

P

parameter *See* argument, formal.

parameter, ellipsis^{C89} *See* ellipses.

parameter type list A comma-separated list of types (optionally containing identifiers) as used in a function prototype.

parentheses punctuator The tokens `()` that can be used as a grouping punctuator. Parentheses may be used in expressions to override operator precedence or to document the default precedence. For example, in `a + (b * c)` the grouping parentheses are redundant, whereas in `(a + b) * c` they are not. A parenthesized expression has the same type and value as the expression without the parentheses. For example, `(i) = ((6))` is equivalent to `i = 6`. Note that grouping is not related to, and cannot be used to control, the order of evaluation of the individual terms.

parenthesized expression An expression surrounded by parentheses. *See also* expression, parenthesized; expression, primary.

p_cs_precedes^{C89} An `lconv` structure member that is a nonnegative number set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

perror A function that writes a message to `stderr` corresponding to the current value of `errno`. The message includes the user-supplied string pointed to by `s`.

```
#include <stdio.h>
void perror(const char *s);
```

The message output is prefixed with the string pointed to by `s` followed by a colon and a space, provided `s` is not `NULL` and doesn't point to an empty string. The contents and format of the message are implementation defined and are the same as those returned by the `strerror` function with argument `errno`.

phases of translation^{C89} These are described by Standard C as follows:

“The precedence among the syntax rules of translation is specified by the following phases:¹

¹Implementations must behave as if these separate phases occur, even though many are typically folded together in practice.

1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.
2. Each instance of a backslash character (`\`) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.
3. The source file is decomposed into preprocessing tokens² and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation defined.
4. Preprocessing directives are executed, macro invocations are expanded, and, in C99, `_Pragma` unary operator expressions are executed. Also in C99, if a character sequence that matches the syntax of a universal character name is produced by token concatenation, the behavior is undefined. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.
5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.³
6. Adjacent character string literal tokens are concatenated and, in C99, adjacent wide string literal tokens are concatenated.
7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated.
8. All external object and function references are resolved. Library components are linked to satisfy external references to functions

²The process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of `<` within a `#include` preprocessing directive.

³An implementation need not convert all noncorresponding source characters to the same execution character.

and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.”

plain char *See* char, plain.

plain int bit-field *See* bit-field, plain int.

plus operator, unary^{C89} *See* unary plus operator.

pointer An object that contains the address of an object or function, or an expression that designates the address of an object or function. For example, given the following:

```
int i = 10;
int *pi = &i;
```

`pi` is a pointer object and `&i` is a pointer expression that does not involve a pointer object.

A pointer expression is dereferenced using the unary `*` operator or the subscript operator `[]`. Pointers to structures or unions are dereferenced using the arrow operator `->`. A pointer can be initialized by using the unary `&` operator, although this operator is redundant if the initializer is of some function type.

A function can be called by dereferencing a pointer to it.

pointer declarator A declarator involving the derived type “pointer to some type.” *See also* declarator, punctuation in.

pointer, null *See* constant, null pointer.

pointer subtraction Two pointers can be subtracted, one from the other, provided both point to elements of the same array or one points to an element of an array and the other points to the (nonexistent) element immediately following the last element of that same array. The type of the result is `ptrdiff_t`, an integer type defined in `stddef.h`. Note that the difference between two such pointers is neither inclusive nor exclusive of the objects to which they point. For example, given the following:

```
int i[10];
int *p1 = &i[3];
int *p2 = &i[5];
```

the expression `p2 - p1` results in 2 while the expression `p1 - p2` results in `-2`. In both cases, the magnitude of the difference is 2, the number of elements between the start of `i[3]` and the start of `i[5]`. The sign indicates direction.

pointer to function An expression or variable used to represent the address of a function. Conceptually, a function pointer contains the address of the first byte or word of the executable code for that function. However, in some systems it contains the address of an object, which in turn points to the function. You cannot perform arithmetic operations on function pointers. *See also* jump table.

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

pointer type conversion *See* conversion, pointer.

portability The degree of ease with which a program can be moved from one computer system to another (somehow) dissimilar system. A primary obstacle in porting C code is that many aspects of C are implementation defined. *See also* program, strictly conforming.

positive_sign^{C89} An `lconv` structure member that is a pointer to a string used to indicate a nonnegative-valued formatted monetary quantity. If the string consists of "", the value is not available in the current locale or is of zero length. In the "C" locale this member must have the value "".

POSIX A portable operating system definition based on UNIX, produced by IEEE committee P1003. POSIX defines many library functions over and above those defined by Standard C. The international counterpart to P1003 is WG15.

pow[f|l] A function that computes the power function x^y .

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

If `x` is negative and `y` is not a whole number, a domain error occurs. If `x` is 0 and `y` is less than or equal to 0, and the result cannot be represented, a domain error occurs. A range error may occur.

The `float` and `long double` versions were an invention of C89, where they were optional; however, in C99, they are required.

power functions The `math.h` functions `pow` and `sqrt`, and their `float` and `long double` counterparts, both floating and complex.

_Pragma^{C99} A unary preprocessor operator that is used in expressions as follows:

```
_Pragma( string-literal )
```

Such an expression is processed as follows:

_Pragma

1. The string literal is destringized by deleting the L prefix, if present, deleting the leading and trailing double quotes, replacing each escape sequence \" by a double quote, and replacing each escape sequence \\ by a single backslash.
2. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the tokens in a pragma directive.
3. The original four preprocessing tokens in the unary operator expression are removed.

Unlike #pragma, _Pragma can be used in a macro definition.

#pragma^{C89} An implementation-defined preprocessor directive. If an implementation comes across a pragma it does not recognize, it ignores it. A pragma has the following general form:

```
#pragma [ preprocessor-tokens ]
```

Some compilers recognize a pragma that specifies how tightly adjacent members in a structure are packed; for example,

```
#pragma pack( n )
```

where *n* can be 1, 2, or 4 indicating byte, word, or double-word alignment, respectively. Other compilers define pragmas to change the way in which arguments are passed to functions and to help the compiler break the code into threads for a parallel processing environment.

Although an implementation is permitted to perform macro replacement in nonstandard pragmas, it not required to do so.

For information on standard pragmas, see #pragma STDC.

#pragma STDC^{C99} A standard pragma, which must have one of the following forms:

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC FP_CONTRACT on-off-switch
```

No macro replacement is performed on standard pragma directives. It is implementation defined whether macro replacement is performed on nonstandard pragma directives.

#pragma STDC CX_LIMITED_RANGE^{C99} A standard pragma that can be used to indicate whether or not the usual mathematical formulas for complex

multiply, divide, and absolute value are acceptable, based on their treatment of infinities and because of undue overflow and underflow. There are limits on where this pragma should be placed in a source file. The three forms of this pragma are:

```
#pragma STDC CX_LIMITED_RANGE ON
#pragma STDC CX_LIMITED_RANGE OFF
#pragma STDC CX_LIMITED_RANGE DEFAULT
```

where the default state is **OFF**.

#pragma STDC FENV_ACCESS^{C99} A standard pragma that provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under nondefault floating-point control modes. There are limits on where in a source file this pragma should be placed. The three forms of this pragma are

```
#pragma STDC FENV_ACCESS ON
#pragma STDC FENV_ACCESS OFF
#pragma STDC FENV_ACCESS DEFAULT
```

where the default state is implementation defined.

#pragma STDC FP_CONTRACT^{C99} A standard pragma that provides a way to allow or disallow the implementation to contract expressions. The three forms of this pragma are

```
#pragma STDC FP_CONTRACT ON
#pragma STDC FP_CONTRACT OFF
#pragma STDC FP_CONTRACT DEFAULT
```

where the default state is implementation defined.

precedence The hierarchy and associativity of operators. *See also* operator precedence.

predefined macro *See* macro, predefined.

preprocessing directives Source lines that have the following general format:

```
# [ directive-name ] [ preprocessing-tokens ]
```

Horizontal white space may exist before the #, between the # and *directive-name*, and before, after, or in between the preprocessing tokens. Unless

continued by a backslash/new-line pair, a preprocessing source line terminates at the end of the current physical source line. The complete set of such directives is as follows:

<i>Name</i>	<i>Purpose</i>
#define	Define a macro
#undef	Remove a macro definition
#include	Include a header
#if	Compile based on truth of expression
#ifdef	Compile based on macro being defined
#ifndef	Compile based on macro not being defined
#else	Conditional compilation false path
#elif	Compound else/if
#endif	End conditional compilation group
#line	Override line number/source file name
#error	Generate a translation error
#pragma	Implementation-defined action
#	Null directive

preprocessing operator A number of operators are available only to the preprocessor. They are as follows:

<i>Name</i>	<i>Purpose</i>
#	Stringize
##	Token pasting
defined	Expression form of #ifdef
_Pragma ^{C99}	Allows a pragma in a macro definition

preprocessing token A token in translation phases 3 through 6. The complete set of preprocessing tokens defined by C89 is character-constant, header-name, identifier, operator, preprocessing number, punctuator, string-literal, and each non-white-space character that cannot be one of the above. C99 dropped operator, moving such tokens into the punctuator category instead.

preprocessor A program that scans a C source file for lines beginning with a **#**, which it assumes to be directives that indicate some action to be taken before subsequent source code lines are handed off to the compiler. Typically, the preprocessor is actually integrated with the compiler proper.

PRIdN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `intN_t`. For example,

```
int32_t x = 10;
printf("x = %2" PRId32 "\n", x);
```

PRIdLEASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `int_leastN_t`.

PRIdFASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `int_fastN_t`.

PRIdMAX^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `intmax_t`.

PRIdPTR^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `intptr_t`.

printf A function that writes formatted output to `stdout` as specified by `format`.

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

Characters in `format` other than `%` and those specifying a particular conversion specifier are output verbatim. To output a `%` character, use `%%`.

`printf` returns the number of characters it transmitted. If an output error occurred, a negative value is returned.

A call to `printf` is equivalent to a call to `fprintf` using a stream of `stdout`.

The general format of a `printf` conversion specifier is as follows:

`%[flags][width][.precision][modifier]specifier`

where the values for `flags`, `modifier`, and `specifier` are as follows:

printf Flags	
<i>Symbol</i>	<i>Meaning</i>
-	Left-justify
+	Leading sign
<i>space</i>	Leading space
#	Alternate output form
0	Leading zeros

printf Modifiers

<i>Symbol</i>	<i>Meaning</i>
h	short int
hh ^{C99}	char
j ^{C99}	[u]intmax_t
l	long int
L	long double
t ^{C99}	ptrdiff_t
z ^{C99}	size_t

printf Specifiers

<i>Symbol</i>	<i>Meaning</i>
a ^{C99}	Hex floating
A ^{C99}	Uppercase hex floating
c	Character
d	Signed decimal
e	Lowercase exponent
E	Uppercase exponent
f	Fractional (6 decimal places)
F ^{C99}	Uppercase NaN/Infinity
g	Shorter of e or f
G	Shorter of E or f
i	Signed decimal
n	Stores char count in <code>int</code>
o	Unsigned octal
p	Pointer to void
s	String
u	Unsigned decimal
x	Unsigned lowercase hex
X	Unsigned uppercase hex
%	Print % character

See also future library directions.

PRIoN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintN_t`. For example,

```
uint32_t x = 10;
printf("x = %2" PRIo32 "\n", x);
```

PRIoLEASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with

members of the `printf` and `wprintf` family when converting the type `uint_leastN_t`.

`PRIOFASTNC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_fastN_t`.

`PRIOMAXC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintmax_t`.

`PRIOPTRC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintptr_t`.

`PRIUC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintN_t`. For example,

```
uint32_t x = 10;
printf("x = %2" PRIu32 "\n", x);
```

`PRIULEASTNC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_leastN_t`.

`PRIUMAXC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintmax_t`.

`PRIUFASTNC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_fastN_t`.

`PRIUPTRC99` A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintptr_t`.

`private` 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 `private` as an identifier in new C code you write.

PRIXN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintN_t`. For example,

```
uint32_t x = 10;
printf("x = %2" PRIX32 "\n", x);
```

PRIXN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintN_t`. For example,

```
uint32_t x = 10;
printf("x = %2" PRIx32 "\n", x);
```

PRIXLEASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_leastN_t`.

PRIXLEASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_leastN_t`.

PRIXFASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_fastN_t`.

PRIXFASTN^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uint_fastN_t`.

PRIXMAX^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintmax_t`.

PRIXMAX^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintmax_t`.

PRIXPTR^{C99} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintptr_t`.

PRIxPTR^{C89} A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `printf` and `wprintf` family when converting the type `uintptr_t`.

program One or more (possibly separately compiled) functions of which one must be called `main`. These functions, along with any external data, make up an execution unit. A program may include functions from an external library.

program, conforming^{C89} A program that is acceptable to a conforming implementation. Note that a conforming program is not necessarily maximally portable, since it might use extensions particular to the conforming implementation that accepts it.

program name *See* `argv`.

program parameters *See* `argc`; `argv`; `envp`.

program startup That vendor-supplied code executed before control is passed to `main` in a host environment, or some implementation-defined function in a free-standing environment.

program, strictly conforming^{C89} A program that uses only those features of the language and library defined by Standard C. Its output must not depend on unspecified, undefined, or implementation-defined behavior. *See also* conforming implementation; program, conforming.

program termination The stopping of the execution of a program. A program may be terminated a number of ways: by dropping through the closing brace of `main`, by executing a `return` statement from `main`, by calling `abort` or `exit`, or by an interrupt occurring that caused program termination. *See also* `exit` code.

program termination, abnormal *See* `abort`.

program termination, normal *See* `exit`.

promotions, default argument *See* conversion, function arguments.

promotions, integer *See* conversion, integer type.

protected 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 `protected` as an identifier in new C code you write.

prototype^{C89} *See* function prototype.

p_sep_by_space^{C89} An `lconv` structure member that is a nonnegative number set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity. A value of `CHAR_MAX` indicates that the value is not available in the current locale. In the "C" locale this member must have the value `CHAR_MAX`.

pseudo-random sequence functions The `stdlib.h` functions `rand` and `srand`.

p_sign_posn^{C89} An `lconv` structure member that is a nonnegative number set to a value indicating the positioning of the `positive_sign` for a nonnegative formatted monetary quantity. The value is interpreted according to the following:

0 Parentheses surround the quantity and `currency_symbol`.

1 The sign string precedes the quantity and `currency_symbol`.

2 The sign string succeeds the quantity and `currency_symbol`.

3 The sign string immediately precedes the `currency_symbol`.

4 The sign string immediately succeeds the `currency_symbol`.

`CHAR_MAX` indicates that the value is not available in the current locale.

In the "C" locale this member must have the value `CHAR_MAX`.

PTRDIFF_MAX^{C99} A macro, defined in `stdint.h`, that indicates the maximum value of the type `ptrdiff_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

PTRDIFF_MIN^{C99} A macro, defined in `stdint.h`, that indicates the minimum value of the type `ptrdiff_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

ptrdiff_t^{C89} The type of the difference between two pointers of the same type. This type is signed and integer. (In practice, both pointers must point to elements within the same array for the subtraction to be meaningful or, in fact, reliable.) `ptrdiff_t` is defined in `stddef.h`. See also pointer subtraction; `printf` conversion specifier `%t`; `PTRDIFF_MAX`; `PTRDIFF_MIN`.

public 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 `public` as an identifier in new C code you write.

punctuator One of the following tokens: `[] () { } . , -> ++ -- & * + - ~ ! / % << >> < > <= >= == != ^ | && || ? , : = ; ; . . . # ## = *= /= %= += -= <<= >>= &= ^= |= <: :> <% %> %: %:~`. Many punctuators also serve as operators.

putc A function that writes the character specified by *c* (converted to **unsigned char**) to the file pointed to by *stream*.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

putc is equivalent to **fputc** except that **putc** is permitted to be an unsafe macro. *See also* **putwc**.

putchar A function that writes the character specified by *c* (converted to **unsigned char**) to **stdout**.

```
#include <stdio.h>
int putchar(int c);
```

putchar is equivalent to **putc** to **stdout**. Therefore, **putchar** may be implemented as an unsafe macro. *See also* **putwchar**.

puts A function that writes the string pointed to by *s*, followed by a new-line, to **stdout**.

```
#include <stdio.h>
int puts(const char *s);
```

The '\0' terminating the string is not written. Unlike **fputs**, **puts** appends a new-line to the output. **puts** returns EOF if an error occurs; otherwise, it returns a nonnegative value.

putwc^{C95} A function that is equivalent to **fputwc**, except that if it is implemented as a macro, it may evaluate *stream* more than once.

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *stream);
```

The value returned is either the wide character written or WEOF.

putwchar^{C95} A function that writes the given wide character to standard output.

```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

The value returned is either the wide character written or WEOF.

