{fmt} – Type-Safe Text Formatting

Proposal for C2y
Date: 2026-09-10
Document: n3864
Revision: 0
Audience: WG14
Proposal Category: New Features
Target Audience: General Developers
Reply-to: Jonathan Grant <jgrantonline@gmail.com>

Abstract
This paper introduces `fmt_print()`, `fmt_fprint()`, `fmt_format()`, and supporting functions for the text formatting system based on C++ P0645 – {fmt} Text Formatting, that was incorporated into C++20. By eliminating error-prone format specifiers such as %s we reduce a class of safety vulnerabilities.

Contents

1. Introduction

This paper proposes a unified, type-safe API for string formatting.

This paper proposes:

```
/* Core Formatting Functions */
fmt_result_t fmt_print(const char *restrict format, ...);
fmt_result_t fmt_fprint(FILE * restrict stream, const char *restrict
format, ...);
fmt_string_t fmt_format(const char *restrict format, ...);

/* Inspection functions */
size_t fmt_length(const fmt_string_t * buf);
size_t fmt_capacity(const fmt_string_t * buf);
const char * fmt_string_data(const fmt_string_t * buf);
const char * fmt_result(const fmt_string_t * buf);
fmt_result_t fmt_result_code(const fmt_string_t * buf);

/* Cleanup functions */
fmt_result_t fmt_free(fmt_string_t * buf);
fmt_result_t fmt_clear(fmt_string_t * buf);
```

`fmt_print()` outputs to stdout.
`fmt_fprint()` outputs to the specified stream.
`fmt_format()` formats to a `fmt_string_t`.

```
typedef struct
{
    char * heap_ptr;  // NULL if using internal_buf
    size_t length;    // Current string length (excluding NUL)
    size_t capacity;  // Current buffer size (internal, or heap)
    fmt_result_t result;
    char internal_buf[FMT_CAPACITY];
} fmt_string_t;
```

The goal is to provide facilities that are at least as fast as current string processing by printf().

```
    fmt_print("Hello {}\n", "world");

    fmt_print("int = {}\n", 42);

    fmt_print("Mixed strings: {} {} {}\n", "name", 10, 1.25);

    fmt_fprint(stderr, "stderr {}\n",  "foo");

    fmt_print("pi = {}\n", 3.14159);
```

If an argument type is not supported, the behavior is implementation-defined. Printing hexadecimal may be appropriate.

2. **Motivation and Scope: Eliminating Bugs and Saving Time**

C lacks a modern type-safe string formatting set of functions. There is a desire to simplify formatting, and not require users to specify correctly %s, %d and other specifiers. Programmers may not know the format specifiers for certain types. which leaves them having to review manuals, compile code, and review compiler suggested format specifier corrections, which also may not work; sometimes they are left adding casts to types they do know the specifiers of.

```
// Passing an integer where string is expected -> UB crash
printf("%s, %d", 3, "string");
```

% is fundamentally tied to unsafe type design. In `printf()` a string encodes expected types. The problem is that there is no connection between `%s` and the actual argument type. The compiler cannot reliably verify, which leads to UB. Some compilers are advanced enough to introduce special case verification of parameters against the format string.

Using {} replacement fields completely decouples formatting from type specification, provides time savings during code maintenance and boosts developer productivity. Furthermore it aligns C with other safety-focused languages like Python (`str.format)`, Rust (`format!`), C++20 with (`std::format()`).

3. **Design Goals and Safety Benefits**

Compile-time safety, automatic type deduction via `_Generic` prevents runtime specifier mismatches.
Time saving, simplifies refactoring and format string debugging sessions.
Bounded buffer safety, prevents buffer overflows and guarantees string composition without truncation or reading beyond the value being formatted.
Zero overhead, performance execution comparable to `printf()` without sacrificing safety.

4. **Proposed Design**

Implementations should use `_Generic()` which was added in C11 with tagged representations to implement argument handling and retain type information for the formatter.

NB: this is needed because pure variadic functions using standard `va_list` arguments lose runtime type information and cannot determine types at runtime. The `fmt_*` family of functions use `_Generic` to prepare a tagged format struct array, which the tag handlers can format by dispatches to those specific handlers. It is for this reason, to make this capability work `fmt_print()`, `fmt_fprint()` and `fmt_format()` and others are defined as macros. The macro entry points wrap the variadic arguments as a tagged array via the `_Generic()` capability.

A format string consists of ordinary characters and replacement fields.
A replacement field is a pair of characters "{}" which consumes the next argument.
The sequences "{{" and "}}" are replaced by "{" and "}" respectively and do not consume arguments.

Where there is a mismatch between {} and arguments, the `result` member is set, and no processing or truncation occurs. The result may be `FMT_ERROR_MANY_ARGS` or `FMT_ERROR_FEW_ARGS`

4.1 Implementation Experience

The preliminary sample implementation is not complete, the github repository contains a list of known issues. The intention is to align as closely with C++ {fmt} as possible.

The implementation defaults to retain a small internal buffer of 0x100 bytes, and allocates only if exceeded. It works using `_Generic(x)` to format each supported type. The size of the internal buffer would be decided by implementations.

The implementation carefully avoids leaving a buffer with truncated strings. In the case of an issue it sets the `result` member, for which a string name can be accessed by `fmt_result()`.

The sample implementation supports int, uint, double, string, float, bool, void*.

Modern compilers may perform static analysis to catch some printf formatting issues.

This is safer because the type is used to infer formatting, there's no crash from %s a double.

Could add further compile-time validation by utilizing C proposal N3950 `compile_assert()`.

4.2 Memory management

The {fmt} library may allocate memory if the string size required is larger than the internal capacity. The heap memory should be released user calls `fmt_free()`. If a heap buffer was allocated, `fmt_free()` releases it and resets to the empty state; if nothing was allocated it resets to initial empty state. As the string is now reset to its initial empty state, it could be re-used as though it was a new string. Programmers should take care to call `fmt_free()` before calling `fmt_format()` again and assigning the same struct variable, without calling `fmt_free()`, memory would be leaked.

4.3 Struct and Enum

<fmt.h> listing

```
#define FMT_CAPACITY 0x100

typedef enum
{
    FMT_OK = 0,
    FMT_ERROR_FORMAT = 1,
    FMT_ERROR_MEMORY = 2,
    FMT_ERROR_IO = 3,
    FMT_ERROR_MANY_ARGS = 4,
    FMT_ERROR_FEW_ARGS = 5,
    FMT_ERROR_NULL = 6,
    FMT_ERROR_UNKNOWN = 7
} fmt_result_t;

typedef enum
{
    FMT_INT = 0,
    FMT_UINT = 1,
    FMT_DOUBLE = 2,
    FMT_STRING = 3,
    FMT_BOOL = 4,
    FMT_PTR = 5
} fmt_type_t;

typedef struct
{
    fmt_type_t type;

    union {
        int64_t i;
        uint64_t u;
        double d;
        const char *s;
        bool b;
        const void *p;
    } data;
} fmt_tag_t;
```

```
typedef struct
{
    char * heap_ptr;     // NULL if using internal_buf
    size_t length;       // Current string length (excluding NUL)
    size_t capacity;     // Current buffer size (internal, or heap)
    fmt_result_t result;
    char internal_buf[FMT_CAPACITY];
} fmt_string_t;
```

4.4 Success and Error handling

`fmt_string_t` contains the `result` member. A formatted string can be obtained by calling `fmt_result()`.

5. Limitations and Future Improvements

The intention is to maintain full compatibility with the C++ design and fully support the same capabilities. That is the full format specification, including width and specifiers.

However, the analogous functions `fmt_vprint()`, `fmt_vfprint()`, `fmt_vformat()` are not compatible with this design because `va_list` does not preserve the type information that the `_Generic()` based type-safe design requires.

6. Constraints

The `format` argument shall not be a NULL pointer. If format is a NULL pointer, no characters are written, the `result` member is set to `FMT_ERROR_NULL`. Likewise for the inspection and clean up functions, if buf is NULL, the `result` member is set to `FMT_ERROR_NULL`.

Where the formatting of a string is not possible, the output is not prepared, there is no truncation. An error is recorded in the `result` member.

7. Acknowledgments

P0645 - C++ {fmt} Text Formatting by Victor Zverovich.

8. References

Sample {fmt} implementation fmt.h and fmt.c
https://github.com/jonnygrant/open/fmt

C++ {fmt} Text Formatting
https://wg21.link/P0645

N3950 compile_assert()
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3950.pdf

Appendix A
Sample program and output

```
fmt_result_t result;

fmt_string_t buf;
buf = fmt_format("Hello {} newline\n", "planet");
printf("%s", fmt_string_data(&buf));

fmt_print("Again {}", fmt_string_data(&buf));

fmt_print("pi = {}\n", 3.14159);

// free any heap that was allocated
fmt_free(&buf);

fmt_string_t my_buf = FMT_INIT;
my_buf = fmt_format("Hello {}", "London");
printf("%s\n", fmt_string_data(&my_buf));

result = fmt_print("Hello {}\n", "world");
if(FMT_OK != result) printf("fmt_print err: %d\n", result);

result = fmt_print("int = {}\n", 42);
if(FMT_OK != result) printf("fmt_print err: %d\n", result);

result = fmt_print("Mixed strings: {} {} {}\n", "name", 10, 1.25);
if(FMT_OK != result) printf("fmt_print err: %d\n", result);

result = fmt_fprint(stderr, "stderr {}\n", "foo");
if(FMT_OK != result) printf("fmt_fprint err: %d\n", result);

result = fmt_fprint(stderr, "NULL {}\n", NULL);
if(FMT_OK != result) printf("fmt_fprint err: %d\n", result);

result = fmt_fprint(stderr, "nullptr {}\n", nullptr);
if(FMT_OK != result) printf("fmt_fprint err: %d\n", result);

result = fmt_fprint(stderr, NULL, nullptr);
if(FMT_OK != result) printf("fmt_fprint(stderr, NULL, nullptr) err:
%d\n", result);

result = fmt_fprint(stderr, NULL, NULL);
if(FMT_OK != result) printf("fmt_fprint(stderr, NULL, NULL) err: %d\
n", result);

// Show malformed floating point
const float test = (float)0xFFFFFFFF;
result = fmt_print("float: {}\n", test);
if(FMT_OK != result) printf("fmt_print err: %d\n", result);

const float test2 = NAN;
```

```
result = fmt_print("float: {}\n", test2);
if(FMT_OK != result) printf("fmt_print err: %d\n", result);

result = fmt_clear(&buf);
if(FMT_OK != result) printf("fmt_print err: %d\n", result);

char temp = {0};
void * vptr = &temp;
result = fmt_print("void*: {}\n", vptr);
if(FMT_OK != result) printf("vptr err: %d\n", result);
```

```
$ ./main.bin
Hello planet newline
Again Hello planet newline
pi = 3.14159
Hello London
Hello world
int = 42
Mixed strings: name 10 1.25
stderr foo
NULL (nil)
nullptr (nil)
fmt_fprint(stderr, NULL, nullptr) err: 6
fmt_fprint(stderr, NULL, NULL) err: 6
float: 4.29497e+09
float: nan
void*: 0x7fff342e3c5f
```