P

parameter — Definition

The term *parameter* refers to an object that is declared with a function or a function-like macro.

With a function, a parameter is declared within a function declaration or definition. It acquires a value when the function is entered. For example, in the following declaration

```
FILE *fopen (const char *file, const char *mode);
```

file and **mode** are both objects that are declared within the function declaration. Both parameters will acquire their values when **fopen** is called.

With a function-like macro, a parameter is one of the identifiers that is bracketed by parantheses and separated by commas. For example, in the following example:

```
#define getchar(parameter) getc(stdin, parameter)
```

parameter is the identifier used with the macro **getchar**.

The scope of a function parameter is the block within which it is enclosed. The scope of a parameter to a function-like macro is the logical source line of the macro's definition.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 202

See Also

argument, Definitions, function definition, scope

Notes

The Standard uses the term "argument" when it refers to the actual arguments of a function call or macro invocation. It uses the term "parameter" to refer to the formal parameters given in the definition of the function or macro.

PATH — Environmental variable

Directories that hold executable files

PATH names a default set of directories that are searched by MS-DOS when it seeks an executable file. You can set **PATH** with the MS-DOS command **path**. For example, typing

```
path c:\bin;a:\bin
```

tells MS-DOS to search for executable files first in **c:\bin**, and then in **a:\bin**.

For more information on the **path** command, see your MS-DOS user's manual.

See Also

environmental variable, path.h

path() — Access checking (libc)

Build a path name for a file
#include <path.h>
#include <stdio.h>
char *__path__**(char** ******path***, char** ******filename***, int** *mode***);**

The function **path** builds a path name for a file.

path points to the list of directories to be searched for the file. You can use the function **getenv** to obtain the current definition of the environmental variable **PATH**, or use the default setting of **PATH** found in the header file **path.h**, or, you can define *path* by hand.

filename is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

1 Execute the file
2 Write to the file
4 Read the file

path uses the function **access** to check the access status of *filename*. If **path** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <path.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *message)
{
      fprintf(stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      char *env, *pathname;
      int mode;

      if (argc != 3)
            fatal("Usage: findpath filename mode");

      if(((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
            fatal("modes: 1=execute, 2=write, 3=read");

      env = getenv("PATH");
      if ((pathname = path(env, argv[1], mode)) != NULL) {
            printf("PATH = %s\n", env);
            printf("pathname = %s\n", pathname);
            return(EXIT_SUCCESS);
      } else
            fatal("search failed");
}
```

See Also

access, access checking, access.h, PATH, path.h

path.h — Header

Declare path()
#include <path.h>

path.h is a header that declares the function **path**. It also contains a number of default definitions for variables, including **PATH** and **LIBPATH**.

LEXICON

See Also

access checking, header, LIBPATH, path, PATH

pattern — Definition

A **pattern** is any combination of ASCII characters and wildcard characters that can be interpreted by a command.

The function **pnmatch** compares two patterns and signals if they match.

See Also

Definitions, egrep, pnmatch, wildcard

peek() — Extended function (libc)

Extract a word from memory
unsigned peek(unsigned *offs***, unsigned** *seg***);**

peek examines an arbitrary location in memory. It reads a word (two bytes) located at the offset *offs* and segment *seg*.

If your program is compiled into SMALL model, you can supply the offset/segment pair by using the macro **PTR**.

The header file **bios.h** declares a structure that defines the entire MS-DOS BIOS data area. You can use it to access an area within the BIOS data area for **peek**ing or **poke**ing.

Example

This example reads the address where the IBM PC stores the current memory size.

```
#include <stdio.h>
#include <stdlib.h>
#define MEMSIZELOC 0x13

main(void)
{
      extern unsigned peek();
      int size;

      size = (int)peek(MEMSIZELOC, 0x0);
      printf("Memory size = %d Kbytes\n",size);
      return EXIT_SUCCESS;
}
```

See Also

BIOS data area, bios.h, extended miscellaneous, peekb, poke, pokeb

peekb() — Extended function (libc)

Extract a byte from memory
unsigned peekb(unsigned *offs***, unsigned** *seg***);**

peekb examines an arbitrary location in memory. It reads a byte located at the offset *offs* and segment *seg*.

Note that if your program is compiled into SMALL model, you can supply the offset/segment pair by using the macro **PTR**.

Example

This example reads the MS-DOS location that holds the amount of memory on your machine.

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

main(void)
{
      extern unsigned peekb();
      unsigned hbyte, lbyte, word;

      hbyte = peekb(0x14,0x0);
      lbyte = peekb(0x13,0x0);
      word = ((hbyte << 8) | lbyte);
      printf("Memory size = %d Kbytes\n",(int)word);
      return EXIT_SUCCESS;
}
```

See Also

_copy, csreg, extended miscellaneous, peek, poke, pokeb, PTR, _zero

perror() — STDIO (libc)

Write error message into standard error stream
#include <stdio.h>
void perror(const char ******string***);**

perror checks the integer expression **errno**, then writes the message associated with the value of **errno** into the standard error stream.

string points to a string that will prefix the error message, followed by a colon. For example, the call

```
perror("example");
```

ensures that the string

```
example:
```

will appear before any message that **perror** writes. If *string* is set to NULL, then the message will have no prefix.

Example

For an example of this function, see **feof**.

Cross-references

Standard, §4.9.10.4
The C Programming Language, ed. 2, p. 248

See Also

clearerr, errno, error codes, feof, ferror, STDIO, strerror

Notes

perror differs from the related function **strerror** in that it writes the error message directly into the standard error stream, instead of returning a pointer to the message.

The text of the message returned by **strerror** and the error-specific part of the message produced by **perror** should be the same for any given error number.

The external array **sys_errlist** gives the list of messages used by **perror**. The external variable **sys_nerr** gives the number of messages in the list.

LEXICON

picture() — Example

Format numbers under mask
double picture(double *number*, **const char** **mask, char* **output*);

picture uses a mask to format a double-precision number. It is designed to be used with programs that require precise formatting of printed numbers.

picture formats a given number by using a mask string. It writes its output into the area pointed to by *output*.

mask may contain any characters; however, only a few have special significance. Non-special characters in *mask* are printed if, during execution, they are preceded by one or more numerals. Trailing non-special characters print if the displayed number is negative.

picture returns all overflow as a **double**. For example, attempting to print **-1234** with mask **(ZZZ)** gives **(234)** and returns **-1**.

The following lists the special characters that control formatting within a mask:

9 Provides a slot for a number. For example, **5** with mask **999 CR** gives **005<sp><sp><sp>**, whereas printing **-5** with mask **999 CR** gives **005 CR**. 'C' and 'R' are not special characters, but are taken literally.

Z Provide a slot for a number but supress leading zeroes. For example, printing **1034** with mask **ZZZ,ZZZ** gives **<sp><sp>1,034**. The comma is not a special character, but is printed literally.

J Provide a slot for a number but shrink out leading zeroes. For example, printing **1034** with mask **JJJ,JJJ** gives **1,034**.

K Provide a slot for a number but shrink out all zeroes. For example, printing **070884** with mask **K9/K9/K9** gives **7/8/84**.

$ Print a dollar sign to the front of the displayed number. For example, printing **105** with mask **$Z,ZZZ** gives **<sp><sp>$105**.

. Separate the number between decimal and integer portions. For example, printing **105.67** with mask **ZZZ.999** gives **105.670**.

T Provide a slot for a number, but supress trailing zeroes. For example, printing **105.670** with mask **ZZ9.9TT** gives **105.67<sp>**.

S Provide a slot for a number, but shrink out trailing zeroes. For example, printing **105.600** with mask **ZZ9.9SS** gives **105.6**.

- If you place a hyphen to the left of the mask, it is printed at the beginning of the number, but only if it is negative. For example, printing **105** with mask **-Z,ZZZ** yields **<sp><sp>105**, whereas printing **-105** yields **<sp><sp>-105**.

(This character acts like the minus sign '-', but prints a '('. For example, printing **105** with mask **(ZZZ)** gives **<sp>105<sp>**, whereas printing **-5** gives **<sp><sp>(5)**.

+ If placed to the left of the mask, this character floats to the front like the minus sign '-', but is replaced by a '-' if the number is minus. For example, printing **5** with mask **+ZZZ** gives **<sp><sp>+5**, whereas printing **-5** gives **<sp><sp>-5**. Placed behind the mask, it is printed if the number is positive, but is replaced by a minus sign '-' if the number is negative. For example, printing **5** with mask **ZZZ+** gives **<sp><sp>5+**, whereas printing **-5** gives **<sp><sp>5-**.

* When placed to the left of the mask, this character fills all leading spaces to its right. For example, printing **104.10** with mask ***ZZZ,ZZZ.99** gives *****104.10**, and printing **104.10** with mask ***$ZZ,ZZZ.99** gives ****$104.10**.

Example

For an example of **picture**, compile the source program **picture.c** with the option **-DTEST**.

See Also

example

Notes

For the source code of **picture**, see the file **picture.c**, which is included with **Let's C**. **picture** is not included in a library.

pnmatch() — Extended function (libc)

Match string pattern
short pnmatch(char **string***, char** **pattern***, short** *flag***);**

pnmatch matches *string* with *pattern*, which is a regular expression.

pnmatch returns a positive number if *pattern* matches *string*, and zero if it does not.

Each character in *pattern* must exactly match a character in *string*. However, the wildcards '*', '?', '[', and ']' can be used in *pattern* to expand the range of matching. See **wildcards** for more information on what these symbols mean.

The *flag* argument must be either zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '^' and '$' can also be used in *pattern*.

Example

This example looks for the pattern given by **argv[1]** in standard input or in file **argv[2]**.

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

#define MAXLINE 128
char buf[MAXLINE];

void
fatal(char *message)
{
      fprintf(stderr, "pnmatch: %s\n", message);
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      /* Check that number of arguments is OK */
      if (argc != 2 && argc != 3)
            fatal("Usage: pnmatch pattern [ file ]");
      if (argc==3 && freopen(argv[2], "r", stdin)==NULL)
            fatal("cannot open input file");
```

```
      /* Get string, check with pattern */
      while (fgets(buf, MAXLINE, stdin) != NULL)
      {
            if (pnmatch(buf, argv[1], 1))
                  printf("%s", buf);
      }
      if (!feof(stdin))
            fatal("read error");
      exit(EXIT_SUCCESS);
}
```

See Also

extended miscellaneous, strcmp, strncmp, strstr, wildcards

Notes

flag must be zero or one for **pnmatch** to yield predictable results.

pnmatch is not described in the ANSI standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or other environments.

pointer — Definition

A *pointer* is an object whose value is the address of another object. The name "pointer" derives from the fact that its contents "point to" another object. A pointer may point to any type, complete or incomplete, including another pointer. It may also point to a function, or to nowhere.

The term *pointer type* refers to the object of a pointer. The object to which a pointer points is called the *referenced type*. For example, an **int** * ("pointer to **int**") is a pointer type; the referenced type is **int**. Constructing a pointer type from a referenced type is called *pointer type derivation*.

The Null Pointer

A pointer that points to nowhere is a *null pointer*. The macro **NULL**, which is defined in the header **stddef.h**, defines the null pointer for a given implementation. The null pointer is an integer constant with the value zero, or such a constant cast to the type **void** *. It compares unequal to a pointer to any object or function.

Declaring a Pointer

To declare a pointer, use the indirection operator '*'. For example, the declaration

```
int *pointer;
```

declares that the variable **pointer** holds the address of an **int**-length object.

Likewise, the declaration

```
int **pointer;
```

declares that **pointer** holds the address of a pointer whose contents, in turn, point to an **int**-length object. See **declarations** for more information.

Wild Pointers

Pointers are omnipresent in C. C also allows you to use a pointer to read or write the object to which the pointer points; this is called *pointer dereferencing*. Because a pointer can point to any place within memory, it is possible to write C code that generates unpredictable results, corrupts itself, or even obliterates the operating system if running in unprotected mode. A pointer that aims where it ought not is called a *wild pointer*.

When a program declares a pointer, space is set aside in memory for it. However, this space has not yet been filled with the address of an object. To fill a pointer with the address of the object you wish

to access is called *initializing* it. A wild pointer, as often as not, is one that is not properly initialized.

Normally, to initialize a pointer means to fill it with a meaningful address. For example, the following initializes a pointer:

```
int number;
int *pointer;
   . . .
pointer = &number;
```

The address operator '&' specifies that you want the address of an object rather than its contents. Thus, **pointer** is filled with the address of **number**, and it can now be used to access the contents of **number**.

The initialization of a string is somewhat different than the initialization of a pointer to an integer object. For example,

```
char *string = "This is a string."
```

declares that **string** is a pointer to a **char**. It then stores the string literal **This is a string** in memory and fills **string** with the address of its first character. **string** can then be passed to functions to access the string, or you can step through the string by incrementing **string** until its contents point to the null character at the end of the string.

Another way to initialize a pointer is to fill it with a value returned by a function that returns a pointer. For example, the code

```
extern void *malloc(size_t variable);
char *example;
   . . .
example = (char *)malloc(50);
```

uses the function **malloc** to allocate 50 bytes of dynamic memory and then initializes **example** to the address that **malloc** returns.

Reading What a Pointer Points To

The indirection operator '*' can be used to read the object to which a pointer points. For example,

```
int number;
int *pointer;
   . . .
pointer = &number;
   . . .
printf("%d\n", *pointer);
```

uses **pointer** to access the contents of **number**.

When a pointer points to a structure, the elements within the structure can be read by using the structure offset operator '->'. See the entry for **->** for more information.

Pointers to Functions

A pointer can also contain the address of a function. For example,

```
char *(*example)();
```

declares **example** to be a pointer to a function that returns a pointer to a **char**.

This declaration is quite different from:

```
char **different();
```

LEXICON

The latter declares that **different** is a function that returns a pointer to a pointer to a **char**.

The following demonstrates how to call a function via a pointer:

```
(*example)(arg1, arg2);
```

Here, the '*' takes the contents of the pointer, which in this case is the address of the function, and uses that address to pass to a function its list of arguments.

A pointer to a function can be passed to another function as an argument. The library functions **bsearch** and **qsort** both take function pointers as arguments. A program may also use of arrays of pointers to functions.

void *

void * is the generic pointer; it replaces **char** * in that role. A pointer may be cast to **void** * and then back to its original type without any change in its value. **void** * is also aligned for any type in the execution environment.

For more information on the use of the generic pointer, see **void**.

Pointer Conversion

One type of pointer may be converted, or *cast*, to another. For example, a pointer to a **char** may be cast to a pointer to an **int**, and vice versa.

Any pointer may be cast to type **void** * and back again without its value being affected in any way. Likewise, any pointer of a scalar type may be cast to its corresponding **const** or **volatile** version. The qualified pointers are equivalent to their unqualified originals.

Pointers to different data types are compatible in expressions, but only if they are cast appropriately. Using them without casting produces a *pointer-type mismatch*. The translator should produce a diagnostic message when it detects this condition.

Pointer Arithmetic

Arithmetic may be performed on all pointers to scalar types. Pointer arithmetic is quite limited and consists of the following:

1. One pointer may be subtracted from another.
2. An **int** or a **long**, either variable or constant, may be added to a pointer or subtracted from it.
3. The operators **++** or **--** may be used to increment or decrement a pointer.

No other pointer arithmetic is permitted.

Cross-references

Standard, §3.1.2.5, §3.2.2.1, §3.2.2.3, §3.3.2.2-3, §3.5.4.1
The C Programming Language, ed. 2, pp. 93*ff*

See Also

NULL, types, void

Notes

The Rationale cautions against using NULL as an explicit argument to any function that expects a pointer on the grounds that, under some environments, pointers to different data types may be of different lengths. All such problems will be avoided if a function prototype is within the scope of the function call. Then, NULL will be transformed automatically to the proper type of pointer. See *function prototype* for more information.

pointer declarators — Definition

A *pointer declarator* declares a pointer.

An asterisk '*' marks an identifier as being a pointer. For example:

```
int *example;
```

states that **example** is a pointer to **int**. Likewise, the use of two asterisks marks an identifier as being a pointer to a pointer. For instance,

```
int **example;
```

declares a pointer to a pointer to an **int**. It is sometimes helpful to read a C declarator backwards, i.e., from right to left, to decipher it.

A pointer declarator may be modified by the type qualifiers **const** or **volatile**. For example, the declarator

```
int *const example;
```

declares that **example** is a constant pointer to a variable value of type **int**, whereas the declaration

```
const int *example;
```

declares that **example** is a variable pointer to a constant integer value. The same syntax applies to **volatile**. The declaration

```
const int *const example;
```

declares a constant pointer to a constant **int**.

Cross-references

Standard, §3.5.4.1
The C Programming Language, ed. 2, p. 94

See Also

***, declarators, pointer**

poke() — Extended function (libc)

Insert a word into memory
unsigned poke(unsigned *offs***, unsigned** *seg***, unsigned** *data***);**

poke writes a word (two bytes) into an arbitrary location in memory. It writes the word *data* into the memory location given by the segment *seg* and offset *offs*, and returns *data*.

If your program is compiled into SMALL model, you can supply the full offset/segment pair by using the macro **PTR**. See its entry in the Lexicon for more information.

Example

This program will print a reverse video 'A' on an IBM-PC monochrome screen.

```
#include <stdlib.h>
main(void)
{
      /* '70' = reverse video, '41'= 'A' */
      poke(0x000, 0xB000, 0x7041);
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, pokeb, PTR, _zero

Notes

Because memory is not protected on the i8086, be careful that you have the correct memory location when using **poke**.

pokeb() — Extended function (libc)

Insert a byte into memory
unsigned pokeb(unsigned *offs***, unsigned** *seg***, unsigned** *data***);**

pokeb writes a byte of data into an arbitrary location of memory. It writes *data* into the memory location given by the segment *seg* and offset *offs*, and returns *data*.

If your program is compiled into SMALL model, you can supply the full offset/segment pair by using the macro **PTR**. See its entry in the Lexicon for more information.

Example

This program will print a 'W' in the upper left-hand corner of an IBM-PC monochrome screen.

```
#include <stdlib.h>
main(void)
{
      pokeb(0x0000, 0xB000, 0x57);
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, poke, PTR

Notes

Because memory is not protected on the i8086, be careful that you have the correct memory location when using **pokeb**.

pokeb is not described in the ANSI Standard. All programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

port — Definition

A **port** passes data to and receives data from a remote device.

See Also

aux, com1, fclose, FILE, fopen, lpt1, prn, stream

portability — Definition

The term *portability* refers to a program's ability to be translated and executed under more than one environment. The Standard is designed so that if you adhere to it strictly, you will, in the words of the Rationale, "have a 'fighting chance' to make powerful C programs that are also highly portable ...."

Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

- Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**.

- Do not write routines that depend on a particular order of code evaluation, particular byte ordering, or particular length of data types, except for those specified within the Standard.
- Do not write routines that play tricks with a machine's "magic characters". For example, writing a routine that depends on a file's ending with **<ctrl-Z>** instead of **EOF** ensures that that code can run only under operating systems that recognize this magic character.
- Always use constant such as **EOF** and make full use of **#define** statements.
- Use headers to hold all machine-dependent declarations and definitions.
- Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value. This avoids unforeseen problems with undefined return values.
- Do not assume that all varieties of pointer are the same or can point anywhere. On some machines, for example, a **char** * is longer than an **int** *. On others, a function pointer aims at a different space than does a data pointer.
- NULL should not be used as an explicit argument to any function that expects a pointer because, under some environments, pointers to different data types may be of different lengths. All such problems are avoided if a function prototype is within the scope of the function call. Then, NULL is transformed automatically to the proper type of pointer.
- Always exit or return explicitly from **main**, even when the program has run successfully to its end.
- **int** is the register size of the machine. Use **short** or **long** wherever size is a consideration.
- Inevitably, you will have code that is not 100% portable. Try to separate code that is machine-specific or operating-system specific into its own file.

Cross-reference

The C Programming Language, ed. 2, p. 3

See Also

behavior, Definitions

pow() — Mathematics (libm)

Raise one number to the power of another

#include <math.h>
double pow(double *z***, double** *x***);**

pow calculates and returns *z* raised to the power of *x*.

Cross-references

Standard, §4.5.5.1
The C Programming Language, ed. 2, p. 251

See Also

mathematics, sqrt

Notes

A domain error occurs if *z* equals zero, if *x* is less than or equal to zero, or if *z* is less than zero and *x* is not an integer.

pr — Command

Paginate and print files
pr [*options***] [***file …***]**

The command **pr** paginates each *file* and writes it into the standard output. The file name '-' means standard input. If no *file* is specified, **pr** reads the standard input.

On each page, **pr** writes a header that gives the date, file name, and page and line numbers. **pr** may be used with the following options.

+*n* Skip the first *n* pages of each input file.

-*n* Print the text in *n* columns. This is used to print out material that was typed in one or more columns.

-h *header*
Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed within quotation marks.

-l*n* Set the page length to *n* lines (default, 66).

-m Print the texts simultaneously in separate columns. Each text will be assigned an equal amount of width on the page. Any lines longer than that will be truncated. This is used to print several similar texts or listings simultaneously.

-n Number each line as it is printed.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left justified in a fixed-width field.

-t Suppress the printing of the header on each page, as well as the header and footer space.

-w*n* Set the page width to *n* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 256 columns.

See Also

commands

preprocessing numbers — Definition

A *preprocessing number* is one of the intermediate lexical elements handled during translation phases 1 through 6. As semantic analysis occurs in translation phase 7, the set of valid preprocessing numbers forms a superset of valid C numeric tokens.

A preprocessing number is any floating constant or integer constant. A preprocessing number begins with either a digit or a period '.', and may consist of digits, letters, periods, and the character sequences **e+**, **e-**, **E+**, or **E-**.

Cross-reference

Standard, §3.1.8

See Also

lexical elements, preprocessing, token, translation phases

printf() — STDIO (libc)

Format and print text into the standard output stream
#include <stdio.h>
int printf(const char **format* **...);**

printf constructs a formatted string and writes it into the standard output stream.

format points to a string that can contain text, character constants, and one or more *conversion specifications*. A conversion specification describes how a particular type is to be converted into text.

Each conversion specification is introduced by the percent sign '%', and is followed, in order, by one or more of the following:

- A flag, which modifies the meaning of the conversion specification.
- An integer, which sets the minimum width of the field upon which the text is printed.
- A period and an integer, which sets the precision with which a number is printed.
- One of the following modifiers: **h**, **l**, or **L**. Their use is discussed below.
- Finally, a character that specifies the type of conversion to be performed. These are given below. This is the only element required after a '%'.

After *format* can come one or more arguments. There should be one argument for each conversion specification within *format* of the type appropriate to its conversion specifier. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments, being, respectively, an **int**, a **long**, and a pointer to **char**.

If there are fewer arguments than conversion specifications, then **printf**'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 **printf** is undefined.

If it writes the formatted string correctly, **printf** returns the number of characters written; otherwise, it returns a negative number. **printf** can generate a string that is up to at least 509 characters long.

The following sections describe in detail the elements of the conversion specification.

Conversion Specifiers

If *format* includes any conversion specifiers other than the ones shown below, the behavior is undefined. If a **union**, an aggregate, or a pointer to a **union** or an aggregate is used as an argument, behavior is undefined.

c Convert the **int** or **unsigned int** argument to a character.

d Convert the **int** argument to signed decimal notation.

D Convert the **long** argument to signed decimal. This specifier is not described in the Standard. Programs that use it do not comply strictly with the Standard, and may not be portable to other compilers or environments.

e Convert the **double** argument to exponential form. The format is

*[-]d.dddddd***e+/-***dd*

At least one digit always appears to the left of the decimal point and as many as *precision* digits to the right of it (default, six). If the precision is zero, then no decimal point is printed.

LEXICON

E Same as **e**, except that 'E' is used instead of 'e'.

f Convert the **double** argument to a string of the form

[-]d.dddddd

At least one digit always appears to the left of the decimal point, and as many as *precision* digits to the right of it (default, six). If the precision is zero, then no decimal point is printed.

g Convert the **double** argument to either of the formats **e** or **f**. The number of significant digits is equal to the precision set earlier in the conversion specification. Normally, this conversion selects conversion type **f**. It selects type **e** only if the exponent that results from such a conversion is either less than -4 or greater than the precision.

G Same as **g**, except that it selects between conversion types **E** and **f**.

i Same as **d**.

n This conversion specification takes a pointer to an integer, into which it writes the number of characters **printf** has generated to the current point within *format*. It does not affect the string **printf** generates.

o Convert the **int** argument to unsigned octal digits.

O Convert the **long** argument to unsigned octal. This specifier is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

p This conversion sequence takes a pointer to **void**. It translates the pointer into a set of characters and prints them. What it generates is defined by the implementation.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char *** that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string. This is roughly equivalent to the library function **vprintf**.

This specifier is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

s Print the string to which the corresponding argument points; the argument must point to a C string. It prints either the number of characters set by the precision, or to the end of the string, whichever is less. If no precision is specified, then the entire string is printed.

u Convert the **int** argument to unsigned decimal digits.

U Convert the **long** argument to unsigned decimal. This specifier is not described in the ANSI Standard. Programs that use it do not conform strictly to the Standard, and may not be portable to other compilers or environments.

x Convert the **int** argument to unsigned hexadecimal characters. The values 10, 11, 12, 13, 14, and 15 are represented, respectively, by 'a', 'b', 'c', 'd', and 'e'.

X Same as **x**, except that the values 10, 11, 12, 13, 14, and 15 are represented, respectively, by 'A', 'B', 'C', 'D', and 'E'. In previous releases of **Let's C**, this specifier converted a **long** to unsigned hexadecimal characters. This change was made to conform to the ANSI Standard, and may require that some code be rewritten.

The description of each conversion specifier assumes that it will be used with an argument whose type matches the type that the specifier expects. If the argument is of another type, it is cast to the type expected by the specifier. For example,

```
float f;
printf("%d\n", f);
```

will truncate **f** to an **int** before printing its value.

Flags

The '%' that introduces a conversion specification may be followed immediately by one or more of the following flags:

- Left-justify text within its field. The default is to right-justify all output text within its field.

+ Precede a signed number with a plus or minus sign. For example,

```
printf("%+d %+d\n", -123, 123);
```

yields the following when executed:

```
-123 +123
```

<space>
If the first character of a signed number is its sign, then that sign is appended to the beginning of the text string generated; if it is not a sign, then a space is appended to the beginning of the text string. For example,

```
printf("% d\n", -123);
printf("% d\n",  123);
```

generates the following:

```
-123
 123
```

This flag can be used with every conversion specifier for a numeric data type. It forces **printf** to use a special format that indicates what numeric type is being printed. The following gives the effect of this flag on each appropriate specifier:

e always retain decimal point
E always retain decimal point
f always retain decimal point
F always retain decimal point
g always retain decimal point; keep trailing zeroes
G always retain decimal point; keep trailing zeroes
x print '0x' before the number
X print '0X' before the number

Any specified precision is expanded by the appropriate amount to allow for the printing of the extra character or characters. Using '#' with any other conversion specifier yields undefined results.

0 When used with the conversion specifiers **d**, **e**, **E**, **f**, **g**, **G**, **i**, **o**, **u**, **x**, or **X**, a leading zero indicates that the field width is to be padded with leading zeroes instead of spaces. If precision is indicated with the specifiers **d**, **i**, **o**, **u**, **x**, **X**, then the **0** flag is ignored; it is also ignored if it is used with the **-** flag. If this flag is used with any conversion specifier other than the ones listed above, behavior is undefined.

Field Width

The field width is an integer that sets the minimum field upon which a formatted string is printed.

LEXICON

If a field width is specified, then that many characters-worth of space is reserved within the output string for that conversion. When the text produced by the conversion is *smaller* than the field width, spaces are appended to the beginning of the text to fill out the difference; this is called *padding*. Beginning the field width with a zero makes the padding character a '0' instead of a space. When the text is *larger* than the allotted field width, then the text is given extra space to allow it to be printed. Setting the field width never causes text to be truncated.

By default, text is set flush right within its field; using the '-' flag sets the text flush left within its field.

Using an asterisk '*' instead of an integer forces **printf** to use the corresponding argument as the field width. For example,

```
char *string = "Here's a number:";
int    width = 12;
int  integer = 123;
printf("%s%*d\n", string, width, integer);
```

produces the following text:

```
Here's a number:            123
```

Here, **width** was used to set the field width, so 12 spaces were used to pad the formatted integer.

Precision

The precision is indicated by a decimal point followed by a number. If a decimal point is used without a following number, then it is regarded as equivalent to '.0'.

The precision sets the number of characters to be printed for each conversion specifier. Setting the precision to *n* affects each conversion specifier as follows:

d print at least *n* digits
e print *n* digits after decimal point
E print *n* digits after decimal point
f print *n* digits after decimal point
g print no more than *n* significant digits
G print no more than *n* significant digits
i print at least *n* digits
o print at least *n* digits
s print no more than *n* characters
u print at least *n* digits
x print at least *n* digits
X print at least *n* digits

The precision differs from the field width in that the field width controls the amount of space set aside for the text, whereas the precision controls the number of characters to be printed. If the amount of padding called for by the precision conflicts with that called for by the field width, the amount called for by the precision is used.

Using an asterisk '*' instead of an integer forces **printf** to use the corresponding argument as the precision.

For example, this code

```
int   foo = 12345;
float bar = 12.345;
char *baz = "Hello, world";

printf("Example 1: %7.6d\n", foo);
printf("Example 2: %7.6f\n", bar);
printf("Example 3: %7.6s\n", baz);
```

produces the following text when executed:

```
Example 1:  012345
Example 2: 12.345000
Example 3:  Hello,
```

Modifiers

The following three modifiers may be used before a conversion specifier:

h When used before the specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument is a **short int** or an **unsigned short int**. When used before **n**, it indicates that the corresponding argument is a **short int**. In implementations where **short int** and **int** are synonymous, it is not needed; however, it is useful in writing portable code.

l When used before **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument is a **long int** or an **unsigned long int**. When used before 'n', it indicates that the corresponding argument is a **long int**. In implementations where **long int** and **int** are synonymous, it is not needed; however, it is useful in writing portable code.

L When used before **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument is a **long double**.

Using **h**, **l**, or **L** before a conversion specifier other than the ones mentioned above results in undefined behavior.

Default argument promotions are performed on the arguments. There is no way to suppress this.

Example

This example implements a mini-interpreter for **printf** statements. It is a convenient tool for seeing exactly how some of the **printf** options work. To use it, type a **printf** conversion specification at the prompt. The formatted string will then appear. To reuse a format identifier, simply type **<return>**.

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

/* the replies go here */
static char reply[80];

/* ask for a string and echo it in reply. */
char *
askstr(char *msg)
{
      printf("Enter %s ", msg);
      fflush(stdout);

      if(gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return(reply);
}
```

LEXICON

```
main(void)
{
      char fid[80], c;

      /* initialize to an invalid format identifier */
      strcpy(fid, "%Z");

      for(;;) {
            askstr("format identifier");
            /* null reply uses previous FID */
            if(reply[0])
                  /* leave the '%' */
                  strcpy(fid + 1, reply);

            switch(c = fid[strlen(fid) - 1]) {
            case 'd':
            case 'i':
                  askstr("signed number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, atoi(reply));
                  break;

            case 'o':
            case 'u':
            case 'x':
            case 'X':
                  askstr("unsigned number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, (unsigned)atol(reply));
                  break;

            case 'f':
            case 'e':
            case 'E':
            case 'g':
            case 'G':
                  printf(fid, atof(askstr("real number")));
                  break;

            case 's':
                  printf(fid, askstr("string"));
                  break;

            case 'c':
                  printf(fid, *askstr("single character"));
                  break;

            case '%':
                  printf(fid);
                  break;

            case 'p':
                  /* print pointer to format id */
                  printf(fid, fid);
                  break;
```

```
            case 'n':
                  printf("n not implemented");
                  break;

            default:
                  printf("%c not valid", c);
            }

      printf("\n");
      }
}
```

Cross-references

Standard, §4.9.6.3
The C Programming Language, ed. 2, p. 244

See Also

fprintf, sprintf, STDIO, vfprintf, vprintf, vsprintf

Notes

printf can construct and output a string at least 509 characters long.

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

Because the **printf** routines that print floating-point numbers are quite large, they are included only optionally. If you wish to have **printf** print **float**s or **double**s, you must compile your program with the **-f** option to the **cc** command. See **cc** for more details.

prn — Operating system device

MS-DOS logical device for parallel port

MS-DOS gives names to its logical devices. **Let's C** uses these names, to allow the **STDIO** library routines to access these devices via MS-DOS.

prn is the logical device for the the parallel port.

Example

The following example checks to see if the parallel port can be opened.

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

main(void)
{
      FILE *fp, *fopen();
      if ((fp = fopen("prn","w")) != NULL)
            fprintf(fp,"prn enabled.\n");
      else printf("prn cannot open.\n");
      return EXIT_SUCCESS;
}
```

See Also

aux, com1, con, lpt1, nul, operating system device

process — Definition

A **process** is a program in the state of execution.

See Also

daemon, Definitions, file

program startup — Definition

Program startup occurs when the execution environment invokes the program. Execution begins, and continues until program termination occurs. A program's execution may be suspended by the environment and resumed at a later time. The program, however, only starts once.

Cross-reference

Standard, §2.1.2

See Also

Environment, program termination

program termination — Definition

Program termination occurs when a program stops executing and returns control to the execution environment. Program termination may be triggered when the program calls either of the functions **abort** or **exit**, when **main** returns, when the environment or hardware raises a signal, or when program termination has been requested by some other program or event.

There are two types of termination: *unsuccessful* and *successful*.

Unsuccessful termination occurs either when a program aborts due to a significant problem in its operation (such as memory violation or division by zero), or when the program did not function as expected (such as when a requested file cannot be found).

A program indicates unsuccessful termination either by calling the function **exit** with the argument **EXIT_FAILURE**, by calling the function **abort**, or by using the function **raise** to generate the signal **SIGABRT**. **exit** is used to stop a program that cannot perform correctly, but does not threaten the integrity of the environment. **abort** and **raise** are used to stop a program that has gone seriously wrong.

Successful termination is declared to occur when the program runs to its conclusion correctly. A program indicates successful termination by calling the function **exit** with the argument **EXIT_SUCCESS**, or when **main** returns **EXIT_SUCCESS**.

Cross-reference

Standard, §2.1.2, §4.10.4.1, §4.10.4.3

See Also

abort, environment, execution environment, exit, EXIT_FAILURE, EXIT_SUCCESS, main, program startup, signal

pun — Definition

In the context of C, a **pun** occurs when a programmer uses one data form interchangeably with another. Puns are supported by C's willingness to apply implicit conversion rules.

A pun most often occurs unintentionally when the programmer fails to prototype or declare a function that returns a pointer. By default, the function is then assumed to return an **int**, and is handled as such. No trouble will arise if the program is run on a machine that defines an **int** and a pointer to have the same length (e.g., i8086 SMALL model); however, such code cannot be transported to an environment in which this is not the case (e.g., i8086 LARGE model).

See Also

Definitions, pointer, portability

punctuators — Overview

A *punctuator* is a symbol that has syntactic meaning but does not represent an operation that yields a value. All lexical elements that do not fall into another meaningful category are lumped together as punctuators.

Most often, a punctuator is used to mark or delimit an identifier or a portion of code, rather than modify it.

The set of punctuators consists of the following:

[]	Mark an array/delimit its size
()	Mark a parameter/argument list
{}	Delimit a block of code or a function
*	Identify a pointer type in a declaration
,	Delimit a function argument
:	Delimit a label
;	Mark end of a statement
...	(ellipsis) Indicate function takes flexible number of arguments
#	Indicate a preprocessor directive

The punctuators

```
{}   []   ()
```

must be used in pairs.

A symbol that acts as a punctuator may also act as an operator, depending upon its context.

Cross-reference

Standard, §3.1.6

See Also

lexical elements, operators, statements

putc() — STDIO (stdio.h)

Write a character into a stream
#include <stdio.h>
int putc(int *character*, **FILE** **fp*);

putc writes *character* into the stream pointed to by *fp*.

putc returns *character* if it was written correctly. Otherwise, it sets the error indicator for *fp* and returns **EOF**.

Example

This example writes newline characters into a file until the disk is full. Because this example uses the function **tmpfile**, the file it writes disappears when the program terminates. It is not recommended that you run this program on a multi-user system.

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

LEXICON

```
main(void)
{
      long count;
      FILE *tmp;

      if((tmp = tmpfile()) == NULL) {
            fprintf(stderr, "Can't open tmp file\n");
            exit(EXIT_FAILURE);
      }

      for(count = 0; putc('\n', tmp) != EOF; count++)
            ;

      fprintf(stderr, "We wrote %ld characters\n", count);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.8
The C Programming Language, ed. 2, p. 247

See Also

getc, getchar, gets, putchar, puts, STDIO, ungetc

Notes

Because **putc** is implemented as a macro, *fp* may be read more than once. Therefore, one should beware of the side-effects of evaluating the argument more than once, especially if the argument itself has side-effects. See the entry for **macro** for more information. Use **fputc** if this behavior is not acceptable.

putchar() — STDIO (stdio.h)

Write a character into the standard output stream
#include <stdio.h>
int putchar(int *character***);**

putchar writes a character into the standard output stream. It is equivalent to:

```
putc(character, stdout);
```

putchar returns *character* if it was written correctly. If *character* could not be written, **putchar** sets the error indicator for the stream associated with stdout and returns **EOF**.

Example

This example prints all of the printable ASCII characters.

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

main(void)
{
      char c;

      for(c = ' '; putchar(c) <= '}'; c++)
            ;

      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.9
The C Programming Language, ed. 2, p. 247

See Also

getc, getchar, gets, puts, STDIO, ungetc

puts() — STDIO (libc)

Write a string into the standard output stream
#include <stdio.h>
int puts(char ******string***);**

puts replaces the null character at the end of *string* with a newline character, and writes the result into the standard output stream.

puts returns a non-negative number if it could write *string* correctly; otherwise, it returns **EOF**. In previous versions of **Let's C**, **puts** returned nothing. This was changed to conform to the ANSI Standard.

Example

This example uses **puts** to print a string into the standard output stream.

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

main(void)
{
      puts("Hello world.");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.10
The C Programming Language, ed. 2, p. 247

See Also

putc, putchar, STDIO, ungetc

Notes

For historical reasons, **fputs** writes *string* unchanged, whereas **puts** appends a newline character.

putw() — Extended macro (xstdio.h)

Write word to stream
#include <xstdio.h>
short putw(short *word***, FILE** ******fp***);**

The macro **putw** writes *word* onto the file stream *fp*. It returns the value written.

putw differs from **putc** in that **putw** writes an **int**, whereas **putc** writes a **char** that is promoted to an **int**.

putw returns **EOF** when an error occurs. You may need to call **ferror** to distinguish this value from a genuine end-of-file flag.

See Also

extended STDIO, ferror, xstdio.h

LEXICON

Notes

Because **putw** is implemented as a macro, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

putw is not described in the ANSI Standard. A program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

The Standard requires that ANSI headers contain only functions that are described within the Standard. Therefore, **putw** has been moved from **stdio.h** to **xstdio.h**.