Imports cannot ...

C-style variadic functions in modular C++

Document #:	P3550R0
Date:	2025-01-13
Project:	Programming Language C++
Audience:	Evolution Incubator
	Library Incubator

Reply-to:	Alisdair Meredith
	<ameredith1@bloomberg.net>

Contents

1 Abstract

Macros cannot be exported from a C++ module. This presents a problem for modular C++, which relies on importing the Standard Library module `std` to access Standard Library functionality. In particular, when trying to access the arguments to a C style variadic function, a core language functionality, the user is further required to `#include <cstdarg>` or `#include <stdarg.h>`, as neither header is importable.

This paper reviews the value and risks of supporting such variadic functions in the modern modular C++ world, paying particular attention to the growing concerns about secure coding and the risks associated with this kind of function. Considering several approaches to resolving the concerns raised above, we recommend removing support for C-style variadic functions from C++ at the earliest opportunity, and deprecating any parts that we cannot immediately remove.

2 Revision history

R0 January 2025 (pre-Hagenberg mailing)

Initial draft of this paper.

3 Introduction

Macros cannot be exported from a C++ module. This presents a problem when trying to access the arguments to a C style variadic function while relying on importing the Standard Library module `std` to access library functionality. This paper proposes turning the relevant C macros into keywords, to remove the macro dependency from a core language facility.

Further, C-style variadic functions have been identified as an attack vector for security vulnerabilities in C and C++ programs. It would be desirable to scale back or entirely remove support for these facilities in C++, rather than extend the language to support their use in modular code.

4 Addressing the `import` Problem

4.1 Remove C-style var-arg functions from C++

Where we say *remove*, we mostly mean *deprecate* in C++26 with a stated goal to phase in removal in subsequent standards. However, we can address the s

C-style variadic function are very prone to undefined behavior, potentially exploitable as a security vulnerability, and essentially supplanted by variadic function templates. Our preference should be to remove them from the language.

We must retain a level of C-compatibility support, which suggests supporting the declaration of var-arg functions only in the global namespace with `extern "C"` linkage.

We will need to maintain support for var-arg declarations in C++ code much longer than their definition, as they are used in popular SFINAE techniques as a default lowest ranked function match in overload resolution. However, such functionality can be addressed slowly through the more traditional approach of a period of deprecation before removal. The SFINAE functionality is no longer needed as `requires` clauses are a superior alternative.

5 Addressing the Security Concerns

Variadic functions push a different number of arguments onto the calling stack frame for each call — although this is fully determined at compile-time and not at run-time.

Interpretation of the arguments passed in the stack memory is entirely dependant on runtime analysis, including guessing the size of type of each argument. This part is extremely error prone and may easily lead into undefined behavior.

There have been various attempts to document and address these risks from the C language in the past, notably in the work on the [HexVASAN] code sanitizer.

Notable risks arising from supporting this syntax are:

— Open to attacks through the function call stack
— Loss of type safety leading to undefined behavior

6 Remediation

6.1 Variadic templates

C++ variadic templates are type-safe, and guaranteed to process the correct number of arguments. They decode parameter type at compile-time, not runtime, potentially saving some computation.

We have a proof-of-concept for the superior reach of C++ variadic templates solving the same problem as C-style var-args in the C++ Standard Library `print` facility that is a superior drop-in replacement for the C Library `printf` family of functions.

6.2 Library replacement

After an audit of the C23 Library, these are the only functions that depend on variadic functionality.

— `printf` family of functions
— `scanf` family of functions

In `<cstdio>`

```
int fprintf(FILE* stream, const char* format, ...);
int fscanf(FILE* stream, const char* format, ...);
int printf(const char* format, ...);
int scanf(const char* format, ...);
int snprintf(char* s, size_t n, const char* format, ...);
int sprintf(char* s, const char* format, ...);
int sscanf(const char* s, const char* format, ...);
int vfprintf(FILE* stream, const char* format, va_list arg);
int vfscanf(FILE* stream, const char* format, va_list arg);
int vprintf(const char* format, va_list arg);
int vscanf(const char* format, va_list arg);
int vsnprintf(char* s, size_t n, const char* format, va_list arg);
int vsprintf(char* s, const char* format, va_list arg);
int vsscanf(const char* s, const char* format, va_list arg);
```

In `<cwchar>`

```
int fwprintf(FILE* stream, const wchar_t* format, ...);
int fwscanf(FILE* stream, const wchar_t* format, ...);
int swprintf(wchar_t* s, size_t n, const wchar_t* format, ...);
int swscanf(const wchar_t* s, const wchar_t* format, ...);
int vfwprintf(FILE* stream, const wchar_t* format, va_list arg);
```

```
int vfwscanf(FILE* stream, const wchar_t* format, va_list arg);
int vswprintf(wchar_t* s, size_t n, const wchar_t* format, va_list arg);
int vswscanf(const wchar_t* s, const wchar_t* format, va_list arg);
int vwprintf(const wchar_t* format, va_list arg);
int vwscanf(const wchar_t* format, va_list arg);
int wprintf(const wchar_t* format, ...);
int wscanf(const wchar_t* format, ...);
```

6.2.1 Print functionality

`std::print` handles narrow and wide characters, print to stream (including `stringstream`), and print to C file.

There is no need to emulate the `v` variants in C that take `va_list` arguments since that functionality is deliberately not supported.

6.2.2 Scan functionality

We have no drop-in replacement in the standard library yet. The `<iostream>` family of headers remains our best supported alternative, having its detractors along with its benefits.

— folks may prefer `scanf` for the same reason they preferred `printf`
— handling errors is more approachable for a newcomer than `scanf`

The `<regex>` header provides another feature that supports parsing text, but requires considerably more learning than a simple replacement for reading a pattern through a `scanf` string.

7 Design Principles

— Design for "safety" in C++ by reducing the range of undefined behavior in the Standard
— Design for "safety" in C++ by reducing type-unsafe access to the program stack
— Ensure compatibility with C-language interfaces, notably operating system APIs
— Avoid the breakage of existing "safe" code
— Minimize the breakage of existing "unsafe" code

8 Proposed Solution

This paper proposes two basic changes:

— Deprecate the declaration of functions using C-style `...` var-args unless the function is in the global namespace has `extern "C"` linkage. This supports the ability to *call* regular C functions, including the `printf` and `scanf` families of functions in the C standard library. Notes that the `printf` and `scanf` families of functions in namespace `std`, as supplied by the headers `<cstdio>` and `<cwchar>` are implicitly deprecated by this change, but might want to be formally deprecated and moved to Annex D by the LWG.
— Remove the header `<cstdarg>` and `<stdarg.h>` from the C++ Standard Library. This change removes the ability for C++ programs accessing the variable arguments on the program stack in a function definition. Functions that do not access such arguments remain well-defined, but deprecated by implication as the function declaration that is being defined is deprecated. Note that this change would mean that there is a subset of C programs that could not be "just recompiled as C++" for safer type analysis, as we would no longer support defining var-arg functions. We accept this, rather than continuing to support the C header `<stdarg.h>` as part of the cost of securing C++.

A less ambitious proposal might simply deprecate the `<cstdarg>` and `<stdarg.h>` headers, allowing time for the deprecation of function declarations to be processed by the community. This approach leaves open the security hole that we are trying to close, although that may still be addressed by security profiles, see [P3081R0] section 3.8.

9 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5001], the latest draft at the time of writing.

This drafting implements the strongest proposal — the complete removal of support for C-style variadic arguments. It is expected that some weaker resolution will be adopted, and this guideline wording highlights all the text in the Standard that will likely be affected, requiring some rewrites.

7.7 [expr.const] Constant expressions

10 An expression E is a *core constant expression* unless the evaluation of E, following the rules of the abstract machine (6.9.1 [intro.execution]), would evaluate one of the following:

(10.1) — `this` (7.5.3 [expr.prim.this]), except
(10.1.1) — in a constexpr function (9.2.6 [dcl.constexpr]) that is being evaluated as part of E or
(10.1.2) — when appearing as the *postfix-expression* of an implicit or explicit class member access expression (7.6.1.5 [expr.ref]);
(10.2) — …
(10.28) — an *asm-declaration* (9.10 [dcl.asm]);
(10.29) — ~~an invocation of the `va_arg` macro (17.13.2 [cstdarg.syn]);~~
(10.30) — a non-constant library call (3.35 [defns.nonconst.libcall]); or
(10.31) — a `goto` statement (8.7.6 [stmt.goto]).

[*Note 4:* A `goto` statement introduced by equivalence (8 [stmt.stmt]) is not in scope. For example, a `while` statement (8.6.2 [stmt.while]) can be executed during constant evaluation. —*end note*]

11 It is implementation-defined whether E is a core constant expression if E satisfies the constraints of a core constant expression, but evaluation of E has runtime-undefined behavior.

12 It is unspecified whether E is a core constant expression if E satisfies the constraints of a core constant expression, but evaluation of E would evaluate an operation that has undefined behavior as specified in 16 [library] through 33 [exec].

(12.1) — an operation that has undefined behavior as specified in 16 [library] through 33 [exec] or
(12.2) — an invocation of the `va_start` macro 17.13.2 [cstdarg.syn])].

9.3.4.6 [dcl.fct] Functions

2 The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called.

[*Note 1:* The *parameter-declaration-clause* is used to convert the arguments specified on the function call; see 7.6.1.3 [expr.call]. —*end note*]

If the *parameter-declaration-clause* is empty, …

…

[*Example 1:* The declaration …

However, the first argument must be of a type that can be converted to a `const char*`. —*end example*]

[*Note 2:* The standard header `<cstdarg>` (17.13.2 [cstdarg.syn]) contains a mechanism for accessing arguments passed using the ellipsis (see 7.6.1.3 [expr.call] and 17.13 [support.runtime]). —*end note*]

16.4.2.3 [headers] Headers

3 The facilities of the C standard library are provided in the additional headers shown in Table 25.[1]

[1]It is intentional that there is no C++ header for any of these C headers: `<stdarg.h>`, `<stdnoreturn.h>`, `<threads.h>`.

Table 25 — C++ headers for C library facilities [tab:headers.cpp.c]

`<cassert>` `<cctype>` `<cerrno>` `<cfenv>` `<cfloat>` `<cinttypes>` `<climits>` `<clocale>` `<cmath>` `<csetjmp>` `<csignal>` ~~`<cstdarg>`~~ `<cstddef>` `<cstdint>` `<cstdio>` `<cstdlib>` `<cstring>` `<ctime>` `<cuchar>` `<cwchar>` `<cwctype>`

6 Names which are defined as macros in C shall be defined as macros in the C++ standard library, even if C grants license for implementation as functions.

[*Note 2:* The names defined as macros in C include the following: `assert`, `offsetof`, and `setjmp`~~, `va_arg`, `va_end`, and `va_start`~~. —*end note*]

16.4.2.4 [std.modules] Modules

6 *Recommended practice*: Implementations should avoid exporting any other declarations from the C++ library modules.

[*Note 2:* Like all named modules, the C++ library modules do not make macros visible (10.3 [module.import]), such as `assert` (19.3.2 [cassert.syn]), `errno` (19.4.2 [cerrno.syn]), and `offsetof` (17.2.1 [cstddef.syn]) ~~, and `va_arg` ([cstdarg.syn])~~. —*end note*]

16.4.3.2 [using.headers] Headers

Table 27 — C++ headers for freestanding implementations [tab:headers.cpp.fs]

Subclause		Header
17.11	Comparisons	`<compare>`
17.12	Coroutines support	`<coroutine>`
17.13	Other runtime support	~~`<cstdarg>`~~`<csetjmp>`, `<csignal>`
Clause 18	Concepts library	`<concepts>`

17.1 [support.general] General

Table 42 — Language support library summary [tab:support.summary]

Subclause		Header
17.11	Comparisons	`<compare>`
17.12	Coroutines support	`<coroutine>`
17.13	Other runtime support	`<csetjmp>`, `<csignal>`, ~~`<cstdarg>`,~~ `<cstdlib>`

16.4.5.3.4 [extern.names] External linkage

2 Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.[2]

17.13 [support.runtime] Other runtime support

17.13.1 [support.runtime.general] General

1 Headers `<csetjmp>` (nonlocal jumps), `<csignal>` (signal handling), ~~`<cstdarg>` (variable arguments),~~ and `<cstdlib>` (runtime environment `getenv`, `system`), provide further compatibility with C code.

[2]The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>` (17.13.3 [csetjmp.syn]) ~~, and `va_end(va_list)`, declared or defined in `<cstdarg>` (17.13.2 [cstdarg.syn])~~.

17.13.2 [cstdarg.syn] Header <cstdarg> synopsis

```
// all freestanding

namespace std {
  using va_list = see below;
}

#define va_arg(V, P) see below
#define va_copy(VDST, VSRC) see below
#define va_end(V) see below
#define va_start(V, P) see below
```

1 The contents of the header <cstdarg> are the same as the C standard library header <stdarg.h>, with the following changes:

— In lieu of the default argument promotions specified in ISO/IEC 9899:2018 6.5.2.2, the definition in 7.6.1.3 applies.

— The restrictions that C places on the second parameter to the va_start macro in header <stdarg.h> are different in this document. The parameter parmN is the rightmost parameter in the variable parameter list of the function definition (the one just before the ...).[3] If the parameter parmN is a pack expansion (13.7.4 [temp.variadic]) or an entity resulting from a lambda capture (7.5.6 [expr.prim.lambda]), the program is ill-formed, no diagnostic required. If the parameter parmN is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.

SEE ALSO: ISO/IEC 9899:2018, 7.16.1.1

17.14 [support.c.headers] C headers

17.14.1 [support.c.headers.general] General

Table 44 — C headers [tab:c.headers]

<assert.h> <inttypes.h> <signal.h> <stdckdint.h> <tgmath.h> <complex.h> <iso646.h> <stdalign.h> <stddef.h> <time.h> <ctype.h> <limits.h> ~~<stdarg.h>~~ <stdint.h> <uchar.h> <errno.h> <locale.h> <stdatomic.h> <stdio.h> <wchar.h> <fenv.h> <math.h> <stdbit.h> <stdlib.h> <wctype.h> <float.h> <setjmp.h> <stdbool.h> <string.h>

28.7.3 [cwchar.syn] Header <cwchar> synopsis

```
namespace std {                                          // freestanding
  using size_t = see 17.2.4[support.types.layout];       // freestanding
  using mbstate_t = see below;                           // freestanding
  using wint_t = see below;                              // freestanding

  struct tm;
```

```
  int fwprintf(FILE* stream, const wchar_t* format, ...);
  int fwscanf(FILE* stream, const wchar_t* format, ...);
  int swprintf(wchar_t* s, size_t n, const wchar_t* format, ...);
  int swscanf(const wchar_t* s, const wchar_t* format, ...);
  int vfwprintf(FILE* stream, const wchar_t* format, va_list arg);
  int vfwscanf(FILE* stream, const wchar_t* format, va_list arg);
  int vswprintf(wchar_t* s, size_t n, const wchar_t* format, va_list arg);
```

[3] ~~193) Note that va_start is required to work as specified even if unary operator& is overloaded for the type of parmN.~~

```
  int vswscanf(const wchar_t* s, const wchar_t* format, va_list arg);
  int vwprintf(const wchar_t* format, va_list arg);
  int vwscanf(const wchar_t* format, va_list arg);
  int wprintf(const wchar_t* format, ...);
  int wscanf(const wchar_t* format, ...);
```

```
  wint_t fgetwc(FILE* stream);
  wchar_t* fgetws(wchar_t* s, int n, FILE* stream);
  wint_t fputwc(wchar_t c, FILE* stream);
  int fputws(const wchar_t* s, FILE* stream);
  int fwide(FILE* stream, int mode);

  // …

  // 28.7.5[c.mb.wcs], multibyte / widestring and character conversion functions
  int mbsinit(const mbstate_t* ps);
  size_t mbrlen(const char* s, size_t n, mbstate_t* ps);
  size_t mbrtowc(wchar_t* pwc, const char* s, size_t n, mbstate_t* ps);
  size_t wcrtomb(char* s, wchar_t wc, mbstate_t* ps);
  size_t mbsrtowcs(wchar_t* dst, const char** src, size_t len, mbstate_t* ps);
  size_t wcsrtombs(char* dst, const wchar_t** src, size_t len, mbstate_t* ps);
}

#define NULL see 17.2.3[support.types.nullptr]// freestanding
#define WCHAR_MAX see below // freestanding
#define WCHAR_MIN see below // freestanding
#define WEOF see below // freestanding
```

31.13.1 [cstdio.syn] Header <cstdio> synopsis

```
namespace std {
  using size_t = see 17.2.4[support.types.layout];
  using FILE = see below ;
  using fpos_t = see below ;
}

#define NULL see 17.2.3
#define _IOFBF see below
#define _IOLBF see below
#define _IONBF see below

// …


#define stderr see below
#define stdin see below
#define stdout see below

namespace std {
  int remove(const char* filename);
  int rename(const char* old_p, const char* new_p);
  FILE* tmpfile();
  char* tmpnam(char* s);
```

```
  int fclose(FILE* stream);
  int fflush(FILE* stream);
  FILE* fopen(const char* filename, const char* mode);
  FILE* freopen(const char* filename, const char* mode, FILE* stream);
  void setbuf(FILE* stream, char* buf);
  int setvbuf(FILE* stream, char* buf, int mode, size_t size);
```

```
  int fprintf(FILE* stream, const char* format, ...);
  int fscanf(FILE* stream, const char* format, ...);
  int printf(const char* format, ...);
  int scanf(const char* format, ...);
  int snprintf(char* s, size_t n, const char* format, ...);
  int sprintf(char* s, const char* format, ...);
  int sscanf(const char* s, const char* format, ...);
  int vfprintf(FILE* stream, const char* format, va_list arg);
  int vfscanf(FILE* stream, const char* format, va_list arg);
  int vprintf(const char* format, va_list arg);
  int vscanf(const char* format, va_list arg);
  int vsnprintf(char* s, size_t n, const char* format, va_list arg);
  int vsprintf(char* s, const char* format, va_list arg);
  int vsscanf(const char* s, const char* format, va_list arg);
```

```
  int fgetc(FILE* stream);
  char* fgets(char* s, int n, FILE* stream);
  int fputc(int c, FILE* stream);
  int fputs(const char* s, FILE* stream);

  // …

  int feof(FILE* stream);
  int ferror(FILE* stream);
  void perror(const char* s);
}
```

D [depr] Compatibility features

D.X [depr.varags.lib] C Library Variadic Functions

D.X.1 [depr.cstdio] Header <cstdio>

The header <cstdio> declares the following functions that are defined according to the C Library.

```
namespace std {
  int fprintf(FILE* stream, const char* format, ...);
  int fscanf(FILE* stream, const char* format, ...);
  int printf(const char* format, ...);
  int scanf(const char* format, ...);
  int snprintf(char* s, size_t n, const char* format, ...);
  int sprintf(char* s, const char* format, ...);
  int sscanf(const char* s, const char* format, ...);
  int vfprintf(FILE* stream, const char* format, va_list arg);
  int vfscanf(FILE* stream, const char* format, va_list arg);
  int vprintf(const char* format, va_list arg);
  int vscanf(const char* format, va_list arg);
  int vsnprintf(char* s, size_t n, const char* format, va_list arg);
  int vsprintf(char* s, const char* format, va_list arg);
```

```
  int vsscanf(const char* s, const char* format, va_list arg);
}
```

D.X.2 [depr.cwchar] Header <cwchar>

The header <cwchar> declares the following functions that are defined according to the C Library.

```
namespace std {
  int fwprintf(FILE* stream, const wchar_t* format, ...);
  int fwscanf(FILE* stream, const wchar_t* format, ...);
  int swprintf(wchar_t* s, size_t n, const wchar_t* format, ...);
  int swscanf(const wchar_t* s, const wchar_t* format, ...);
  int vfwprintf(FILE* stream, const wchar_t* format, va_list arg);
  int vfwscanf(FILE* stream, const wchar_t* format, va_list arg);
  int vswprintf(wchar_t* s, size_t n, const wchar_t* format, va_list arg);
  int vswscanf(const wchar_t* s, const wchar_t* format, va_list arg);
  int vwprintf(const wchar_t* format, va_list arg);
  int vwscanf(const wchar_t* format, va_list arg);
  int wprintf(const wchar_t* format, ...);
  int wscanf(const wchar_t* format, ...);
}
```

10 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

11 References

[HexVASAN] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. 2017. Venerable Variadic Vulnerabilities Vanquished. https://nebelwelt.net/files/17SEC.pdf

[N5001] Thomas Köppe. 2024-12-17. Working Draft, Programming Languages — C++. https://wg21.link/n5001

[P3081R0] Herb Sutter. 2024-10-16. Core safety Profiles: Specification, adoptability, and impact. https://wg21.link/p3081r0