        std::print("{} ", s.strings[I]);
        std::print(GREEN_BOLD);
        std::print(std::get<I>(s.fields));
        std::print(RESET);
    }
}

```

```
std::print("{} ", s.strings[Str::N]);
}
```

This implementation relies on a `std::print` overload for printing a replacement field. This function is easy to implement as `replacement_field` contains all the parts required:

```
template<typename CharT, typename Arg, typename... Nested>
void print(const replacement_field<CharT, Arg, Nested...>& field)
{
    auto& [...pack] = field.nested;
    std::print(std::dynamic_format(field.format_specifier), field.argument, pack...);
}
```

Note that as the `replacement_field` constructor has already checked the validity of the format-specifier there is no need for additional compile time checking here.

10.1.2.2 Advanced uses of `fliteral_objects`

One advantage of `fliteral_object` and P3951 objects is that a function may take multiple such objects as parameters, or even a pack of such objects. It has been suggested that this would for instance be suitable for structured logging. Allowing this was a driver for using the concept syntax to specify which modification to use as you can now write:

```
template<create_fliteral_object... FLiterals>
void my_log(const FLiterals&);
```

We can also implement a Python style print which just prints its arguments one after the other, including f-literals. This can be implemented without using the `fliteral_object` type but then each *f-literal-construct* would first be formatted to a string before the print function was called:

```
template<typename... Ts>
void python_print(const T& ts); // All f-literals are converted to strings before the call.
```

A more performant implementation of `python_print` can be achieved by a disjunction in the constraint so that f-literals are preserved as `fliteral_objects` and handled by an `if constexpr` in a `template for` over the arguments:

```
template<typename T>
concept fliteral_or_formattable = create_fliteral_object<T> || formattable<T>;

template<fliteral_or_formattable... Ts>
void python_print(const Ts&... args) {
    for template (constexpr int I : std::views::iota(0, Str::N)) {
        if constexpr (template_of(^args...[I]) == ^std::fliteral_object)
            std::print(args...[I]);
        else
            std::print("{} ", args...[I]);
    }
}
```

A similar improvement can be had for the regular print function which covers the case that a f-literal is used as an argument to a format or print call. By constraining the arguments with `create_literal_object || formattable` such f-literal arguments can be “flattened” and extraneous temporary strings be avoided.

10.1.2.3 How about the reliance on the standard library

While this modifying concept relies on the standard library types `fliteral_object` and `replacement_field` and indirectly on `std::tuple` we don’t think that this is a problem as there are no algorithms or I/O related

functionality involved in the related headers. There are already a number of facilities that rely on the standard library in similar ways which can't be replaced by other support libraries or even other implementations of the standard library: `<initializer_list>`, `<type_traits>` and `<meta>` all contain such functionality, and `<meta>` even includes `<vector>` although it is only used at compile time. P3951 also relies on a small struct `std::interpolation` although this could be replaced by a declaration of a nested struct inside each of the synthesized object types.

It would be nice to be able to replace the `std::tuple` with a pack of elements each consisting of a `replacement_field` specialization. This seems like a straight-forward extension of the pack related functionality but is far out of scope for this proposal.

10.1.3 3. Expansion without coalescing literal parts.

It has been proposed that functions like `std::print` could be made faster by providing the literal parts of the *f-string-construct* separately as `std::print` would not have to parse the concatenated literal it gets to find the `{` character starting each replacement field. It is however possible that any such advantage is offset by the work to push those extra arguments to the stack before calling the `std::print` overload. On the other hand the call may be inlined so no argument pushing is required.

In the `fmt` library there is a feature `FMT_COMPILE` which creates a data structure suitable for fast formatting at compile time. We may get this into the standard sooner or later, replacing this modification concept with something that has even better optimization opportunities.

10.1.4 4. f-literal objects without the `replacement_field` level.

Implementing `std::print` for the `fliteral_object` type shown above may be a bit more challenging than for the objects of P3951 as there is no easy way to convert all the argument expressions to a pack. However, as shown by the `highlight_print` example above it is not *that* complicated given a `std::print` overload for one `replacement_field` which is easy to implement.

Thus it seems unlikely that this modification would be worthwhile to have in parallel with `fliteral_objects` generated by modification 2 above. Having this instead of modification 2 makes implementing functions like `highlight_print` significantly harder, although with extra information such as the one the P3951 objects provide it is doable.

10.2 Perfect forwarding of f-string-constructs

To be able to forward f-literals to `make_unique<T>` without first doing any modification would require passing the *f-string-construct* as such to then determine in which way it needs to get modified depending on if the constructors of `T` are constrained by a *modifying concept* or not. While this may be possible for regular forwarding as in the `make_unique` case it is not possible for the `apply` case where the data has to be stored before use. Instead, as the parameters of functions like `make_unique` are not constrained by modifying concepts f-literals are converted to strings before `make_unique` is called. It would be possible to include a function to explicitly force a f-literal to be treated as a `fliteral_object` that can be used to pass a f-literal without converting it to a string:

```
auto to_fliteral_object(create_fliteral_object auto obj) { return obj; }
```

This line of thought has not been fully explored, maybe a solution can be found later.

11 The `__format__` function

One reason that the default modification of an *f-string-construct* results in a call to a function called `__format__` is to allow for code bases that don't use the standard library to do formatting using their own facilities.

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

A drawback of using of a name that is reserved for the implementation is that, although unlikely, some implementation may already have used it. This should be fairly easy to survey before settling on a final name, though.

Another argument against such a name would be that we're now allowing users to define this name even as it is in the set of implementation reserved names. This should not be very controversial as it is just one name with a specific spelling stated in the standard.

Of course a name that is not reserved for the implementation could also be selected if one that is sufficiently unique can be found. An example would be `default_fliteral_format_function` which is unlikely to be defined in any code base.

11.1.1 An alternative spelling

It would also 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 named 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.

In this spelling alternative phase 5 must output an explicit call to the operator function when an f-literal is encountered:

```
f"Value {x}";

// Translates to this if the default modification is done:
operator f"("Value {x}", (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 function 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 this doesn't require any changes to the core language or preprocessor just to be able to declare the function.

11.2 Location of the `__format__` function

To allow linking of translation units including the standard library definition of the `__format__` function to translation units using some proprietary definition the standard library `__format__` function declaration is located in a nested inline namespace of `std` called `fliterals`. The only reason to make this namespace inline is to make the `using namespace std;` anti-pattern make `__format__` visible. This is consistent with the `std::literals` namespace which seems to be declared `inline` for the same reason. An alternative is in fact to place the `fliterals` namespace inside `std::literals` but this proposal does not do that as it would make all code that currently makes use of `using namespace std::literals` for other purposes automatically opt-in to the standard definition of `__format__` which may not be desired.

For convenience this proposal specifies a new header file `<fliterals>` which just includes `<format>` and then contains the single line: `using namespace std::fliterals;`. The `<format>` header itself contains the definition of the `_format__` function in the `std::fliterals` namespace. Users requiring fine granularity only include `<format>` and places the using directive where f-literals formatted using the standard library are needed. Other parts of the same translation unit can format using another library's `__format__` function after a corresponding using directive.

In this proposal importing `std` as a module implies a `using namespace std::fliterals;` directive, as not doing so would hamper the new ease of use that `import std;` offers and it seems unlikely that a project which at least sometimes uses a different library for formatting would contain such a blanket import of the standard library. It would be possible to specify a new module `std_without_fliterals` which `std` imports but this seems overly complicated.

11.3 The standard implementation of `__format__`

The standard library implementation of `__format__` is located in the `<format>` header, in a and just perfectly forwards to `std::format`:

```
namespace std {

inline namespace fliterals {

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)...);
}

} // fliterals
} // std
```

With the alternative spelling this instead becomes:

```
namespace std {

inline namespace fliterals {

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)...);
}

} // fliterals
} // std
```

11.4 Alternative `__format__` functions

To create an alternative `__format__` function that calls `std::format` is no problem as including `<format>` does not make the unqualified name `__format__` visible. The simplest way to declare your own `__format__` is to just put it in the global namespace. This prevents all uses of the standard `__format__` function after this declaration has been seen which may be desired in some code bases. To preserve the possibility to mix standard f-literal behavior with alternate behavior the alternative `__format__` function should be placed in a namespace which is then mentioned in a using directive as needed.

11.5 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-constructs* created from f-literals in phase 3. As phase 5 looks for the `f-string-keyword` pp-token source code `__format__` calls are not treated differently than any other function call.

12 User-defined literal suffixes

It is not obvious how user-defined string literal operator functions should work when applied to an f-literal. The UDL identifier could be intended for the concatenated string literal, which would not need any other language changes but probably brings little value, or it could be viewed as a way to change the name `f-string-keyword` to the name of the operator function. This proposal does neither of those things.

P3951R1 proposes to allow user defined literal suffixes with the meaning that the synthesized template string object is used as argument to a new style `operator""s` function. This would come with the above mentioned caveat that any identifier following immediately after the trailing `"` of the t-literal would be treated as a user defined literal name. While P3951R1 does not mention this the logical transformation that the preprocessor would do would be to enclose the argument list in an explicit operator call:

```
t"Value {v}"s
```

is translated to:

```
operator""s(a template string object)
```

This would be the first time that a so called literal operator can take runtime varying argument values.

13 Contexts where string interpolation works

With the risk of stating the obvious: String interpolation only works in contexts where 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 constant 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.

14 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, uR, 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.

15 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 handled as usual when those macros are lexed but their usage can never occur outside the module unit as macros are not exportable.

16 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.

17 Implementation experience

There are two implementations of R0, both by Bengt.

17.1 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 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.

17.2 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 the concatenated literal. 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 literal (P3412R0 had only one) 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
// "{:.{c}}" 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 " "{:.{c}}", (a), (b), (c))
```



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

17.2.1 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 be a significant issue.

17.3 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.

17.3.1 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.

17.3.2 EDG

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

17.3.3 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.

17.3.4 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.

18 Alternatives

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

18.1 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.

18.2 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()"` literal operator 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 it 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 call similarly to how it is done in this proposal would require being able to inspect the parameter lists of the surrounding function including constraints of placeholder types. Possibly annotations could be used instead of modifying concepts.

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. There is a function `tokenize` in the latest code injection proposal, which offers some possibilities in this area.

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 f-literal to find the end of expressions involves considerably more computations than the current string literal validation done by the `basic_format_string` constructor.

19 History

Years before P3412R0 there was a proposal [P1819R0] which used an approach applied after preprocessing. A major drawback with this proposal was that it didn't handle macro expansion.

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 a similar idea of letting the preprocessor extract the expressions out of the format string and create a *f-string-construct* resembling a function call. Different revisions of this proposal have had different ways to handle these constructs during the later stages of compilation:

P3412R0 used an object to collect all information about a f-literal and relied on supporting proposals *implicit conversion functions* and *user specified type decay* to prevent dangling and make usage of f-literals convenient. As EWGI was not favorable to any of the supporting proposals...

P3412R1 gave up and specified two different prefixes `f` and `x` where f-literals always converted to a string via a `__format__` function call and x-literals resulted in an argument list suitable for `std::print` directly. This system was considered too cumbersome by EWGI as it required users to remember which letter to use where. [P3951R2] essentially only provides the `x` half of this proposal although creating an object of a synthesized type instead of just an argument list, causing a lot of extra work for the compiler for the most common use cases.

P3412R2 introduced an attribute controlling if the `__format__` function call was to be removed. This revision was not discussed at a meeting but private communication revealed that a solution not relying on any special marking of the called functions would be preferred.

In P3412R3 a system relying on certain properties of the function parameters matched to the `__format__` function call determined if it was to be removed. When pitted against [P3951R0] this version was less favoured, but not mainly for this reason but because P3951 was perceived to be more flexible for advanced uses.

In this P3412R4 version we rely on constraining function parameters as a way to not only achieve our original goal of converting to a string when not requested to do otherwise, but also to match the functionality of the P3951 objects when this is preferred. This version preserves the *f-string-construct* into phase 7 and only converts it to a `__format__` function call if not modified otherwise to satisfy constraints on the parameters of functions in the overload set.

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 5 (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 5 to create the `__format__` call which was the result from phase 5.

In newer revisions 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 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* 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 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 first pp-token is a unspellable keyword 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 parentheses added around each *expression-field* are 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 otherwise be very misleading.

20.2.1 The # operator

A special feature of the preprocessor is the prefix stringization `#` 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 amounts to counting matching parentheses.

20.3 Phase 5: Encoding detection and string concatenation

In phase 5 *f-string-constructs* are included as elements of string literal sequences to be concatenated. The encoding detection and concatenation can not be done sequentially if the output is to be produced sequentially as the encoding prefix may be found on any of the original string-literals or f-string-literals and with *f-string-constructs* the output is not just one pp-token. Instead the wording describes this as two phases of manipulation of the overall pp-token sequence.

When a string-literal or f-string-construct is encountered in the pp-token stream scanning continues to see if there is one or more *f-string-constructs* in the sequence and there is their wrapping **f-string-keyword** and the outer parenthesis around the *f-string-constructs*, all the while keeping track of any encoding prefix. In the next step, starting on the first pp-token of the sequence apply the common encoding prefix and concatenate all groups of adjacent string-literal tokens (including those that always start and end *f-string-constructs*), while skipping over the expression-fields by counting parentheses. Finally enclose everything in a **f-string-keyword** and an enclosing parenthesis.

This process can be worded quite succinctly and is not that far from a realistic implementation strategy.

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:
f-string-keyword(L"{{Hello {" , (name) , L"} , you{" , (" lucky one."), L"}}")");

// The default modification results in:
__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"), "}")

// Program output
The name is Bjarne
```

20.4 Phase 7: Compilation

20.4.1 The new overload resolution step

This step is presented as a new paragraph [over.match.modify] which defines the new procedure of expanding the last `__format__` call to its arguments unless modifying concepts on function parameters instructs the compiler to do another modification.

The description of the modifying concepts themselves could be a new subchapter of concepts but as these are named concepts they could also be described in the library part of the standard. Maybe it is best to split the description so that the basic idea of modifying concept is described as a chapter under concepts and the actual modifying concepts are described under library. This is however rather fragmenting and maybe not necessary as there will be a maximum of a handful of modifying concepts and no possibility for users to add more.

21 Wording

This chapter presents diff towards N5054.

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 first string literal of each **f-string-construct** is then considered to have that common encoding-prefix. Then, 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 5

[Editor's note: This addition renumbers the subsequent paragraphs, but the paragraph numbers of N5054 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.]

- ⁵ 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 lexing one *f-literal* or *raw-f-literal*. These types of string literals can contain *expression-fields* enclosed in brace characters, consisting 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, brace and splice 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.

[Editor's note: Maybe move Note 2 from after paragraph 4 or duplicate it for the new paragraph 5.]

21.2.3 Add paragraphs before 5.1

- (5.1) — 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*
- (5.2) — 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 is applied before any d-char, raw-f-char, or delimiting parenthesis is identified. The raw-f-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 5.1 (now 6.3) to start:

- (5.3) — 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}

replacement-field:

expression-field
expression-field : *format-specifier*_{opt}

f-string-contents:

f-char-sequence
*f-char-sequence*_{opt} { *replacement-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-replacement-field:

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

```

raw-f-string-contents:
  raw-f-char-sequence
  raw-f-char-sequenceopt { raw-replacement-field } raw-f-string-contentsopt

raw-f-string:
  " d-char-sequenceopt ( raw-f-string-contentsopt ) d-char-sequenceopt "

```

21.4.3 After paragraph 5

Add new paragraphs:

[Editor's note: This addition renumbers the subsequent paragraphs, but the paragraph numbers of N5054 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-fields* and pp-token sequences created by regular lexing of the characters constituting the *expression-fields*, 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.

[*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*]

- (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, braces and splicers are matched and if a right brace, colon or **>** pp-token follows in the input stream when all such pairs are matched the *expression-field* has ended.
- (6.3) When an expression-field ends with a colon pp-token and the succeeding characters are any number of space characters followed by a colon or right bracket one of these space characters is removed.

[*Example:* The following f-string:

```
f"Hexcodes: {vec: :x}"
```

results in the following pp-token sequence:

```
f-string-keyword("Hexcodes: {", (vec), " :x}")
```

which allows specifying an element format when formatting containers. — *end example*]

- (6.4) Matching of nested pairs of parenthesis, bracket, braces and splicers is the process of keeping track of the number of left parenthesis, left bracket, left brace and left splicer (**[**:) pp-tokens generated minus the number of right parenthesis, right bracket, right brace and right splicer (**]**:) pp-tokens generated. Whenever this sum is 0 all pairs are matched. If this sum is negative or if parenthesis, bracket, brace and splicer pairs are not properly nested the program is ill-formed.
- (6.5) Parenthesis, bracket, brace and splicer pairs are properly nested if for each right parenthesis, bracket, brace and splicerr pp-token **R** generated, the closest previous left parenthesis, bracket, brace or splice pp-token **L** that has

not already been matched to a right parenthesis, bracket, brace or splicer 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, if **R** is a right brace pp-token then **L** is left brace pp-token and if **R** is a right splicer pp-token then **L** is left splicer pp-token.

- (6.6) 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*]

[*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. This internal state is part of the expression field state. — *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] and phase 5 (determining encoding prefix and 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*: A string-literal's rawness has no effect on the determination of the common encoding-prefix. — *end note*]

21.4.5 In paragraph 8

Change the first sentence to:

In translation phase 5 (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 itself constituting a *f-string-construct*. This is done by removing all **f-string-keyword** pp-tokens and the outer parenthesis pair of each *f-string-construct* of the sequence, then concatenating all sequences of string-literals outside of matched parentheses and applying the common encoding prefix to them, and finally enclosing the resulting pp-token sequence in a parenthesis pair after a **f-string-keyword**.
- (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*.

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

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

After phase 3:

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

After removing all f-string-keyword and outer parentheses, as well as doubling the braces of the string-literal:

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

After concatenating all string-literals:

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

After phase 5, i.e. after wrapping in a new *f-string-construct*:

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

Resulting string after formatting would typically be:

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

— *end example*]

[*Note:* To determine if a concatenation is to result in a string literal pp-token or a combined *f-string-construct* requires looking ahead in the pp-token sequence to determine if there is any *f-string-construct* following the first zero or more *string-literal* pp-tokens. If there is the result is a combined *f-string-construct*. Apart from this look-ahead requirement the concatenation procedure can be performed pp-token by pp-token, if enough state is kept to determine where each f-string-construct ends. — *end note*]

[*Note:* As the pp-token sequences resulting from the *expression-fields* of an outer *f-string-construct* can contain *string-literals* and *f-string-constructs* that are to be concatenated the concatenation process must be performed on these pp-token sequences. If concatenation is performed pp-token by pp-token the state used to determine the end of a *f-string-construct* has to be saved when a nested sequence to concatenate starts, and restored after it is complete. — *end note*]

21.5 In [over.match.general]

Add a new paragraph after paragraph 2.7 briefly describing the new step in [over.match.modify] which runs before viable function selection :

- (2.8) First, arguments that consist of a *f-string-construct* are *modified* into code that can be further compiled. This modification is done in different ways depending on whether any of the *modifying-concepts* is present on corresponding parameters of candidate functions (12.2.3).
- (2.9) **First**Then, a subset of the candidate functions (those that have the proper number of arguments and meet certain other conditions) is selected to form a set of viable functions (12.2.~~3~~4).

21.6 After 12.2.2 [over.match.funcs]

Add a new level three clause [over.match.modify] as 12.2.3, bumping [over.match.viable] to 12.2.4

21.6.1 12.2.3 Modifying *f-string-construct* arguments

- ¹ For the first argument in the argument list that consists of a *f-string-construct* inspect the types of the corresponding parameter of all candidate functions, disregarding candidate functions with too few parameters. Determine the set of *modifying-concepts* these types are constrained by. If a parameter type is a user defined type with converting constructors any *modifying-concept* constraining the parameters of these converting constructors are added to the set. If a *modifying concept* is part of a disjunction with other constraints (after constraint normalization) it is added to the set.

If the resulting set has one member perform the argument modification associated with this *modifying-concept*. If the resulting set is empty replace the leading **f-string-keyword** of the argument *f-string-construct* with the

identifier `__format__` and concatenate all the string literals in the *f-string-construct* to the first one, deleting the preceding comma of each. If the resulting set has more than one member the function call is ill-formed.

Repeat this process until all *f-string-constructs* have been modified. The resulting argument list is used when selecting viable functions (12.2.4).

[*Note:* As the modification of an argument may change the number of arguments in the argument list the corresponding parameter for any succeeding *f-string-construct* argument can change due to preceding modifications. — *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.

- ³ Each occurrence of a *f-string-construct* in the *stringizing argument* is replaced by the original f-string 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 interspersed with the original spelling of the pp-tokens of the pp-token sequences of the *expression-fields* in the *f-string-construct* excluding their enclosing parentheses and separating commas. Whitespace is handled as for all other pp-tokens being stringized.

Any *f-string-construct* in an expression field is stringized by the same procedure.

As for all stringized string literals the quotes delimiting a stringized *f-string-construct* are quoted.

[*Note:* The original encoding prefix can be determined from the first *string-literal* pp-token inside the *f-literal-construct*. Any *d-char-sequence* of a *f-string-construct* resulting from the lexing of a *raw-f-string* must be determined as for regular *raw-string* pp-tokens being part of the *stringizing argument*. — *end note*]

[*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-string. 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-strings.

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
namespace fliterals {
    template<class... Args>
        string __format__(format_string<Args...> fmt, Args&&... args);
    template<class... Args>
        wstring __format__(wformat_string<Args...> fmt, Args&&... args);
}
```

[Editor's note: Once we have established a more detailed design and naming of the `fliteral_object` and `replacement_field` data types their synopsis would go here and the description after the description of the `__format__` functions.]

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.

21.11 ... somewhere ...

Somewhere in the wording the available *modifying concepts* and their modifications to the *f-string-construct* must be described. It is hard to find a good place for it though. Maybe in the library section along with the `__format__` function but as the *modifying concepts* affect the core language this may not be appropriate. Maybe as a special chapter after the general description of concepts is better. It is also possible that the *default modification* should be described too although it is not associated with a named modifying concept.

Here is however an initial wording attempt:

21.12 Modifying concepts

Modifying concepts are special built-in concepts which are modelled by *f-string-constructs* only. In addition each named *modifying-concept* controls how the *f-string-construct* is modified in [\[over.match.modify\]](#).

21.12.1 Expanding argument expressions

The modifying concept `fliteral_expand_arguments` causes the string literals in the *f-string-construct* to be concatenated and then the *f-string-construct* is replaced by this string literal followed by the expression-fields separated by commas.

21.12.2 Creating a fliteral_object

The modifying concept `create_fliteral_object` causes the *f-string-construct* to be replaced by a constructor call to a specialization of the `std::fliteral_object` class. It is unspecified how this is done and which constructors `fliteral_object` and its subordinate `replacemenet_field` types have, just that an object corresponding to the original *f-string* is created.

[Editor's note: Additional modifying concepts and their modifications can be described if we want them.]

22 Acknowledgements

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 wording review before publication of R3.

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

23 References

- [N5054] Thomas Köppe. 2026-07-16. Working Draft, Programming Languages — C++. <https://wg21.link/n5054>
- [P1819R0] Vittorio Romeo. 2019-07-20. Interpolated Literals. <https://wg21.link/p1819r0>
- [P2826R4] Gašper Ažman. 2026-06-26. Expression Aliases. <https://wg21.link/p2826r4>
- [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>
- [P3951R0] Barry Revzin. 2026-01-10. String Interpolation Objects. <https://wg21.link/p3951r0>
- [P3951R2] Barry Revzin. 2026-09-12. String Interpolation Objects. <https://wg21.link/p3951r2>
- [P4235R0] Matthias Wippich. 2026-09-22. Removing Digraphs. <https://wg21.link/p4235r0>
- [P4364R0] Barry Revzin. 2026-09-10. Emitting messages at compile time with std::format. <https://wg21.link/p4364r0>
- [P4380R0] Barry Revzin, Daveed Vandevoorde, Andrei Alexandrescu. 2026-09-22. Token Sequence Injection. <https://wg21.link/p4380r0>