```
Proposal for C23
WG14 2975

Title:               Relax requirements for variadic parameter lists, v3
Author, affiliation: Alex Gilding, Perforce Software
                     JeanHeyd Meneide, thephd.dev
Date:                2022-04-15
Proposal category:   Feature enhancement
Target audience:     Compiler/tooling developers
```

Abstract

C's variadic function features are leaky and expose outdated assumptions about the implementation of variadics. The requirements on variable argument lists are stricter than they need to be, in service of a restriction that no longer applies, introducing error surface and inconvenience to the user while leaking its assumptions about both.

C should relax the restrictions that are no longer needed, because doing so does not break any existing code and may open up new possibilities for the use of ellipsis notation in future, by freeing it from the final baggage of `varargs.h`.

Relax requirements for variadic parameter lists, v3

```
Reply-to:           Alex Gilding (agilding@perforce.com)
                    JeanHeyd Meneide (phdofthehouse@gmail.com)
```

```
Document No:        N2975
Revises Document No: N2919
Date:               2022-04-15
```

Summary of Changes

N2975

- Provide examples of usage without leading named argument. Note result of C++ liaison discussion. Retain wording options dependent on status of `__VA_OPT__`.

N2919

- Avoid “obsolescent”; provide alternative signatures for optional argument; provide more implementation experience

N2854

- original proposal

Introduction

C permits functions to be declared as accepting a variable number of arguments through the use of what the Standard refers to as "ellipsis notation". The type and number of arguments passed to a function through the ellipsis are unknown to the callee, and may be zero.

Currently, the syntax for function declarators requires that a parameter list either be empty, or consist of a list of *parameter-declaration* productions optionally terminated by an ellipsis. This is not a constraint; expressing a function declaration with a *parameter-type-list* that only contains an ellipsis is a syntax violation.

We propose that the syntax and constraints be relaxed slightly so that a function declaration's *parameter-type-list* may also consist of an ellipsis with no leading explicit *parameter-declarations*.

Rationale

There is an implicit assumption that in order to do anything with a variable argument list, the function being declared must have at least one named and typed parameter in order to provide some basic minimum amount of handling information, such as a format string (which encodes type and number), or at least an integer listing the number of arguments actually passed.

Since there is no mechanism provided to query a `va_list` for number of items, this is true from the perspective of usefully handling the content of the list in a function defined in C. However:

- externally linked functions may be defined in other languages with more information or more fine grained control over their management of the list, such as assembly;

- a function may reasonably consume arguments without processing them (such as a nop-overload or to signify intent to evaluate), in which case handling the argument values internally is not important and should not impose an unnecessary restriction on the signature.

In future, the unrestricted signature may also be useful for moving towards generic/unknown parameter types replacing the existing uses of the unspecified-parameter type declaration syntax (nothing between the parentheses).

There is also an insidious implicit assumption that in order to access the variable argument list from within the variadic function, the features provided by `stdarg.h` require a named parameter to use as an "anchor". This is either untrue, or exposes a misuse of undefined behaviour in the Standard Library API; in either case it is not necessary and the Standard should not impose this as a requirement.

This assumption appears to be a holdover from the predecessor to `stdarg.h`, [POSIX's `varargs.h`] (https://pubs.opengroup.org/onlinepubs/007908799/xsh/varargs.h.html). This header was designed with KnR-style function definitions in mind, and did not make use of the ellipsis notation at all. Instead, the macros `va_alist` and `va_dcl` abstracted the platform dependent positioning of parameters that the header could rely on as anchors. Consequently the version of `va_start` provided by `varargs.h` does not require an explicit anchor, because the declaration macros attempted to abstract this and hide the use of undefined behaviour from the user. The ellipsis notation removes the ability of the header to hide this use of an anchor. This method of implementing variadic functions goes back to and generalizes the older method of simply providing more argument space in the definition than the caller necessarily needed to fill, because KnR function definitions did not match a prototype anyway. This method is seen in KnR first edition, and even as far back as the ancient B manuals.

This betrays an assumption that the calling convention will *always* be linear and stack-based; this assumption is no longer true now that the language is able to make parameter types visible to the caller.

In addition, the current wording also exposes the assumption that UB will be used to walk from one argument to another in its requirement that the anchor parameter have an addressable object type (not `register`, not function or array), or else the behaviour is undefined. This is an unnecessary constraint on platforms that are able to use register-based calling conventions.

There is prior art for the change: declaring a function with a variable argument list and no preceding named parameters is legal in C++ ([dcl.fct] p3).

In practice, the anchor is not used on many modern Standard Library header implementations: for instance on x64, the arguments are still passed in registers even to a variadic function. Most headers simply forward to a compiler magic builtin such as `__builtin_va_start`. Given:

```
extern int foo (int, ...);

int bar (int a, int b, int c) {
  return foo (0, 1, 2, 3);
}
```

GCC for x64 will generate:

```
bar:
    push    rbp
    mov     rbp, rsp
    mov     ecx, 3
    mov     edx, 2
    mov     esi, 1
    mov     edi, 0
    mov     eax, 0
    call    foo
    pop     rbp
    ret
```

i.e. all four arguments are passed in registers. The implementation of `bar` therefore cannot actually use undefined behaviour to walk-off a pointer to the anchor argument, because it hasn't got one. At least some absolute minimum level of compiler support is needed.

Finally, it is logically impossible for a compiler to support named parameters at all, but to not know the location of the start of the variable argument list. The assumption that an anchor needs to be specified comes from the feature's origins as a header-only solution that pre-dates compiler awareness of variadic functions altogether. In the presence of ellipsis notation, allowing the user to manually specify an anchor achieves nothing except to potentially create an obscure bug should the user make a mistake and provide the compiler with an *inaccurate* version of information it should already have perfect access to.

If the calling convention (e.g. `cdecl`) does pass all arguments in a linear stack-based order, then using undefined-behaviour address-walking in the header remains a valid implementation decision. However, we believe this decision should be hidden from the user, and the anchor should be provided by the compiler if one is used.

Proposal

We propose a relaxation of the existing rules that will leave all current C code valid with no changes required, but will allow new C23 code to avoid touching the details of obsolete `varargs.h` implementations (that no longer reflect reality).

- the syntax of function declarations should add the option for the *parameter-type-list* to consist only of an ellipsis, with no commas or other parameters.
- the `va_start` macro should specify that only the first argument is used. The second argument is optional and may be written for backward compatibility purposes. The Standard should specify that the second argument to `va_start` is **never expanded** and therefore is not used or evaluated in any way. It may be elided and `va_start` used with only one parameter, as it was in its original POSIX/`varargs.h` form.

Alternatives

No direct alternative is proposed because the relaxation does not break any existing user code.

In the longer term we expect C to move away from providing variadic functions in the Standard Library. However, the language will need to continue supporting the declaration syntax so that it can make external calls to functions implemented in assembly or other languages with variadic support; and if the declaration syntax is supported there is no reason to take the feature away from the user within definitions either. We therefore do not propose deprecation of variadic functions as an alternative solution and would vote to oppose their removal.

Impact

There is no impact to correct code in specifying that the optional second argument to `va_start` is never evaluated. Since the current wording requires that the second argument be the *identifier* of a parameter, it is not expected to be able to express a side effect as an expression by any means except possibly `volatile` access. Implicitly the `volatile` access will not occur since the user understands the parameter to be being used for its address in the current version of the library, so a user relying on an access here is already relying on implicitly unspecified behaviour.

All current variadic function declarations will remain valid because there is no implied requirement to *not* provide typed and named parameters before the ellipsis.

All current variadic function definitions in C will remain valid because there is no need to force `va_start` to only take one argument. It is already expected to have macro-like semantics.

This change is necessary but **not sufficient** to allow ellipsis notation to replace the empty parentheses notation used to declare a function or function pointer as accepting "any" parameters. On some platforms the presence of the ellipsis implies a different calling convention from "any" parameters, as the ellipsis can choose to force the linear stack-based layout while the "any" may use a register-based convention that expects only one true underlying signature type.

However, by removing the restriction that the ellipsis is preceded by at least one explicitly typed parameter, we come closer to unifying these two features in another paper. Implementers may also choose to use a completely new ABI for functions which have no named parameter and then an ellipsis. We anticipate this will be used to replace the purpose of removed KnR declarations, and therefore can use that ABI. The standard does not talk about such things, but the ABI space is open to such implementations, and therefore it is possible for an implementation to support the semantics they would like, without waiting for further unification papers from WG14.

There is a small impact to compiler and library developers in that finding the start of the variable argument list now requires implementation support. In practice this support is already widespread. We do not believe this poses any substantial implementation burden, because at most one new magic compiler builtin is required, `__builtin_get_va_start_loc` or something similar. The compiler only needs to expose the virtual name it would already have generated for the ellipsis parameter. More complex calling conventions already require in-depth compiler support and will not change their behaviour at all.

Discussion with the C++ liaison subgroup SG22 indicated that there is no conflict between this proposal and the feature as-imported by C++. A followup paper for C++ relaxing the restriction on `va_start` would be helpful but is not required for acceptance by C, and the change to the C wording does not conflict with the specification as it is currently imported by C++.

Implementation experience

As above, declaring a function with a variable argument list and no preceding named parameters is legal in C++ ([dcl.fct] p3). The list is not accessible in-language but makes the function available for overload resolution.

The user may find another way to pass type and number information to make use of the argument list. At present no such code is expressible in conforming C. Functions written in other languages (most likely, assembly) will find their own ways to access and interpret the argument list that are out of scope for the C Standard.

A pure library implementation of access to the variadic arguments from a function with no named parameters can be found at: https://ztdvargs.readthedocs.io/en/latest/

It is not intended for production use, but demonstrates that on x64 and similar targets the arguments are easy to extract without a named start for UB-walking. A compiler would be expected to do a much better job here, since it would have access to `__builtin_ellipsis_param()` or some equivalent. A compiler will always know the location of the implicit ellipsis parameter, even if it doesn't expose a name to user scope.

We consider the nature of the library implementation to sufficiently justify that this feature needs to exist inside the compiler.

Proposed Wording

Changes are proposed against the wording in C2x draft n2596. Bolded text is new text.

Modify 6.5.2.2 "Function calls" paragraph 6 to delete the comma before the ellipsis:

> ends with an ellipsis (`...`) or the

Modify 6.5.2.2 "Function calls" paragraph 7 to indicate that there may not have been a preceding parameter:

> The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter **, if present**.

Modify 6.7.6 "Declarators" to add a third option to the *parameter-type-list* syntax rule:

> *parameter-type-list*:
> *parameter-list*
> *parameter-list* `, ...`
> `...`

Modify 6.7.6.3 "Function declarators" paragraph 14 to refer to a **final ellipsis** rather than to an "ellipsis terminator", reducing the strength of the wording.

There is no need to modify 6.7.6.3 Example 3.

Modify 7.16 "Variable arguments `<stdarg.h>`" paragraph 2 to remove the reference to *parmN*, deleting all but the first sentence:

> A function may be called with a variable number of arguments of varying types **if its *parameter-type-list* ends with an ellipsis.**

Modify 7.16.1.1 "The `va_arg` macro" paragraph 3 to remove the reference to *parmN*:

> The first invocation of the `va_arg` macro after that of the `va_start` macro returns the value of the **first argument without an explicit parameter, which matches the position of the `...` in the parameter list**. Successive invocations return the values of the remaining arguments in succession.

Modify 7.16.1.4 "The `va_start` macro", removing references to *parmN* and explaining that the second argument is optional and unused.

Paragraph 1:

```
#include <stdarg.h>
void va_start(va_list ap, ...);
```

Delete paragraph 4 entirely and replace:

> **Only the first argument passed to `va_start` is evaluated. Any additional arguments are not used by the macro and will not be expanded or evaluated for any reason.**
>
> **NOTE: The macro allows additional arguments to be passed to `va_start` for compatibility with older versions of the library only.**

Modify examples 1 and 2 to remove reference to the `parmN` argument:

```
if (n_ptrs > MAXARGS)
  n_ptrs = MAXARGS;
va_start(ap);
while (ptr_no < n_ptrs)
...

if (n_ptrs > MAXARGS)
  n_ptrs = MAXARGS;
va start(ap);
while (ptr_no < n_ptrs) {
...
```

Add a new example 3 demonstrating a possible use of a function with no named parameters before the ellipsis:

> **EXAMPLE 3** The function `f5` is similar to `f1`, but instead of passing an explicit number of strings as the first argument, the argument list is terminated with a null pointer.

```
#include <stdarg.h>
#define MAXARGS 31

void f5(...)
{
  va_list ap;
  char *array[MAXARGS];
  int ptr_no = 0;
  va_start(ap);
```

```
    while (ptr_no < MAXARGS)
    {
      char *ptr = va_arg(ap, char *);
      if (!ptr)
        break;
      array[ptr_no++] = ptr;
    }
    va_end(ap);
    f6(ptr_no, array);
  }
```

Each call to `f5` is required to have visible the definition of the function or a declaration such as

```
 void f5(...);
```

and implicitly requires the last argument to be a null pointer.

Modify B.15 "Variable arguments `<stdarg.h>`" to remove mention of *parmN*:

```
type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, ...);
```

Delete the undefined behaviour from J.2 (number ???) that links back to (7.16.1.4) as this text and the associated undefined behaviour are removed from the main section.

Optionally, if the changes specified in N2856 “Comma omission and comma deletion” are **not** accepted, update 7.16.1.4 and B.15 to use single-parameter syntax, (less clear but conforming):

Modify 7.16.1.4 "The `va_start` macro", removing references to `ap` and referring obliquely to the first argument:

Paragraph 1:

```
#include <stdarg.h>
void va_start(...);
```

Paragraph 3:

> The `va_start` macro initializes **the object named in its argument** for subsequent use by the `va_arg` and `va_end` macros. Neither the `va_start` nor `va_copy` macro shall be invoked to reinitialize the object without an intervening invocation of the `va_end` macro for the same argument.

In B.15:

```
void va_start(...);
```

Optionally, depending on the committee’s understanding of “obsolescent”, we could also change:

Add a new section in 7.31 "Future library directions" after 7.31.9:

> **7.31.10 Variable arguments** `<stdarg.h>` The ability to pass a second argument to the `va_start` macro is an obsolescent feature.

This should be voted on separately if there is sentiment to mark a feature as *obsolescent* even though it was necessary practice (not just optional or poor style, but required to use it at all) in previous language versions. If there is no sentiment to retroactively indicate that a required usage in previous versions is poor style in C23, we should not adopt this optional change.

(This optional change was not considered in the first round of discussion.)

References

- C23 n2731
- C++17 n4659
- KnR First Edition
- B printf
- varargs.h

- Köppe, Comma omission and comma deletion