String interpolation

Document #:	P3412R0
Date:	2024-10-14
Project:	Programming Language C++
Audience:	EWG/EWGI

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

Contents

1 Abstract

This proposal adds string interpolation (so called f-literals) to the C++ language. Each f-literal is transformed by the preprocessor to a token sequence constituting a call to a `make_formatted_string` function which returns an instance of a new library type `basic_formatted_string<CharT, Args...>` which has a conversion function to `std::basic_string<CharT>`. The preprocessor extracts each expression-field into a separate argument expression inside the function call, and as the first function argument it provides the remaining string literal after extracting the expressions-fields.

In addition this proposal includes overloads of `std::print` and `std::println` which have only a `basic_formatted_string<char, Args...>` as parameter (apart from any `ostream&` or `FILE*`). This avoids any performance overhead that would have been incurred if the extracted argument list had been wrapped in a std::format call directly.

An `operator<<(ostream&, basic_formatted_string<char, Args...>)` which optimizes performance when outputting f-literals to `ostream` is also included. This optimization can't be done with `std::format` and therefore actually reduces the need for `std::print`.

This proposal has been implemented in a Clang fork. An effort to make this implementation available on Compiler Explorer is ongoing. It has also been implemented as a separate program, which demonstrates the viability of this proposal for tools like syntax coloring editors

Preferably the conversion operator to `basic_string<CharT>` should be declared `implicit` according to [P3298] and `basic_formatted_string<CharT, Args...>` should have a `decays_to(std::basic_string<CharT>)` specifier according to [P3398]. Without these proposals f-literals as defined here are less convenient and present a new risk of dangling references.

2 Examples

Assuming a Point class which has a formatter we can now use string interpolation to format Points, and with [P3298] and [P3398] we can do it transparently and without risk of dangling references.

```
Point getCenter();

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

auto b = f"Center is: {getCenter()}";             // No dangling problem with P3398

size_t len = f"Center is: {getCenter()}".size(); // Works with P3298

std::println(f"Center is: {getCenter()}");        // Uses optimal overload of println
```

```
std::cout << f"Center is: {getCenter()}";        // Uses optimal overload of operator<<
```

3 History

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

The proposal presented here uses the same basic idea of letting the preprocessor extract the expressions out of the format string and place them as an argument list after the remaining literal. In this proposal all of this is enclosed in a function call, while the original proposal had both f and x literals where the x variety did not enclose the resulting argument list leaving it useful for the current overloads of `std::print` and similar facilities, while f literals enclosed the result in a `std::format` call directly producing a `std::string`.

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

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

4 Motivation

Before this proposal:

```
int calculate(int);

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

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

After this proposal:

```
int calculate(int);

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

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

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()` that the C++ `std::format()` was based on.

Many other programming languages also offer string-format interpolation, and use identical syntax. (full list)

The main benefit of f-literals is that it is far easier to see the argument usage locations, and it is less verbose.

For example in the code snippets above, in the second example it is easier to see that "`prefix`" goes before `__LINE__`, and "`bits`" is displayed in hex.

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

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

5 Terminology

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

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

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

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

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

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

6 Expression-field contents

The contents of an expression-field is a full *expression*. The grammar for *expression* includes the comma operator so when the expression is extracted by the preprocessor and placed after the literal each extracted expression is enclosed in a parenthesis. This prevents an extracted expression from being interpreted as multiple arguments to the `make_formatted_string` function call that the f-literal results in.

Allowing a full *expression* instead of only a *assignment-expression* is needed to avoid causing errors due to commas in template argument lists, which can't be easily differentiated from other commas by the preprocessor. This is illustrated by the following examples:

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

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

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

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

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

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

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

For comparison, Python has supported string interpolation for many years but in 2022 the definition of expression fields was changed to a full Python expression, including nested string literals with the same quote kind as the containing f-literal (Python allows enclosing strings in either single our double quotes, and previously nested string literals had to use the opposite quote kind compared to the enclosing f-literal). This change was proposed in PEP-701 which was incorporated into Python 3.12.

6.1 Detecting where the expression ends

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

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

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

- Scope resolution operators. These are double colons followed by an identifier or the `operator` keyword. If there is something else after a colon-colon token the first colon must be the start of a *format-specifier* and the second a colon used as the fill character. The standard format-specifier's fill character syntax requires a `<`, `^` or `>` character after the fill character, none of which can't start an identifier. While it is possible that a user-defined formatter allows a leading colon followed by one or more letters this seems unlikely. If such a formatter exists its users will have to change the format specifier syntax to for instance allow a space between the colons in order to be able to use f-literals for their formatting.
- The colon of ternary operators must not be mistaken for the start of a *format-specifier*. This can be handled by just counting the number of `?` tokens and ignoring as many colons. An alternative, used in the Clang implementation, is to recurse to the *expression-field* handler for each `?` encountered., basically following the C++ grammar.
- The digraph `:>` could be handled either by not supporting digraphs, in which case it would immediately be lexed as a colon followed by a `>` which thus means a `format-specifier` starting by a right-alignment specification. As std::format does not support using the digraphs `<%` and `%>` to enclose *extraction-fields* instead of braces we may assume that anyone able to type a f-literal can also type a `]` and does not have to resort to the problematic `:>` digraph. An alternative, which is used in the Clang implementation, is to do a special test if an unmatched `]` token is encountered: If it was formed from the digraph sequence break it up into the separate `:` and `>` to form the expected start of the *format-specifier*. This allows using :> as a `]` substitute as long as it is balanced within the *expression-field*.

The current implementations both use a somewhat more complex parser where nested parentheses and square brackets are also skipped over. This improves error handling by detecting mismatched parentheses in expression-

fields immediately and aids in the handling of :> digraphs. With the currently proposed syntax for reflection splicing using `[:` and `:]` to enclose an expression of type `meta::info` ignoring colons inside matched square brackets becomes mandatory.

Further into the future, if more uses of colons inside expressions are specified, the rules for f-literal lexing may have to be updated, and such new uses of colons would have to be denied if it would mean that it is impossible to detect the end of an expression-field. Thus specifying a full *expression* as allowed in expression-fields is future proof. That is to say, the rules above need not to be written explicitly in the standard, it is enough to refer to the grammar for *expression* and the rules follow from this, including any future modifications.

6.2 Preprocessor directives in expression-fields

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

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

6.3 Error handling

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

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

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

6.4 Implementation in other tools

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

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

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

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

7 Nested expression-fields

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

8 Encoding and raw literal prefixes

The f prefix can be combined with the R and L prefixes. Theoretically it can also be combined with with the u, U and u8 prefixes, but as `std::format` is only available for char and wchar_t this does not currently work. Another proposal to implement formatting for other character types than char and wchar_t would be needed to address this limitation.

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

9 User defined suffixes

It is unclear how user defined string literal operator functions would work. As the constructor parameter of `std::format_string` must be a literal only the compile-time version of string literal operators seems feasible. But on the other hand a reasonable use case for an user-defined string literal suffix would be translation. At the cost of loosing compile time format checking it may seem feasible to allow translation via user-defined suffix as vformat just takes a `std::string_view` as the formatting string, but this would introduce a new way to end up with a dangling reference.

A major problem could be for the preprocessor to know whether an identifier after a f-literal is to be moved into the formatted_string constructor call or left behind it. Maybe this could be depending on whether there is intervening whitespace or not but at least it requires the preprocessor to check the next token before it can produce the final output for the f-literal.

Given a compile time user-defined string literal suffix `_uc` that uppercasifies the literal we get this example:

```
// What would this input translate to?
f"value: {x}"_uc;
f"value: {x}" _uc;

make_formatted_string("value: {}"_uc, (x)); // #1

// OR

make_formatted_string("value: {}", (x))_uc;   // #2
```

Given that use cases are marginal and dangling is an issue this proposal does not special treat identifiers after f-literals and a conforming implementation should produce output according to #2 regardless of if there is an intervening space or not, under the assumption that if this names a user-defined string literal suffix a compilation error will result. As a QoI improvement an implementation could produce a more specific error message detailing that f-literals can't have user-defined string literal suffixes. If we get match expressions this means that the expression `f"Value {1}"match { "Value 1" => true; _ => false; }` is expanded to `make_formatted_string("Value {}", (1))match { "Value 1" => true; _ => false; }` which should work as expected (and return true).

10 Macro expansion

Macro expansion occurs in the expression-fields just as in any expression. It must happen before the tokens in the expression-field are lifted out and placed after the remaining-literal as otherwise a macro that expands to a token sequence containing an unmatched parenthesis of some kind, a colon or a question mark could fool the rather simplistic mechanism that the preprocessor has to detect the end of an expression-field.

One issue is if the token that ends the expression field is itself part of a macro expansion. This must be an error as the remainder of that macro expansion would textually be part of the remaining-literal, which is counter-intuitive and basically requires converting the token sequence of the macro back to a string. A preprocessor implementation surely has a way to detect that a token in the lexer is part of a macro expansion and can thus produce a diagnostic in this case. In simple words the `}`or `:` that ends the expression-field must be part of the character string of the enclosing f-literal.

```
#define COLON :

// Ok: The COLON is part of the ternary operator expression.
f"Value: {a > b ? 1 COLON 0 :<5}";

// Bad: The macro expansion contains the end of the expression-field.
f"Value: {a > b ? 1 : 0 COLON<5}";
```

11 Contexts where string interpolation works

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

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

It is assumed that any later proposal that makes `std::format` constexpr will also add constexpr specifiers appropriately on `make_format_string` and member functions of `basic_formatted_string`.

12 String literal concatenation

String literal concatenation is supported in a way that moves all extracted expressions, in order, to after the concatenation of all the remaining-literals of the sequence. Only f-literals are subjected to extraction if the string literal sequence contains both f-literals and regular literals. Letting one f prefix make preceding concatenated string literals f-literals is not possible as the preprocessor would have to go back and reinterpret the preceding string literals as f-literals. This may result in their end points moving, as what was thought to be the end of the literal was actually the start of a nested literal inside an expression field. This makes such a definition infeasible.

The concatenation itself works according to the current rules, i.e. that non-raw and raw literals can be concatenated but only one encoding prefix kind may exist and it affects all literals in the sequence.

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

Here is a contorted example which shows different types of string literals being concatenated, some of which are expanded from macros and some of which are f-literals. Note that the expansion steps shown below are for

illustrative purposes only, a preprocessor/compiler is free to take other steps or just one step as long as the result on the line #2 is the same.

```
#define LITERAL " lucky one."
#define FLITERAL f" {name},"

const char* name = "John Doe";
L"{Hello" FLITERAL fR"abc( you{LITERAL}}})abc";     // #1

// The preprocessor first expands macros to get:
L"{Hello" f" {name}," fR"abc( you{" lucky one."}}})abc"

// Then the detection of an f-literal causes this to be transformed to:
::std::make_formatted_string(L"{Hello" " {}," R"abc( you{}}})abc",
                             (name), (" lucky one."));

// String concatenation as we know it then transforms this further to:
::std::make_formatted_string(L"{{Hello {}, you{}}",
                             (name), (" lucky one."));  // #2

// The std::wstring conversion function then calls vformat at runtime to get:
L"{Hello John Doe, you lucky one.}"    // #3
```

One of the literals at #1 above has an encoding prefix and two have f-prefixes. Extraction fields are only processed in those literals that have the f-prefix and that the encoding prefix extends to all the literals, while the raw prefix only applies to the immediately following literal. The expressions in all the f-literals are moved after the remaining-literal sequence to allow vformat to operate correctly when called inside the `operator std::wstring()` of the `basic_formatted_string<wchar_t, Args...>` instance.

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

12.1 Quoting of non-f literal contents during concatenation

The example above illustrates a problem with concatenating f-literals with regular literals containing `{` or `}` characters. The programmer writing the line at #1 does not think doubling of the initial `{` character is needed as it is not in an f-literal. However, after concatenation, at #2, this quote is doubled, which is something the preprocessor has to do inside the contents of regular literals when concatenated to f-literals. Thanks to this doubling the final result again contains single braces at #3.

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

13 Code breakage risk

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

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

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

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

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

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

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

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

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

14 Debugging feature

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

```
f"{x=}";

translates to

std::make_formatted_string("x={}", x);
```

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

15 The basic_formatted_string type and its maker

Each f-literal results in the construction of an instance of the type `basic_formatted_string<CharT, Args...>` where CharT is the character type implied by the original f-literal's encoding prefix and Args… are the types of the extracted expressions.

Note that to use string interpolation the header `<format>` must be included or a corresponding module imported. The quality of any error message related to not including `<format>` is implementation defined.

```
// Defined in <format>
template<typename CharT, typename... Args>
struct basic_formatted_string decays_to(basic_string) {
    basic_formatted_string(basic_format_string<CharT, Args...> fmt, const Args&... as) :
        args(make_format_args<__select_fc_t<CharT>>(as...)), literal(fmt.get()) {}

    implicit operator string() const { return vformat(literal, args); }

    decltype(make_format_args<__select_fc_t<CharT>>(declval<const Args&>()...)) args;
    basic_string_view<CharT> literal; // Bike-sheddable!
};
```

```
template<typename... Args>
using formatted_string = basic_formatted_string<char, Args...>;

template<typename... Args>
using wformatted_string = basic_formatted_string<wchar_t, Args...>;
```

This requires a *exposition-only helper* `__select_fc_t` which can select the format_context specialization for the character type (which has an unspecified first template parameter) as shown here. This and the fact that `std::vformat` is only defined for `string_view` and `wstring_view` as the first parameter limits the use to `char` and `wchar_t` unless another proposal extends the set of character types usable for formatting.

Due to limitations in CTAD we need a special `make_formatted_string` function that can pick up the character type from the remaining-string literal and the argument types from the extracted argument expressions. This function is trivial but needs an explicitly written overload for each supported character type or construction of the `fmt` function parameter will fail.

```
// Defined in <format>
template<typename... Args>
auto make_formatted_string(std::format_string<Args...> fmt, Args&&... as)
{
    return formatted_string<Args...>(fmt, as...);
}
template<typename... Args>
auto make_formatted_string(std::wformat_string<Args...> fmt, Args&&... as)
{
    return formatted_string<Args...>(fmt, as...);
}
```

A viable alternative is to only specify separate `formatted_string` and `wformatted_string` class templates. However, as all the other formatting classes are specified as templates starting with `basic_` the proposal is to continue with this convention.

The `make_formatted_string` name could be exposition only as it is not intended that any user code should call this function, call sites should only be generated by the preprocessor. However, considering the preprocess-only case where a programmer may inspect the preprocesor output it seems that making this function name well-defined is a good choice.

16 ostream insertion operator and print functions

This proposal contains new overloads of `operator<<` to use f-literals with `ostream` objects without having to produce an intermediate `std::string` as well as overloads of `std::print` and `std::println` that provide modern style printing of f-literals.

```
// Defined in <print>
template<typename... Args>
void print(FILE* stream, const formatted_string<Args...>& fs);
template<typename... Args>
void println(FILE* stream, const formatted_string<Args...>& fs);

template<typename... Args>
void print(const formatted_string<Args...>& fs) { print(stdout, fs); }

template<typename... Args>
void println(const formatted_string<Args...>& fs) { println(stdout, fs); }
```

```cpp
// Defined in <ostream>
template<typename CharT, typename... Args>
basic_ostream<CharT>&
operator<<(basic_ostream<CharT>& os, const basic_formatted_string<CharT, Args...>& fs)
{
    vformat_to(ostream_iterator<CharT, CharT>(os), fs.literal, fs.args);
    return os;
}

template<typename... Args>
void print(ostream& os, const formatted_string<Args...>& fs)
{
    os << fs;
}
template<typename... Args>
void println(ostream& os, const formatted_string<Args...>& fs)
{
    os << fs << "\n";
}
```

Note that thanks to the use of `vformat_to` and `ostream_iterator` these functions are as performant as the current `std::print` overloads and more performant than inserting the `std::string` returned by `std::format` into an ostream.

It is a bit unclear which way to go with the print functions that take a basic_ostream and a basic_formatted_string as parameters as there are no corresponding print function with a format_string and a list of corresponding arguments as parameters. Either we could remove the CharT template parameter on all the new `print` and `println` overloads or we could add the missing overloads within the current standard. In the interest of minimizing feature creep this proposal does not include any print overloads for `wchar_t`. A future proposal may add `print` and `println` functions for other character types than `char`, at least `wchar_t`.

These classes and functions are demonstrated on Compiler Explorer here.

17 Implementation experience

There are two implementations, both by Bengt.

17.1 A stand alone implementation

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

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

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

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 a f-literal as a header file name and when the end of an expression-field is inside a macro expansion. As this fork currently does not implement [P3398] it is less safe to use due to dangling risk, and to avoid double user-defined conversions f-literals may have to be converted explicitly to std::string in some cases as [P3298] is not implemented.

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

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

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

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

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

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

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

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 `make_formatted_string` function call can be made directly during lexing. An advantage of this is that running Clang just for preprocessing would work without additional coding, but a drawback is that for concatenated literals without any f-literal there is a small performance overhead as the literal sequence must be injected back into the preprocessor which involves additional heap allocations. As only a small fraction of string literals involve concatenation this should not a be a significant issue.

18 Alternatives

A few other approaches to get string interpolation into C++ have been proposed, which are discussed here, along with the prospect of standardizing this proposal without its supporting proposals.

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 [P3398] to avoid dangling in the simple case of assigning an auto variable to a f-literal. A bigger disadvantage seems to be that, at least according to the proposal, there is no way to implicitly convert the f-literal to a std::string, usage is restricted to printing and ostream insertion.

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. Secondly there seems to be no inherent way that the leading f can be handled by reflection. A mechanism where a certain identifier can be connected to some kind of reflection mechanism would be needed. The closest approximation would be something like `std::f("...")` which is not the level of ergonomy we aim at for string interpolation.

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

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.

18.3 Implementation without supporting proposals

It would be possible to implement this proposal without the [P3298] and [P3398] proposals in place. However, the dangling risks involved if P3398 is not available are quite severe as seen from the first examples of this proposal:

```
Point getCenter();

auto b = f"Center is: {getCenter()}";            // getCenter return value dangles.
```

Not implementing P3298 is a smaller problem but introduces somewhat hard to understand errors such as in the simple case below:

```
void runFile(std::filesystem::path fileName);

runFile("file5.dat");       // Ok of course.

int n = 5;
runFile(f"file{n}.dat");    // Does not work without P3298.
```

If the operator `std::string()` of formatted_string can't be declared `implicit` the conversion from formatted_string to filesystem::path requires both a user defined conversion and a constructor call which is not allowed.

The capabilities offered by P3298 can be added later but the authors think that standardizing this proposal without simultaneous standardization of P3398 is too risky.

18.4 Implementation with magic formatted_string

Another approach that would work is to make the `basic_formatted_string` class template magic in the sense that the compiler acts as if it had the `decays_to(std::string)` specifier and `implicit` conversion function without actually adding those features to the language. It should be possible to *disenchant* this type later if those proposals are accepted as the semantics of the program would not change, only the definition of the class in the standard header.

The obvious drawback inherent in defining a magic type is somewhat offset by the fact that the preprocessor would need to conjure up calls to `make_formatted_string` anyway, which binds the core language to the library.

19 Wording

None yet.

20 Acknowedgements

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

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

21 References

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