

V

va_arg A macro, defined in `stdarg.h`, that causes the next function argument, of the type *type*, to be retrieved from the variable argument list and returned as the value of the whole expression.

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

The `ap` argument must be the same as that passed to the corresponding `va_start` or `va_copy`. If there are no more arguments, or *type* is not the type of the next argument, the result is undefined.

__VA_ARGS__^{C99} An identifier that can appear only in the replacement list of a function-like macro that contains an ellipsis. It is replaced by the variable argument list to which it corresponds.

Here is an example of its use:

```
#define trace(...) printf(__VA_ARGS__)

trace("value = %d\n", value);
trace("x = %d, y = %d\n", x, y);
```

va_copy^{C99} A macro, defined in `stdarg.h`, that allows a `va_list` to be copied.

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

va_end A macro, defined in `stdarg.h`, that performs any necessary cleanup of variable argument list processing.

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

The `ap` argument must be the same as that passed to the corresponding `va_start`. Once you have called `va_end`, you must call `va_start` before attempting to reprocess the argument list; otherwise, the behavior is undefined. `va_end` must be invoked from the same function in which the corresponding `va_start` was invoked. *See also* `va_copy`.

validation The process whereby a compiler that claims conformance to a standard is actually tested for such conformance. This process is usually performed by an accredited agency of a government, a national standards body, or a recognized professional association.

va_list A type, defined in `stdarg.h`, that is suitable for holding information needed by the `stdarg.h` macros `va_start`, `va_arg`, `va_copy`, and `va_end`. It is defined in `stdarg.h`.

value preserving rule The rule by which expressions having type `unsigned int` bit-field, `unsigned char`, or `unsigned short` are converted to `int` (if their value can be represented); otherwise, they are converted to `unsigned int` when these “narrow” types are widened. This rule was not widely followed prior to the advent of Standard C, which now requires it to be used. *See also* conversion, integer type; unsigned preserving rule.

varargs.h A common nonstandard version of, and predecessor to, `stdarg.h`.

variable argument list *See* argument list, variable; ellipsis.

variable-length array^{C99} *See* array, variable-length.

variable name The name used to designate a named-object. Such names are called identifiers, which, in the case of variables, share the same name space as enumeration constants, functions, and typedef names.

va_start A macro, defined in `stdarg.h`, that prepares for processing of a variable argument list by initializing the `ap` (argument pointer) argument using the address of the last fixed parameter. `parmN` is the last of the list of fixed arguments.

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

`va_start` must be called before `va_arg` or `va_end` are used for a given variable argument list. If `register` is used with `parmN`, or `parmN` has type function or array, or `parmN` is a narrow type, the behavior is undefined. *See also* `va_copy`.

vertical-tab character One of the white space characters allowed in source text.

vertical-tab escape sequence An escape sequence, `\v`, which allows the vertical tab character to be represented in a string literal or character constant.

vfprintf^{C89} A function that is equivalent to `fprintf`, except that the argument list has been replaced by a variable argument list identified by `arg`, which has been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls).

```
#include <stdio.h>
int vfprintf(FILE * restrict stream,
             const char * restrict format, va_list arg);
```

The return value is the number of characters transmitted or a negative value on error. *See also* stdarg.h.

vfscanf^{C99} A function that is like **fscanf**, except that the variable argument list is replaced by a **va_list**.

```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE * restrict stream,
            const char * restrict format, va_list arg);
```

The **va_list** must have already been initialized by **va_start**. This function does not call **va_end**.

vwprintf^{C95} A function that is like **fprintf**, except that the variable argument list is replaced by a **va_list**.

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vwprintf(FILE * restrict stream,
             const wchar_t * restrict format, va_list arg);
```

The **va_list** must have already been initialized by **va_start**. This function does not call **va_end**.

vwscanf^{C95} A function that is like **fscanf**, except that the variable argument list is replaced by a **va_list**.

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vwscanf(FILE * restrict stream,
            const wchar_t * restrict format, va_list arg);
```

The **va_list** must have already been initialized by **va_start**. This function does not call **va_end**.

virtual A C++ keyword that is not part of Standard C. If you think you might wish to move C code to a C++ environment in the future you should refrain from using **virtual** as an identifier in new C code you write.

visibility of identifiers *See* linkage; scope.

VLA^{C99} *See* array, variable-length.

void The type of an expression that has no value. Obtained by calling a **void** function or by casting an expression to type **void**.

The keyword also may be used in two other unrelated contexts: as the only token inside a function prototype to indicate there are no arguments, or as part of the generic pointer type **void ***.

void * type^{C89} *See* void pointer.

void cast An explicit conversion of an expression to type **void**. Such a cast results in the value of that expression's being discarded. An expression of type **void** can be cast to type **void**, even though such a cast is vacuous.

The need for an explicit **void** cast is rare because failing to use the value of an expression results in that value being discarded anyway. However, there is one valuable application. If you are writing a macro that mimics what might otherwise be a **void** function, you should explicitly cast the macro definition to **void** to prohibit its value being incorrectly used; for example,

```
void swap(int, int);

#define swap(a, b) ((a) ^= (b), \
                   (b) ^= (a), (void) ((a) ^= (b)))
```

See also conversion, void type.

void expression An expression that has no value. It results from either having a function returning **void** or from having explicitly cast an expression to type **void**.

void in prototypes Two distinct and unrelated uses of the **void** keyword in a prototype. Consider the following example:

```
void f(void);
```

The first use of **void** indicates that function **f** is a **void** function and, as such, does not return a value. The second use of **void** (as introduced by C89) indicates that the function expects no arguments. If the second **void** were omitted, the declaration would indicate that the function expected an unknown number of arguments and the compiler could do no checking.

void pointer^{C89} A pointer type adopted from C++ to help implementers on systems where pointers are not all the same size and/or representation

(such as with word architectures) and to provide a generic pointer type. All standard library routines that previously used `char *` for such pointers now use `void *` (for example, `malloc`, `calloc`, and `memcpy`). A `void` pointer cannot be dereferenced directly or have arithmetic performed on it.

void type conversion *See* conversion, void type.

volatile^{C89} A keyword used as a type qualifier. It indicates that the object to which it applies is not owned entirely by this program. That is, it might be read or written asynchronously by some other program (or by an interrupt handler in the same program) or hardware device as well, and the optimizer had better not eliminate various accesses to this object. The `volatile` qualifier also may be applied to the underlying object in a pointer declaration. Consider the following cases:

```
char *nvpnvc;
char * volatile vpnvc;
volatile char *nvpvc;
volatile char * volatile vpvc;
```

`nvpnvc` is a non-volatile pointer to a non-volatile `char`. `vpnvc` is a volatile pointer to a non-volatile `char`. `nvpvc` is a non-volatile pointer to a volatile `char`. `vpvc` is a volatile pointer to a volatile `char`.

Declarations containing the `volatile` qualifier also may contain the `const` and `restrict` qualifiers.

VP rule *See* value preserving rule.

vprintf^{C89} A function that is equivalent to `printf`, except that the argument list has been replaced by a variable argument list identified by `arg`, which has been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls).

```
#include <stdio.h>
int vprintf(const char * restrict format, va_list arg);
```

The return value is the number of characters transmitted or a negative value on error. *See also* `stdarg.h`.

vscanf^{C99} A function that is like `scanf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

`vsnprintf`^{C99} A function that is like `snprintf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char * restrict s, size_t n,
              const char * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

`vsprintf`^{C89} A function that is equivalent to `sprintf`, except that the argument list has been replaced by a variable argument list identified by `arg`, which has been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls).

`vsscanf`^{C99} A function that is like `sscanf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * restrict s,
            const char * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

`vswprintf`^{C95} A function that is like `swprintf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t * restrict s, size_t n,
              const wchar_t * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

`vswscanf`^{C95} A function that is like `swscanf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <wchar.h>
int vswscanf(const wchar_t * restrict s,
             const wchar_t * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

`vwprintf`^{C95} A function that is like `wprintf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

`vwscanf`^{C95} A function that is like `wscanf`, except that the variable argument list is replaced by a `va_list`.

```
#include <stdarg.h>
#include <wchar.h>
int vwscanf(const wchar_t * restrict format, va_list arg);
```

The `va_list` must have already been initialized by `va_start`. This function does not call `va_end`.

◇ ◇ ◇