N3359 (p1): `stdarg.h` wording...

`stdarg.h`, especially in C2x, is byzantine.
Modernising the language can alleviate this.

наб, seb, rCs

N3359 (p1): `stdarg.h` wording...

`stdarg.h`, especially in C2x, is byzantine.
Modernising the language can alleviate this.

наб, seb, rCs

Document #: 3359
Date: 2024-09-27
Revisions: N3285
Project: Programming Language C
Reply-to: наб <nabijaczleweli@nabijaczleweli.xyz>

1. Casus belli

seb <@sebastian@jittr.click> had identified a series of inconsistencies both in the wording of **`stdarg.h`** in the current draft C2X standard N3301 and in compilers' interpretations thereof. These have been refined in subsequent discussion, this paper presents a summary of diffs, along with rationales.

Comments on the previous revision of this paper echo the desire to standardise the nomenclature — the subclause referred to the same concept ad lib as "varying arguments" and "unnamed arguments", i.a. compound nouns thereof — the concept of a "variadic functions" with "varying arguments" is defined and replaced greedily,

2. Proposed wording

2.1. 7.16.1 (`<stdarg.h>`, `...`)

1 The header `<stdarg.h>` declares a type and defines five macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.
3 A function may be called with a variable number of arguments of varying types if its parameter type list ends with an ellipsis.

replace with

1 The header `<stdarg.h>` declares a type and defines five macros and functions, for use with variadic functions. Variadic functions accept a list of arguments whose number and types are not known to the called function when it is translated.
3 A function is variadic if its parameter type list ends with an ellipsis. Its varying arguments are those whose positions match or come after the ellipsis in the parameter list.

2.1.1. Rationale

As above, so below. Also, clearly note that some of these can be either, not *just* macros.

*2.2. 7.16.1 (**`va_list`**)*

4 The type declared is

    `va_list`

which is a complete object type suitable for holding information needed by the macros **`va_start`**, **`va_arg`**, **`va_end`**, and **`va_copy`**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as `ap` in this subclause) having type **`va_list`**. The object `ap` may be passed as an argument to another function; if that function invokes the **`va_arg`** macro with parameter `ap`, the representation of `ap` in the calling function is indeterminate and shall be passed to the **`va_end`** macro prior to any further reference to `ap`.[292] Whether a byte copy of **`va_list`** can be used in place of the original is implementation-defined.

Replace

> **`va_arg`**, **`va_end`**, and **`va_copy`**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as `ap` in this subclause) having type **`va_list`**.

with

> **`va_arg`**, **`va_end`**, and **`va_copy`** to access the varying arguments. Objects of type **`va_list`** are generally referred to as `ap` in this subclause.

and replace

> The object `ap` may be passed as an argument to another function; if that function invokes the **`va_arg`** macro with parameter `ap`, the representation of `ap` in the calling function is indeterminate and shall be passed to the **`va_end`** macro prior to any further reference to `ap`.[292)]

with

> If an `ap` object is passed as an argument to another function and that function invokes the **`va_arg`** macro on `ap` then the representation of `ap` in the calling function is indeterminate and `ap` shall be passed to the **`va_end`** macro before being passed to any other **`va_...`** macros.

2.2.1. Rationale

Beside updating the ancient-style wording ("if … is desired, the function … shall"), it hinted at a restrixion of where **`va_list`**s may be created. There are none such.

"reference to" is clarified to be w.r.t. the other **`va_...`** macros exclusively. It's still a valid object, and it's entirely okay to take its address, for example.

2.3. 7.16.2

> **7.16.2 Variable argument list access macros**

replace with

> **7.16.2 Varying argument access macros**

2.3.1. Rationale

As above, so below.

2.4. 7.16.2.1

> 1 The **`va_start`** and **`va_arg`** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **`va_copy`** and **`va_end`** are macros or identifiers declared with external linkage. If a macro definition is suppressed to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the **`va_start`** and **`va_copy`** macros shall be matched by a corresponding invocation of the **`va_end`** macro in the same function.

Append:

> The **`va_list`** argument given to every macro defined in this subclause shall be an lvalue of this type or the result of array-to-pointer decay of such an lvalue.

and append or add footnote:

> For conciseness only, this subclause refers to **`va_copy`** and **`va_end`** just as "macros". This is to be understood as a short-hand, not as constraining only one of the possible implementations.

2.4.1. Rationale

This codifies existing practice, since the macros modify `ap`, allthewhile it must be allowed to be passed to *functions*, wherein **`va_list`** decays to a pointer if it's an array type, verbatim.

Kinda odd that it says these can be macros *or* symbols but then it calls them macros, innit. If it said "the **`va_end`** macro or symbol" then that would be worse though.

2.5. 7.16.2.2

3 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.

replace

3 first argument without an explicit parameter, which matches the position of the `...` in the parameter list.

with

3 first varying argument.

2.5.1. Rationale

As above, so below. The definition is moved to *7.16.1* (and all other prose calls the ellipsis the ellipsis and not `...` so that's standardised there as well).

2.6. 7.16.2.4

3 The **va_end** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the **va_start** macro, or the function containing the expansion of the **va_copy** macro, that initialized the **va_list** `ap`. The **va_end** macro may modify `ap` so that it is no longer usable (without being reinitialized by the **va_start** or **va_copy** macro). If there is no corresponding invocation of the **va_start** or **va_copy** macro, or if the**va_end** macro is not invoked before the return, the behavior is undefined.

replace

3 from the function whose variable argument list was referred to by the expansion of the **va_start** macro, or the function containing the expansion of the **va_copy** macro, that initialized the **va_list** `ap`.

with

3 from the variadic function whose varying arguments were referred to by the expansion of the **va_start** macro, or the function containing the expansion of the **va_copy** macro, that initialized `ap`.

2.6.1. Rationale

As above, so below. Also, nothing else fully-specifies "the **va_list** `ap`" since we define that that's what is meant generally, so flatten that out.

*2.7. 7.16.2.5 (**va_start**)*

2 The **va_start** macro shall be invoked before any access to the unnamed arguments.

replace with

2 The **va_start** macro may only be invoked in the block scope of a variadic function.

2.7.1. Rationale

There is no other way to access the varying arguments (pt. 3 defines the way **va_start** facilitates this) anyway, so this can be deleted.

Currently, the way this limits where the standard allows **va_start** to be invoked is strictly by domain error of the counterfactual (if there are no varying arguments). Can you use **va_start** if there is an ellipsis but no varying arguments were given? Yes. Does the current wording allow it? No, for the same reason.

Even then, this allows

```
void f(va_list ap, int [(va_start(ap), 1)], ...) { va_end(ap); }
```

which makes little sense, and yet GCC permits it, while Clang refuses it (`'va_start' cannot be used outside a function`). This limits **va_start** to the scopes where it's meaningful.

2.8. 7.16.2.5 (examples)

9 EXAMPLE 2 The function `f3` is similar, but saves the status of the variable argument list after the indicated number of arguments; after `f2` has been called once with the whole list, the trailing part of the list is gathered again and passed to function `f4`.

10 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.

replace

9 but saves the status of the variable argument list after the indicated number of arguments; after `f2` has been called once with the whole list, the trailing part of the list is gathered the argument list is terminated with a null pointer.

with

9 but saves the position in the varying arguments after the indicated number of arguments; after `f2` has been called once with all arguments, the trailing arguments are gathered varying arguments are terminated with a null pointer.

2.8.1. Rationale

As above, so below.

3. References

The seminal post: https://jittr.click/@sebastian/statuses/01HYYTSHPDNAFDNQSTXVXYSAY2
Joseph Myers' *Pre-DR#8: va_list objects* (as additional rationale for this paper's *2.4* diff 1): https://www.polyomino.org.uk/computer/c/pre-dr-8.txt

4. Notes

All comments for N3285 were applied, with the exception of replacing "being passed to any other va_... macros" with "referenced again". The rationale in *2.2.1* was expanded to further justify this change.

Contents