NAME

stdarg – handle variable argument list

SYNOPSIS

```
#include <stdarg.h>

va_list pvar;

void va_start(va_list pvar, parmN);

type va_arg(va_list pvar, type);

void va_end(va_list pvar);
```

DESCRIPTION

This set of macros allows portable procedures that accept variable numbers of arguments of variable types to be written. Routines that have variable argument lists (such as **printf**) but do not use *stdarg* are inherently non-portable, as different machines use different argument-passing conventions.

va_list is a type defined for the variable used to traverse the list.

The **va_start()** macro is invoked before any access to the unnamed arguments and initializes **pvar** for subsequent use by **va_arg()** and **va_end()**. The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the **, ...**). If this parameter is declared with the **register** storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

The parameter *parmN* is required under strict ANSI C compilation. In other compilation modes, *parmN* need not be supplied and the second parameter to the **va_start()** macro can be left empty (for example, **va_start(pvar,);**). This allows for routines that contain no parameters before the **...** in the variable parameter list.

The **va_arg()** macro expands to an expression that has the type and value of the next argument in the call. The parameter **pvar** should have been previously initialized by **va_start()**. Each invocation of **va_arg()** modifies **pvar** so that the values of successive arguments are returned in turn. The parameter *type* is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The **va_end()** macro is used to clean up.

Multiple traversals, each bracketed by **va_start** and **va_end**, are possible.

EXAMPLE

This example gathers into an array a list of arguments that are pointers to strings (but not more than **MAXARGS** arguments) with function **f1**, then passes the array as a single argument to function **f2**. The number of pointers is specified by the first argument to **f1**.

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

void f1(int n_ptrs, ...)
{
        va_list ap;
        char *array[MAXARGS];
        int ptr_no = 0;

        if (n_ptrs > MAXARGS)
                n_ptrs = MAXARGS;
        va_start(ap, n_ptrs);
        while (ptr_no < n_ptrs)
                array[ptr_no++] = va_arg(ap, char*);
        va_end(ap);
        f2(n_ptrs, array);
}
```

Each call to **f1** shall have visible the definition of the function or a declaration such as

```
void f1(int, ...)
```

SEE ALSO

vprintf(3S)

NOTES

It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame. For example, **execl** is passed a zero pointer to signal the end of the list. **printf** can tell how many arguments there are by the format. It is non-portable to specify a second argument of **char**, **short**, or **float** to **va_arg**, because arguments seen by the called function are not **char**, **short**, or **float**. C converts **char** and **short** arguments to **int** and converts **float** arguments to **double** before passing them to a function.