V

va_arg() — Variable arguments (stdarg.h)

Return pointer to next argument in argument list
#include <stdarg.h>
typename ***va_arg(va_list** *listptr***,** *typename***);**

va_arg returns a pointer to the next argument in an argument list. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help write such functions portably. It is always used with **va_end** and **va_start** within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is an object defined in the header **stdarg.h**. It must first be initialized by the macro **va_start**.

typename is the name of the type for which **va_arg** is to return a pointer. For example, if you wish **va_arg** to return a pointer to an integer, *typename* should be of type **int**.

va_arg can only handle "standard" data types, i.e., those data types that can be transformed to pointers by appending an asterisk '*'.

Example

For an example of this macro, see the entry for **variable arguments**.

Cross-references

Standard, §4.8.1.2
The C Programming Language, ed. 2, p. 254

See Also

va_end, va_start, variable arguments

Notes

If there is no next argument for **va_arg** to handle, or if *typename* is incorrect, then the behavior of **va_arg** is undefined.

va_arg must be implemented only as a macro. If its macro definition is suppressed within a program, the behavior is undefined.

va_end() — Variable arguments (libc)

Tidy up after traversal of argument list
#include <stdarg.h>
void va_end(va_list *listptr***);**

va_end helps to tidy up a function after it has traversed the argument list for a function that takes a variable number of arguments. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help write such functions portably. It should be used with the routines **va_arg** and **va_start** from within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is declared in header **stdarg.h**. *listptr* must first have been initialized by macro **va_start**.

The manner of "tidying up" that **va_end** performs will vary from one computing environment to another. In many computing environments, **va_end** is not needed, and it may be implemented as an empty function.

Example

For an example of this function, see the entry for **variable arguments**.

Cross-references

Standard, §4.8.1.3
The C Programming Language, ed. 2, p. 254

See Also

va_arg, va_start, variable arguments

Notes

If **va_list** is not initialized by **va_start**, or if **va_end** is not called before a function with variable arguments exits, then behavior is undefined.

va_list — Type

Type used to handle argument lists of variable length

va_list is a **typedef** declared in the header **stdarg.h**.

va_list is used to help implement functions like **printf** and **scanf**, which can take an indeterminate number of arguments.

Example

For an example of this type, see the entry for **variable arguments**.

Cross-references

Standard, §4.8

See Also

va_arg, va_end, va_start, variable arguments

va_start() — Variable arguments (stdarg.h)

Point to beginning of argument list
#include <stdargs.h>
void va_start(va_list *listptr, type rightparm***);**

va_start is a macro that points to the beginning of a list of arguments. It can be used with functions that take a variable number of arguments, such as **printf** or **scanf**, to help implement them portably. It is always used with **va_arg** and **va_end** from within a function that takes a variable number of arguments.

listptr is of type **va_list**, which is a type defined in the header **stdarg.h**.

rightparm is the rightmost parameter defined in the function's parameter list — that is, the last parameter defined before the **...** punctuator. Its type is set by the function that is using **va_start**. Undefined behavior results if any of the following conditions apply to **rightparm**: if it has storage class **register**; if it has a function type or an array type; or if its type is not compatible with the type that results from the default argument promotions.

Example

For an example of this macro, see the entry for **variable arguments**.

Cross-references

Standard, §4.8.1.1
The C Programming Language, ed. 2, p. 254

See Also

va_arg, va_end, va_list, variable arguments

Notes

va_start must be implemented only as a macro. If the macro definition of **va_start** is suppressed within a program, the behavior is undefined.

value preserving — Definition

With respect to integral promotions, the Standard has adopted *value-preserving rules*. This may quietly break some existing code that depended on unsigned-preserving rules, as many UNIX implementations have done.

In most cases, there will be no difference in the results produced by unsigned-preserving rules and those produced by value-preserving rules. There are, however, several instances in which different results will be seen. For example:

```
long l;
unsigned int ui;
   . . .
l = ui + l;
```

In this operation, before the addition is performed, **ui** will first be promoted to type **long** if a **long** can hold the value contained in the **unsigned int**. The operation will then be performed as long addition, assigning the result to the variable **l**.

If a **long** is not large enough to represent the value contained in **ui**, which may occur under an implementation where **int**s and **long**s are the same size, then *both* **ui** and **l** are first converted to **unsigned long** before the addition is performed. Because conversion is needed to preserve the value (as opposed to the sign) of the operand as well as the result, the term “value preserving” is appropriate.

As usual, code may have to be generated to perform the conversion, and a high-quality implementation will usually issue a diagnostic message in such a case.

Cross-references

Standard, §3.2
The C Programming Language, ed. 2, pp

See Also

conversions, integral promotions

variable arguments — Overview

The Standard mandates the creation of a set of routines to help implement functions, such as **printf** and **scanf**, that take a variable number of arguments. These routines are called from within another function to help it handle its arguments. If the ellipsis punctuator ‘...’ appears at the end of the list of arguments in a function’s prototype, then that a function can take a variable number of arguments.

These routines are declared or defined in the header **stdarg.h**, and are as follows:

va_arg	Return pointer to next argument in argument list
va_end	Tidy up after an argument list has been traversed
va_start	Initialize object that holds function arguments

va_arg and **va_start** must be implemented as macros; **va_end** must be implemented as a library function. All three use the special type **va_list**, which is an object that holds the arguments to the function being implemented.

Example

The following example concatenates multiple strings into a common allocated string and returns the string's address.

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

char *
multcat(int numargs, ... )
{
      va_list argptr;
      char *result;
      int i, siz;

      /* get size required */
      va_start(argptr, numargs);
      for(siz = i = 0; i < numargs; i++)
            siz += strlen(va_arg(argptr, char *));

      if ((result = calloc(siz + 1, 1)) == NULL) {
            fprintf(stderr, "Out of space\n");
            exit(EXIT_FAILURE);
      }
      va_end(argptr);

      va_start(argptr, numargs);
      for(i = 0; i < numargs; i++)
            strcat(result, va_arg(argptr, char *));
      va_end(argptr);
      return(result);
}

int
main(void)
{
      printf(multcat(5, "One ", "two ", "three ",
            "testing", ".\n"));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.8
The C Programming Language, ed. 2, p. 254

See Also

Library, stdarg.h, va_list

vfprintf() — STDIO (libc)

Print formatted text into stream
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE**fp***, const char ****format***, va_list** *arguments***);**

vfprintf constructs a formatted string and writes it into the stream pointed to by *fp*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vfprintf** can handle a variable list of arguments of various types. It is roughly equivalent to the 'r' conversion specifier to **fprintf**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how particular data type is converted into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification, and for a table of the type specifiers that can be used with **vfprintf**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start** and points to the base of the list of arguments used by **vfprintf**. For more information, see **variable arguments**. *arguments* should access one argument for each conversion specification in *format*, of the type appropriate to its conversion specification.

For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char** *. *arguments* can take only the data types acceptable to the macro **va_arg**; namely, basic types that can be converted to pointers simply by adding a '*' after the type name. See **va_arg** for more information on this point.

If there are fewer arguments than conversion specifications, then **vfprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding conversion specifier, then the behavior of **vfprintf** is undefined. Thus, presenting an **int** where **vfprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vfprintf** returns the number of characters written. Otherwise, it returns a negative number.

Example

This example sets up a standard multiargument error message. It is the source of the function **fatal**, which is used throughout this manual.

```
#include <math.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *format, ...)
{
      va_list argptr;

      /* if there is a system message, display it */
      if(errno)
            perror(NULL);

      /* if there is a user message, use it */
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}
```

```
main(void)
{
      /*
       * This is guaranteed to be wrong.  It should push
       * an error code into errno.
       */
      sqrt(-1.0);

      /* Now, show the messages */
      fatal("A %s error message%c", "complex", '\n');

      /* If we get this far, something is wrong */
      return(EXIT_FAILURE);
}
```

Cross-references

Standard, §4.9.6.7
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, sprintf, STDIO, vprintf, vsprintf

Notes

vfprintf can construct a string up to at least 509 characters long.

The character that **vfprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

void — C keyword

Empty type

The term **void** indicates the empty type. The following sections describe the ways it is used.

Function Type

void can be used in a function prototype or definition to indicate that a function returns no value. For example, the declaration

```
      void example();
```

indicates that the function **example** returns nothing. It would be an error for **example** to attempt to return a value to a function that calls it, or for the calling function to use its value in an expression.

Function Arguments

void can also be used in a function prototype or function declaration to indicate that a function has no arguments. For example, the declaration

```
      void example(void);
```

indicates that the function **example** not only returns nothing, but it takes no arguments as well. The older practice of writing **example()** remains legal. But as before, it indicates merely that nothing is said about arguments.

Void Expression

void can be used to indicate that the value of an expression is to be ignored. This is done by casting the expression to type **void**. Prefacing an expression with the cast **(void)** throws away its value (i.e., casts it into the void), although the expression is evaluated for possible side-effects.

void *

A **void** * ("pointer to void") is a generic pointer. It is used in much the same way that **char** * ("pointer to **char**") was used in earlier descriptions of C. The new generic pointer type eliminates the earlier confusion between a pointer to **char** (e.g., a string pointer) and a generic pointer.

Because by definition the **void** type includes no objects, a pointer to **void** may not be dereferenced. That is, you should not directly access the object to which it points by using the indirection operator '*'. In the code

```
void *voidp;
    . . .
if (*voidp > 0)
    . . .
```

the behavior of dereferencing the pointer to **void** is undefined. It may or may not generate an error; if it does not, the results may be unpredictable.

It is correct, however, to cast a pointer to **void** to a standard object pointer type and then dereference it. For example, the code

```
void *voidp;
    . . .
if (*(char *)voidp > 0)
    . . .
```

is permitted.

The Standard guarantees that a pointer to **void** may be converted to a pointer to any incomplete type or object type. It also guarantees that a pointer to any incomplete type or object type may be converted into a pointer to **void**. Moreover, converting the result back to the original type results in a pointer equal to the original pointer. That is, conversion of any object pointer type to **void** * and back again does not change the representation of the pointer. However, if an object pointer is converted to **void** * and then converted to a pointer to a type whose alignment is stricter than that of the original type, behavior is undefined.

The Standard also guarantees that the pointer types **char** * and **void** * have the same representation. This prevents the Standard from breaking existing code for functions with generic-pointer arguments (previously defined using type **char** * but now defined with type **void** *).

The introduction of the generic pointer **void** * by the Standard serves several purposes in addition to those noted above. The Standard no longer allows comparison between pointers of different types, except that any object pointer may be compared to a **void** *. Casting object pointers with the expression

```
(void *)
```

allows comparisons that would otherwise be illegal. Library functions that have commonly been written with pointers of various types as arguments (such as **fread**) can be defined with a prototype **void** * argument, which allows the arguments to be quietly converted to the correct type.

The generic pointer **void** * is also used as the type of the value returned by some functions (e.g., **malloc**), to indicate that the returned value is a pointer to something of indeterminate type.

Cross-references

Standard, §3.1.2.5, §3.2.2.2-3, §3.3.4, §3.5.2, §3.5.3.1, §3.5.4.3
The C Programming Language, ed. 2, pp. 199, 218

See Also

NULL, pointer, precedence, types

void expression — Definition

A *void expression* is any expression that has type **void**. By definition, it has no value; therefore, its value cannot be assigned to any other expression. Normally, a void expression is used for its side-effects.

If an expression of any other type is used in a situation that requires a void expression, the value of that expression is discarded.

Cross-reference

Standard, §3.2.2.2

See Also

conversions

volatile — C keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to re-fetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

Cross-references

Standard, §3.5.3
The C Programming Language, ed. 2, p. 211

See Also

const, type qualifier

Notes

volatile was created by the Committee for systems' programs that deal with memory-mapped I/O or ports where the program is not the only task that may modify the given port in memory. **volatile** tells the translator that it does not know everything that is happening to the object.

Another use for **volatile** is for translators that perform optimizations, such as deferring storage of registers or peephole optimization. **volatile** requires that the object be read and stored at exactly those points where the program has specified these actions.

vprintf() — STDIO (libc)

Print formatted text into standard output stream
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char **format*, **va_list** *arguments*)**;**

vprintf constructs a formatted string and writes it into the standard output stream. It translates integers, floating-point numbers, and strings into a variety of text formats. **vprintf** can handle a variable list of arguments of various types. It is roughly equivalent to the 'r' conversion specifier to **printf**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification defines how a particular data type is converted into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vprintf**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start** and points to the base of the list of arguments used by **vprintf**. For more information, see **variable arguments**. *arguments* should access one argument for each conversion specification in *format* of the type appropriate to conversion specification.

For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char ***.

If there are fewer arguments than conversion specifications, then **vprintf**'s behavior is undefined. If there are more, every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then the behavior of **vprintf** is undefined; thus, accessing an **int** where **vprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vprintf** returns the number of characters written; otherwise, it returns a negative number.

Cross-references

Standard, §4.9.6.8
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, sprintf, STDIO, vfprintf, vsprintf

Notes

vprintf can construct a string up to at least 509 characters long.

The character that **vprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

Each *argument* must have basic type, which can be converted to a pointer simply by adding an '*' after the type name. This is the same restriction that applies to the arguments to the macro **va_arg**.

vsprintf() — STDIO (libc)

Print formatted text into string
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char **string*, **const char** **format*, **va_list** *arguments***);**

vsprintf constructs a formatted string in the area pointed to by *string*. It translates integers, floating-point numbers, and strings into a variety of text formats. **vsprintf** can handle a variable list of arguments of various types. It is roughly equivalent to the 'r' conversion specifier to **sprintf**.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how to convert a particular data type into a particular text format. Each conversion specification is introduced with the percent sign '%'. (To print a literal percent sign, use the escape sequence "%%".) See **printf** for further discussion of the conversion specification and for a table of the type specifiers that can be used with **vsprintf**.

After *format* comes *arguments*. This is of type **va_list**, which is defined in the header **stdarg.h**. It has been initialized by the macro **va_start** and points to the base of the list of arguments used by **vsprintf**. For more information, see **variable arguments**. *arguments* should access one argument for each conversion specification in *format* of the type appropriate to the conversion specification.

For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *arguments* access three arguments, being, respectively, an **int**, a **long**, and a **char ***.

If there are fewer arguments than conversion specifications, then **vsprintf**'s behavior is undefined. If there are more, then every argument without a corresponding conversion specification is evaluated and then ignored. If an argument is not of the same type as its corresponding type specification, then the behavior of **vsprintf** is undefined; thus, accessing an **int** where **vsprintf** expects a **char** * may generate unwelcome results.

If it writes the formatted string correctly, **vsprintf** returns the number of characters written; otherwise, it returns a negative number.

Cross-references

Standard, §4.9.6.7
The C Programming Language, ed. 2, p. 245

See Also

fprintf, printf, sprintf, STDIO, vprintf, vsprintf

Notes

vsprintf can construct a string up to at least 509 characters long.

The character that **vsprintf** uses to represent the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.