F

fabs() — Mathematics (libm)

Compute absolute value
#include <math.h>
double fabs(double *z*);

fabs calculates and returns the absolute value for a double-precision floating-point number. It returns *z* if *z* is zero or positive, and it returns -*z* if *z* is negative.

Example

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

Cross-references

Standard, §4.5.6.2
The C Programming Language, ed. 2, p. 251

See Also

abs, ceil, floor, fmod, mathematics

false — Definition

In the context of a C program, an expression is *false* if it is zero.

See Also

Definitions, true

fclose() — STDIO (libc)

Close a stream
#include <stdio.h>
int fclose(FILE ****fp*);**

fclose closes the stream pointed to by *fp*.

fclose flushes all of *fp*'s output buffers. Unwritten buffered data are handed to the host environment for writing into *fp*, and unread, buffered data are thrown away. It then dissociates the stream pointed to by *fp* from the file (i.e., "closes" the file). If the buffer associated with *fp* was allocated, it is then de-allocated.

The function **exit** calls **fclose** to close all open streams.

fclose returns zero if it closed *fp* correctly, and **EOF** if it did not.

Example

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

Cross-references

Standard, §4.9.5.1
The C Programming Language, ed. 2, p. 162

See Also

fclose, fflush, fopen, freopen, setbuf, setvbuf, STDIO

Notes

The function **exit** closes all open streams, which flushes their buffers.

fcvt() — Extended function (libc)

Convert floating-point numbers to strings
char *fcvt(double *d*, **int** *w*, **int** **dp*, **int** **signp***);**

fcvt converts floating point numbers to ASCII strings. Its operation resembles that of the **%f** operator to **printf**. It converts *d* into a null-terminated string of decimal digits with a precision (i.e., the number of characters to the right of the decimal point) of *w*. It rounds the last digit and returns a pointer to the result.

On return, **fcvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string: to the right if positive, and to the left if negative. Finally, it sets *signp* to point to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt** rounds the result to the FORTRAN F-format.

Example

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

See Also

ecvt, extended miscellaneous, frexp, gcvt, ldexp, modf, printf

Notes

fcvt performs conversions within static string buffers that are overwritten by each execution.

fdopen() — Extended function (libc)

Open a stream for standard I/O
#include <xstdio.h>
FILE *fdopen(short *fd*, **char ****type***);**

fdopen allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open**, **creat**, or **dup**.

type is the manner in which you wish to open *fd*, as follows:

r	Read a file
w	Write into a file
a	Append onto a file

fdopen returns **NULL** if it cannot allocate a **FILE** structure.

Example

The following example obtains a file descriptor with **open**, and then uses **fdopen** to build a pointer to the **FILE** structure.

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

/* prototype for extended function */
extern int open(char *file, int type);

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

```
main(int argc, char *argv[])
{
      extern FILE *fdopen();
      FILE *fp;
      short fd;
      short holder;

      if (--argc != 1)
            fatal("Usage: example filename");

      if ((fd = open(argv[1], 0)) == -1)
            fatal("open failed.");
      if ((fp = fdopen(fd, "r")) == NULL)
            fatal("fdopen failed.");

      while ((holder = fgetc(fp)) != EOF) {
            if ((holder > '\177') && (holder < ' '))
                  switch(holder) {
                  case '\t':
                  case '\n':
                        break;
                  default:
                        fprintf(stderr, "Seeing char %d\n", holder);
                        exit(EXIT_FAILURE);
                  }

            fputc(holder, stdout);
      }
      return(EXIT_SUCCESS);
}
```

See Also

creat, dup, fopen, open, STDIO

Notes

Currently, 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fdopen 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.

feof() — STDIO (stdio.h)

Examine a stream's end-of-file indicator
#include <stdio.h>
int feof(FILE ****fp*);**

feof examines the end-of-file indicator for the stream pointed to by *fp*. It returns zero if the indicator shows that the end of file has *not* been reached, and returns a number other than zero if the indicator shows that it has.

Examples

This example checks whether a file can be read directly to the end.

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

```
main(int argc, char *argv[])
{
      long size;
      FILE *ifp;

      if(argc != 2) {
            printf("usage: example inputfile\n");
            exit(EXIT_FAILURE);
      }

      if((ifp = fopen(argv[1], "rb")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      for(size = 0; fgetc(ifp) != EOF; size++)
            ;

      if(feof(ifp))
            printf("EOF at character %ld\n", size);

      if(ferror(ifp)) {
            printf("Error at character %ld\n", size);
            perror(NULL);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.10.2
The C Programming Language, ed. 2, p. 176

See Also

Notes

feof is often used with **getw** or **fgetw**, to distinguish a value of -1 from **EOF**.

ferror() — STDIO (libc)

Examine a stream's error indicator
#include <stdio.h>
int ferror(FILE * *fp***);**

ferror examines the error indicator for the stream pointed to by *fp*. It returns zero if an error has occurred on *fp*, and a number other than zero if one has not.

Cross-references

Standard, §4.9.10.3
The C Programming Language, ed. 2, p. 164

See Also

clearerr, feof, perror, STDIO

Notes

Any error condition noted by **ferror** will persist either until the stream is closed, until **clearerr** is used to clear it, or until the file-position indicator is reset with **rewind**.

fflush() — STDIO (libc)

Flush output stream's buffer
#include <stdio.h>
int fflush(FILE **fp***);**

fflush flushes the buffer associated with the file stream pointed to by *fp*. If *fp* points to an output stream, then **fflush** hands all unwritten data to the host environment for writing into *fp*. If, however, *fp* points to an input stream, behavior is undefined.

With **Let's C**, **stdout** is buffered. Here, **fflush** can be used to write a prompt that is not terminated by a newline.

fflush returns zero if all goes well, and returns **EOF** if a write error occurs.

The function **exit** calls **fclose** to flush all output buffers before the program exits.

Example

This example asks for a string and returns it in reply.

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

static char reply[80];
char *
askstr(char *msg)
{
      printf("Enter %s ", msg);
      /* required by the absence of a \n */
      fflush(stdout);
      if(gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return(reply);
}

main(void)
{
      for(;;)
            if(!strcmp(askstr("a string"), "quit"))
                  break;
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.2
The C Programming Language, ed. 2, p. 242

See Also

fclose, fopen, freopen, setbuf, setvbuf, STDIO

fgetc() — STDIO (libc)

Read a character from a stream
#include <stdio.h>
int fgetc(FILE **fp***);**

fgetc reads a character from the stream pointed to by *fp*. Each character is read initially as an **unsigned char**, then converted to an **int** before it is passed to the calling function. **fgetc** then advances the file-position indicator for *fp*.

fputc returns the character read from *fp*. If the file-position indicator is beyond the end of the file to which *fp* points, **fputc** returns **EOF** and sets the end-of-file indicator. If a read error occurs, **fgetc** returns **EOF** and the stream's error indicator is set.

Example

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

Cross-references

Standard, §4.9.7.1
The C Programming Language, ed. 2, p. 246

See Also

fgets, fgetw, getc, getchar, gets, getw, STDIO

fgetpos() — STDIO (libc)

Get value of file-position indicator
#include <stdio.h>
int fgetpos(FILE **fp*, **fpos_t** **position***);**

fgetpos copies the value of the file-position indicator for the file stream pointed to by *fp* into the area pointed to by *position*. *position* is of type **fpos_t**, which is defined in the header **stdio.h**. The information written into *position* can be used by the function **fsetpos** to return the file-position indicator to where it was when **fgetpos** was called.

fgetpos returns zero if all went well. If an error occurred, **fgetpos** returns nonzero and sets the integer expression **errno** to the appropriate value. See **errno** for more information on its use.

Example

This example seeks to a random line in a very large file.

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

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

      if(errno)
            perror(NULL);
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      int c;
      long count;
      FILE *ifp, *tmp;
      fpos_t loc;
      if(argc != 2)
            fatal("usage: fscanf inputfile\n");
      if((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);
      if((tmp = tmpfile()) == NULL)
            fatal("Cannot build index file");

      /* seed random-number generator */
      srand((unsigned int)time(NULL));

      for(count = 1;!feof(ifp); count++) {
            /* for monster files */
            if(fgetpos(ifp, &loc))
                  fatal("fgetpos error");

            if(fwrite(&loc, sizeof(loc), 1, tmp) != 1)
                  fatal("Write fail on index");
            rand();
            while('\n' != (c = fgetc(ifp)) && EOF != c)
                  ;
      }

      count = rand() % count;
      fseek(tmp, count * sizeof(loc), SEEK_SET);

      if(fread(&loc, sizeof(loc), 1, tmp) != 1)
            fatal("Read fail on index");

      fsetpos(ifp, &loc);
      while((c = fgetc(ifp)) != EOF) {
            if('@' == c)
                  putchar('\n');
            else
                  putchar(c);

            if('\n' == c)
                  break;
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.9.1
The C Programming Language, ed. 2, p. 248

See Also

fseek, fsetpos, ftell, rewind, STDIO

Notes

The Standard introduced **fgetpos** and **fsetpos** to manipulate a file whose file-position indicator cannot be stored within a **long**. Under MS-DOS, **fgetpos** behaves the same as the function **ftell**.

fgets() — STDIO (libc)

Read a line from a stream
#include <stdio.h>
char *fgets(char ****string*, int** *n***, FILE ****fp***);**

fgets reads characters from the stream pointed to by *fp* into the area pointed to by *string* until either *n*-1 characters have been read, a newline character is read, or the end of file is encountered. It retains the newline, if any, and appends a null character to the end of of the string.

fgets returns the pointer *string* if its read was performed successfully. It returns NULL if it encounters the end of file or if a read error occurred. When a read error occurs, the contents of *string* are indeterminate.

Example

This example displays a text file. It breaks up lines that are longer than 78 characters.

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

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

      if(errno)
            perror(NULL);
      if(format!=NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      char buf[79];
      FILE  *ifp;

      if(argc != 2)
            fatal("usage: fgets inputfile\n");
      if((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);

      while(fgets(buf, sizeof(buf), ifp) != NULL) {
            printf("%s", buf);
            if(strchr(buf, '\n') == NULL)
                  printf("\\\n");
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.2
The C Programming Language, ed. 2, p. 247

See Also

fgetc, fgetw, getc, getchar, gets, getw, STDIO

fgetw() — Extended function (libc)

Read integer from stream
#include <xstdio.h>
short fgetw(FILE ****fp*);**

fgetw is a function that reads and returns a word (**short int**) from the stream pointed to by *fp*.

fgetw returns **EOF** if an error occurs. A call to **feof** or **ferror** may be necessary to distinguish this value from a genuine end-of-file signal.

Example

This example copies one binary file into another. It demonstrates the functions **fgetw** and **fputw**.

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

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

main(int argc, char *argv[])
{
      FILE *fpin, *fpout;
      int word;

      if(argc != 3)
            fatal("Usage: example sourcefile newfile");

      if ((fpin = fopen(infile, "rb")) == NULL)
            fatal("Cannot open output file");
      if ((fpout = fopen(outfile, "wb")) != NULL)
            fatal("Cannot open output file");

      while ((word = fgetw(fpin)) != EOF) {
            fputw(word, fpout);
            if (!ferror(fpin))
                  fatal("Read error");
      }

      fclose(fpin);
      fclose(fpout);
      return(EXIT_SUCCESS);
}
```

See Also

extended STDIO, fputw

Notes

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fgetw 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.

field — Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also

bit field, Definitions, struct

file — Definition

A *file* is a mass of bits that has been named and stored on a mass-storage device.

Opening a File

To read a file, alter its contents, or add data to it, a C program must use a *stream*. The term *opening a file* means to establish a stream through which the program can access the file. The stream governs the way data are accessed. The information the stream needs to access the file are encoded within a **FILE** object. Because environments vary greatly in the information they need to access a file, the Standard does not describe the internals of the **FILE** object. If a file does not exist when a program attempts to open it, then it is *created*. Because some environments distinguish the format for a text file from that for a binary file, the Standard distinguishes between opening a stream into text mode and opening it into binary mode.

To open a file, use the functions **fopen** or **freopen**. The former simply opens a file and assigns a stream to it. The latter reopens a file; that is, it takes the stream being used to access one file, assigns it to another file, and closes the original file. **freopen** can also be used to change the mode in which a file is accessed.

Buffering

When a file is opened, it is assigned a *buffer*. Access to the file are made through the buffer. Data written or, in some instances, read from the file are kept in the buffer temporarily, then transmitted as a block. This increases the efficiency with which programs communicate with the environment. To change the type of buffering performed, the size of the buffer used, or to redirect buffering to a buffer of your own creation, use the functions **setbuf** or **setvbuf**. See the entry for **buffer** for more information on the types of buffers used with files.

File-position Indicator

A file has a *file-position indicator* associated with it; this indicates the point within the file where it is being written to or read. Use of this indicator allows a program to walk smoothly through a file without having to use internal counters or other means to ensure that data are received sequentially. It also allows a program to access any point within a file "randomly" — that is, to access any given point in the file without having to walk through the entire file to reach it.

The manipulation of the file-position indicator can vary sharply between binary and text files. In general, the file-position indicator for a binary file is simply incremented as a character is read or written. For a text file, however, manipulation of the file-position indicator is defined by the implementation. This is due to the fact that different implementations represent end-of-line characters differently. To read the file-position indicator, use the functions **fgetpos** or **ftell**; to set it directly, use the functions **fseek** or **fsetpos**.

Error Conditions

When a file is being manipulated, a condition may occur that could cause trouble should the program continue to read or write that file. This could be an error, such as a read error, or the program may have read to the end of the file.

To help prevent such a condition from creating trouble, most environments use two indicators to signal when one has occurred: the *error indicator* and the *end-of-file* indicator. When an error occurs, the error indicator is set to a value that encodes the type of error that occurred; and when the end of the file is read, then the end-of-file indicator is set. By reading these indicators, a program may discover if all is going well. Under some implementations, however, a file may not be manipulated further unless both indicators are reset to their normal values.

To discover the setting of the end-of-file indicator, use the function **feof**. To discover the setting of the error indicator, use **ferror**. To reset the indicators to their normal values, use the function **clearerr**.

Closing a File

When you have finished manipulating a file, you should close it. To close a file means to dissociate it from the stream with which you had been manipulating it. When a file is closed, the buffer associated with its stream is flushed to ensure that all data intended for the file are written into it. To close a file, use the function **fclose**.

Cross-reference

Standard, §4.9.3

See Also

Definitions, STDIO, stdio.h, stream

Notes

When data are written into a binary file, the file is not truncated by the write. This allows writes to binary files to be performed at random positions throughout the file without truncating the file at the position written. Under **Let's C**, the same is true for text files.

file descriptor — Definition

A **file descriptor** is an integer between 1 and 20 that indexes an area in **_psbase**, which, in turn, points to the operating system's internal file descriptors. It is used by routines like **open**, **close**, and **lseek** to work with files. A file descriptor is *not* the same as a **FILE** stream, which is used by routines like **fopen**, **fclose**, or **fread**.

See Also

Definitions, file, FILE
Advanced MS-DOS, page 261

FILENAME_MAX — Manifest constant

Maximum length of file name
#include <stdio.h>

FILENAME_MAX is a that is defined in the header **stdio.h**. It gives the maximum length of a file name that the implementation can open.

Cross-references

Standard, §4.9.1
The C Programming Language, ed. 2, p. 242

See Also

fopen, STDIO, stdio.h

fileno() — Extended function (libc)

Get file descriptor
#include <xstdio.h>
short fileno(FILE ***fp*);**

fileno returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open** or **creat**. It is used by routines such as **fopen** used to create a **FILE** stream.

Example

This example reads a file descriptor and prints it on the screen.

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

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

main(int argc, char *argv[])
{
      FILE *fp;
      int fd;

      if (argc !=2)
            fatal("Usage: fd_from_fp filename");

      if ((fp = fopen(argv[1], "rw")) == NULL)
            fatal("Cannot open input file");

      fd = fileno(fp);
      printf("The file descriptor for %s is %d\n",
            argv[1], fd);
      return(EXIT_SUCCESS);
}
```

See Also

extended STDIO, FILE, file descriptor

Notes

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fileno 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.

float — C keyword

A **float** is a data type that represents a single-precision floating-point number. It is defined as being no larger than a **double**.

Like all floating-point numbers, a **float** consists of one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *mantissa*, or the number upon which the exponent works. The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the mantissa will be increased. The format of a **float** and the range of values that it can encode are set in the following macros, all of which are defined in the header **limits.h**:

FLT_DIG
: This holds the number of decimal digits of precision. This must be at least ten.

FLT_EPSILON
: Where *b* indicates the base of the exponent (default, two) and *p* indicates the precision (or number of base *b* digits in the mantissa), this macro holds the minimum positive floating-point number *x* such that 1.0 + *x* does not equal 1.0, *b*^1-*p*. This must be at least 1E-5.

FLT_MAX
: This holds the maximum representable floating-point number. It must be at least 1E+37.

FLT_MAX_EXP
This is the maximum integer such that the value of **FLT_RADIX** raised to its power minus one is a representable finite floating-point number.

FLT_MAX_10_EXP
This holds the maximum integer such that ten raised to its power is within the range of representable finite floating-point numbers. It must be at least +37.

FLT_MANT_DIG
This gives the number of digits in the mantissa.

FLT_MIN
This gives the minimum value encodable within a **float**. This must be at least 1E-37.

FLT_MIN_EXP
This gives the minimum negative integer such that when the value of **FLT_RADIX** is raised to that power minus one is a normalized floating-point number.

FLT_MIN_10_EXP
This gives the minimum negative integer such that ten raised to that power is within the range of normalized floating-point numbers.

Several formats are used to encode **float**s, including IEEE, DECVAX, and BCD (binary coded decimal). **Let's C** uses IEEE format throughout.

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit mantissa, as follows:

```
Sign  Exponent 1    Mantissa
|s   eeeeeee|e   fffffff|ffffffff|ffffffff|
   Byte 4      Byte 3     Byte 2   Byte 1
```

The exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero. An exponent and mantissa of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The most significant bit in the mantissa is always set to one and is not stored. It is usually called the "hidden bit".

The format for **double**s simply adds another 32 mantissa bits to the end of the **float** representation, as follows:

```
Sign  Exponent        Mantissa
|s    eeeeeee|e  fffffff|ffffffff|ffffffff|
   Byte 8      Byte 7     Byte 6   Byte 5

      ffffffff|ffffffff|ffffffff|ffffffff|
       Byte 4   Byte 3   Byte 2   Byte 1
```

IEEE Format

The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

Unlike the DECVAX format, IEEE format assigns special values to several floating point numbers. In the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

- A tiny exponent with a mantissa of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a mantissa of zero equals infinity, regardless of the setting of the sign bit.
- A tiny exponent with a mantissa greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a mantissa greater than zero is, by definition, not a number. These values can be used to handle special conditions.

An IEEE **double**, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit mantissa, as follows:

```
Sign     Exponent          Mantissa
|s    eeeeeee|eeee  ffff|ffffffff|ffffffff|
  Byte 8        Byte 7    Byte 6   Byte 5

      ffffffff|ffffffff|ffffffff|ffffffff|
       Byte 4   Byte 3   Byte 2   Byte 1
```

The exponent has a bias of 1,023. The rules of encoding are the same as for **float**s.

BCD Format

The BCD ("binary coded decimal") format is used in accounting to eliminate rounding errors that alter the worth of an account by a fraction of a cent. For that reason, BCD format consists of a sign, an exponent, and a chain of four-bit numbers, each of which is defined to hold the digits zero through nine.

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit digits, as follows:

```
Sign Exponent    Mantissa
 |s    eeeeeee|   dddd  dddd|dddd dddd|dddd dddd|
     Byte 4        Byte 3      Byte 2    Byte 1
```

A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit digits, as follows:

```
Sign    Exponent      Mantissa
|s    eeeeeee|eeee  dddd|dddd dddd|dddd dddd|
    Byte 8      Byte 7     Byte 6    Byte 5

      dddd dddd|dddd dddd|dddd dddd|dddd dddd|
        Byte 4    Byte 3    Byte 2    Byte 1
```

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when handling BCD numbers:

- A tiny exponent with a mantissa of zero equals zero.
- A tiny exponent with a mantissa of non-zero indicates a denormalized number.
- A huge exponent with a mantissa of zero indicates infinity.
- A huge exponent with a mantissa of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

Example

For an example of a program that uses **float**, see **sin**.

Cross-references

Standard, §2.2.4.2, §3.1.2.4, §3.1.3.1, §3.5.2
The C Programming Language, ed. 2, p. 211

See Also

double, float.h, long double, types

Notes

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.

float.h — Header

The header **float.h** defines a set of macros that return the limits for computation of floating-point numbers.

The following lists the macros defined in **float.h**. With the exception of **FLT_ROUNDS**, each macro is an expression; each value given is the minimum maximum that each expression must yield. The prefixes **DBL**, **FLT**, and **LDBL** refer, respective, to **double**, **float**, and **long double**.

DBL_DIG
Number of decimal digits of precision. Must yield at least ten.

DBL_EPSILON
Smallest possible floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0. Must be at most 1E-9.

DBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

DBL_MAX
Largest number that can be held by type **double**. Must yield at least 1E+37.

DBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **DBL_MAX**.

DBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **DBL_MAX**.

DBL_MIN
Smallest number that can be held by type **double**.

DBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **DBL_MIN**.

DBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **DBL_MAX**.

FLT_DIG
Number of decimal digits of precision. Must yield at least six.

FLT_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0. Must be at most 1E-5.

FLT_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

FLT_MAX
Largest number that can be held by type **float**. Must yield at least 1E+37.

FLT_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **FLT_MAX**.

FLT_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **FLT_MAX**.

FLT_MIN
Smallest number that can be held by type **float**.

FLT_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **FLT_MIN**.

FLT_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **FLT_MIN**.

FLT_RADIX
Base in which the exponents of all floating-point numbers are represented.

FLT_ROUNDS
Manner of rounding used by the implementation, as follows:

-1	Indeterminable, i.e., no strict rules apply
0	Toward zero, i.e., truncation
1	To nearest, i.e., rounds to nearest representable value
2	Toward positive infinity, i.e., always rounds up
3	Toward negative infinity, i.e., always rounds down

Any other value indicates that the manner of rounding is defined by the implementation.

LDBL_DIG
Number of decimal digits of precision. Must yield at least ten.

LDBL_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0. Must be at most 1E-9.

LDBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

LDBL_MAX
Largest number that can be held by type **long double**. Must yield at least 1E+37.

LDBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **LDBL_MAX**.

LDBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **LDBL_MAX**.

LDBL_MIN
Smallest number that can be held by type **long double**. Must be no greater than 1E-37.

LDBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **LDBL_MIN**.

LDBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **LDBL_MIN**.

Cross-references

Standard, §2.2.4.2
The C Programming Language, ed. 2, p. 257

See Also

Environment, header, numerical limits

floating constant — Definition

A *floating constant* is a constant that represents a floating-point number. A floating constant has three parts: the *value*, an *exponent*, and a *suffix*. Both the exponent and the suffix are optional.

The value section gives the value of the floating-point number. It also has three parts: a sequence of decimal digits, a period, and another set of digits. The first set of digits gives the whole-number part of the number, the period indicates the end of the whole-number part and the beginning of the fractional part, and the second sequence of digits encodes the fractional part. The period (which is sometimes called the "radix point") is always the character that marks the end of the whole-number sequence, regardless of the character recognized by the program's locale. In other words, the format of the C language floating constant is not locale-sensitive.

The exponent is used when the floating constant uses exponential notation. Here, the exponent gives the power of ten by which the base value is multiplied. For example,

```
1.05e10
```

represents the number

```
1.05*10^10
```

or

```
10,500,000,000
```

stored as a **double**. The exponent is introduced by the characters **e** or **E** followed by either **+** or **-**, which indicates the sign of the exponent. There follows the exponent itself, which consists of a sequence of decimal digits.

Finally, a floating constant may be followed by the suffixes **f**, **F**, **l**, or **L**. The first two indicate that the constant is of type **float**; the latter two, that the constant is of type **long double**. If a floating constant has no suffix, the translator assumes that it is of type **double**.

Cross-references

Standard, §3.1.3.1
The C Programming Language, ed. 2, p. 194

See Also

constants, float

floor() — Mathematics (libm)

Numeric floor
#include <math.h>
double floor(double *z***);**

floor returns the "floor" of a number, or the largest integer not greater than *z*. For example, the floor of 23.2 is 23, and the floor of -23.2 is -24.

floor returns the value expressed as a **double**.

Cross-references

Standard, §4.5.6.3
The C Programming Language, ed. 2, p. 251

See Also

ceil, fabs, fmod, mathematics

fmod — Mathematics (libm)

Calculate modulus for floating-point number
#include <math.h>
double fmod(double *number***, double** *divisor***);**

fmod divides *number* by *divisor* and returns the remainder. If *divisor* is nonzero, the return value will have the same sign as *divisor*. If divisor is zero, however, it will either return zero or set a domain error.

Cross-references

Standard, §4.5.6.4
The C Programming Language, ed. 2, p. 251

See Also

ceil, fabs, floor, mathematics

fopen() — STDIO (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (const char ****file*, const char ****mode***);**

fopen opens the stream *file*, and allocates and initializes the data stream associated with it. This makes the file available for STDIO operations. *file* may name either a file on a mass-storage device or a peripheral device. *file* can be no more than **FILENAME_MAX** characters long.

mode points to a string that consists of one or more of the characters "rwab+"; this indicates the mode into which the file is to be opened. The following set of mode strings are recognized:

a	Append, text mode
ab	Append, binary mode
a+	Append, text mode
ab+	Append, binary mode
a+b	Append, binary mode

r	Read, text mode
rb	Read, binary mode
r+	Update, text mode
rb+	Update, binary mode
r+b	Update, binary mode

w	Write, text mode
wb	Write, binary mode
w+	Update, text mode
wb+	Update, binary mode
w+b	Update, binary mode

Note the following:

- Opening *file* into any of the ‘a’ (append) modes means that data can be written only onto the end of the file. These modes set the file-position indicator to point to the end of the file. All other modes set it to point to the beginning of the file.
- To open *file* into any of the ‘r’ (read) modes, it must already exist and contain data. If *file* does not exist or cannot be opened, then **fopen** returns NULL.
- When a file is opened into any of the ‘w’ (write) modes, it is truncated to zero bytes if it already exists, or created if it does not.
- Opening *file* into any of the ‘+’ (update) modes allows you to write data into it or read data from it. When used with ‘r’ or ‘w’, data may be read from *file* or written into it at any point. When used with ‘a’, data may be written into it only at its end. To switch from reading a file to writing into it, either the stream’s input buffer must be flushed with **fflush** or the file-position indicator repositioned with **fseek**, **fsetpos**, or **rewind**.

fopen returns a pointer to the **FILE** object that controls the stream. It returns NULL if the file cannot be opened, for whatever reason.

fopen can open up to **FOPEN_MAX** files at once. This value is 20, including **stdin**, **stdout**, and **stderr**.

Example

This example opens a test file and reports what happens.

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

main(int argc, char *argv[])
{
      FILE *fp;

      if(argc != 3) {
            fprintf(stderr, "usage: fopen filename mode\n");
            exit(EXIT_FAILURE);
      }

      if((fp = fopen(argv[1], argv[2])) == NULL) {
            perror("Fopen failure");
            exit(EXIT_FAILURE);
      }

      fclose(fp);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.3
The C Programming Language, ed. 2, p. 160

See Also

fclose, fflush, freopen, setbuf, setvbuf, STDIO

Notes

To update an existing file, use the mode **r+**

fopen associates a fully buffered stream with *file* only if *file* does not access an interactive device.

A conforming implementation must support all of the modes described above. It may also offer other modes in which to open a file.

for — C keyword

Loop construct
for(*initialization***;** *condition***;** *modification***)** *statement*

for introduces a conditional loop. It takes three expressions as arguments; these are separated by semicolons ';'. *initialization* is executed before the loop begins. *condition* describes the condition that must be true for the loop to execute. *modification* is the statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable **i** to zero; then declares that the loop will continue as long as **i** remains less than ten; and finally, increments **i** by one after every iteration of the loop. This ensures that the loop will iterate exactly ten times (from **i==0** through **i==9**). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement.

The **for** statement is equivalent to:

```
initialization;
while(condition) {
      statement
      modification;
}
```

Example

For an example of this statement, see **putc**.

Cross-references

Standard, §3.6.5.3
The C Programming Language, ed. 2, pp. 60*ff*

See Also

break, C keywords, continue, do, statements, while

fpos_t — Type

Encode current position in a file

The type **fpos_t** is defined in the header **stdio.h**. It is used by the functions **fgetpos** and **fsetpos** to encode the current position within a file (the *file-position indicator*). Its type may vary from implementation to implementation.

fpos_t and its functions are designed to manipulate files whose file-position indicator cannot be encoded within a **long**.

Cross-references

Standard, §4.9.1, §4.9.9.1, §4.9.9.3
The C Programming Language, ed. 2, p. 248

See Also

fgetpos, file, FILE, file-position indicator, fsetpos, STDIO, stdio.h

Notes

The Standard leaves the actual type of **fpos_t** to the implementation. The intent is to define a data type that can be obtained by a call to **fgetpos** and used on later calls to **fsetpos**. It is not wise to try to manipulate this type directly or to dissect it. Code that depends on specific properties of **fpos_t** may not be portable.

fprintf() — STDIO (libc)

Print formatted text into a stream
#include <stdio.h>
int fprintf(FILE ***fp*, const char ***format***, ...);**

fprintf constructs a formatted string and writes it into the stream pointed to by *fp*. It can translate integers, floating-point numbers, and strings in a variety of text formats.

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

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*, and the argument should be of the type appropriate to the conversion specification. 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 **char** *.

If there are fewer arguments than conversion specifications, then **fprintf**'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 **fprintf** is undefined. Thus, presenting an **int** where **fprintf** expects a **char** * may generate unwelcome results.

If it could write the formatted string, **fprintf** returns the number of characters written; otherwise, it returns a negative number.

Example

This example prints two messages: one into **stderr** and the other into **stdout**.

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

main(void)
{
      fprintf(stderr, "A message to stderr.\n");
      printf("A message to stdout.\n");
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.6.1
The C Programming Language, ed. 2, p. 243

See Also

printf, sprintf, STDIO, vfprintf, vprintf, vsprintf

Notes

fprintf can construct and output a string of up to at least 509 characters.

The character that **fprintf** uses 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.

fputc() — STDIO (libc)

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

fputc converts *character* to an **unsigned char**, writes it into the stream pointed to by *fp*, and advances the file-position indicator for *fp*.

fputc returns *character* if it was written successfully; otherwise, it sets the error indicator for *fp* and returns **EOF**.

Example

The following example uses **fputc** to copy the contents of one file into another.

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

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

      if(errno)
            perror(NULL);
      if(format != NULL) {
            va_start(argptr, format);
            vfprintf(stderr, format, argptr);
            va_end(argptr);
      }
      exit(EXIT_FAILURE);
}

main(int argc, char *argv[])
{
      FILE *ifp, *ofp;
      int ch;

      if(argc != 3)
            fatal("usage: fputc oldfile newfile\n");

      if((ifp = fopen(argv[1], "r")) == NULL)
            fatal("Cannot open %s\n", argv[1]);
      if((ofp = fopen(argv[2], "w")) == NULL)
            fatal("Cannot open %s\n", argv[2]);

      while ((ch = fgetc(ifp)) != EOF)
            if (fputc(ch, ofp) == EOF)
                  fatal("Write error for %s\n", argv[2]);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.7.3
The C Programming Language, ed. 2, p. 247

See Also

fputs, fputw, putc, putchar, puts, putw, STDIO

fputs() — STDIO (libc)

Write a string into a stream
#include <stdio.h>
int fputs(char ***string*; FILE ***fp***);**

fputs writes the string pointed to by *string* into the stream pointed to by *fp*. The terminating null character is not written. Unlike the related function **puts**, it does not append a newline character to the end of *string*.

fputs returns a non-negative number if it could write *string* correctly. If it could not, it returns **EOF**.

Cross-references

Standard, §4.9.7.4
The C Programming Language, ed. 2, p. 247

See Also

fputc, putc, putw, putchar, puts, putw, STDIO

fputw() — Extended function (libc)

Write an integer to a stream
#include <xstdio.h>
short fputw(short *word***, FILE ***fp***);**

fputw writes *word* into the file stream *fp*, and returns the value written.

fputw returns EOF when an error occurs. A call to **ferror** or **feof** may be needed to distinguish this value from a valid end-of-file signal.

Example

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

See Also

extended STDIO, fgetw

Notes

To conform to the ANSI Standard, this function has been moved from the header **stdio.h** to the header **xstdio.h**. This may require that some code be altered.

fputw 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.

fread() — STDIO (libc)

Read data from a stream
#include <stdio.h>
size_t fread(void ***buffer*, size_t** *size***, size_t** *n***, FILE ***fp***);**

fread reads up to *n* items, each being *size* bytes long, from the stream pointed to by *fp* and copies them into the area pointed to by *buffer*. It advances the file-position indicator by the amount appropriate to the number of bytes read.

fread returns the number of items read. If the value returned by **fread** is not equal to *n*, use the functions **ferror** and **feof** to find, respectively, if an error has occurred or if the end of file has been encountered.

Example

This example reads data structures into an array of structures. It is more to be read than used.

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#define COUNT 10

struct aStruct {
      double d;
      float  f;
      int    i;
} arrayStruct[COUNT];

main(void)
{
      int i;
      FILE *ifp;

      if((ifp = fopen("a.s", "rb")) == NULL) {
            perror("Cannot open a.s");
            exit(EXIT_FAILURE);
      }

      /*       buffer      blocksize            count FILE */
      i=fread(arrayStruct,sizeof(struct aStruct),COUNT,ifp);
      if(i != COUNT) {
            fprintf(stderr, "Only read %d blocks\n", i);
            return(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.8.1
The C Programming Language, ed. 2, p. 247

See Also

fwrite, STDIO

Notes

If an error occurs while data are being read, then the value of the file-position indicator is indeterminate. If either *size* or *n* is zero, then **fread** returns zero and reads nothing.

free() — General utility (libc)

Deallocate dynamic memory
#include <stdlib.h>
void free(void ****ptr*);**

free deallocates a block of dynamic memory that had been allocated by **malloc**, **calloc**, or **realloc**. Deallocating memory may make it available for reuse.

ptr points to the block of memory to be freed. It must have been returned by **malloc**, **calloc**, or **realloc**. **free** marks the block indicated by *ptr* as unused, so the **malloc** search can coalesce it with contiguous free blocks.

free returns nothing. It prints a message and calls **abort** if it discovered that the arena has been corrupted. This most often occurs by storing data beyond he bounds of an allocated block.

Cross-references

Standard, §4.10.3.2
The C Programming Language, ed. 2, p. 167

See Also

calloc, malloc, general utilities, realloc

Notes

If *ptr* does not point to a block of memory that had been allocated by **calloc**, **malloc**, or **realloc**, the behavior of **free** is undefined.

If *ptr* is equivalent to NULL, then no action occurs.

Finally, if a program attempts to access memory that has been freed, its behavior is undefined.

freopen() — STDIO (libc)

Re-open a stream
#include <stdio.h>
FILE *freopen(const char ****file*, const char ****mode***, FILE ****fp***);**

freopen opens *file* and associates it with the stream pointed to by *fp*, which is already in use. It first tries to close the file currently associated with *fp*. Then it opens *file*, and returns a pointer to the **FILE** object, through which other STDIO routines can access *file*. Under some execution environments, **freopen** can be used to access a peripheral device as well as a file. Thus, **freopen** is often used to change the device associated with the streams **stdin**, **stdout**, or **stderr**, as well as to the change the access modes for an open file.

mode indicates the manner in which *file* is to be accessed. For a table of the modes described by the Standard, see **fopen**.

freopen returns NULL if *file* could not be opened properly; otherwise, it returns *fp*.

Example

This example uses **freopen** to copy a list of files into one file.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.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(int argc, char *argv[])
{
      FILE *ifp, *ofp;
      int i, c;

      if(argc < 3)
            fatal("usage: freopen input1 input2 ... output\n");
      if((ofp = fopen(argv[argc - 1], "wb")) == NULL)
            fatal("Cannot open %s\n", argv[argc - 1]);

      ifp = stdin;
      for(i = 1; i < argc; i++) {
            if((ifp = freopen(argv[i], "rb", ifp)) == NULL)
                  fatal("Cannot open %s\n", argv[i]);

            while((c = fgetc(ifp)) != EOF)
                  fputc(c, ofp);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.5.4
The C Programming Language, ed. 2, p. 162

See Also

fclose, fflush, fopen, setbuf, setvbuf, STDIO

Notes

freopen will attempt to close the file currently associated with *fp*. However, if it cannot be closed, **freopen** will still open *file* and associate *fp* with it.

frexp() — Mathematics (libm)

Fracture floating-point number
#include <math.h>
double frexp(double *real*, **int** **exp*);

frexp breaks a double-precision floating-point number into its mantissa and exponent. It returns the mantissa *m* of the argument *real*, such that 0.5 <= *m* < 1 or *m*=0, and stores the binary exponent in the area pointed to by *exp*. The exponent is an integral power of two.

See **float.h** for more information about the structure of a floating-point number.

Cross-references

Standard, §4.5.4.3
The C Programming Language, ed. 2, p. 251

See Also

atof, ceil, fabs, floor, ldexp, mathematics, modf

fscanf() — STDIO (libc)

Read and interpret text from a stream
#include <stdio.h>
int fscanf(FILE **fp*, **const char** **format*, **...);**

fscanf reads characters from the stream pointed to by *fp*, and uses the string pointed to by *format* to interpret what it has read into the appropriate type of data. *format* points to a string that contains one or more conversion specifications, each of which is introduced with the percent sign '%'. For a table of the conversion specifiers that may be used with **fscanf**, see **scanf**.

After *format* can come one or more arguments. There should be one argument for each conversion specification in *format*, and the argument should point to a data element of the type appropriate to the conversion specification. For example, if *format* contains conversion specifications for an **int**, a **long**, and a string, then *format* should be followed by three arguments: respectively, a pointer to an **int**, a pointer to a **long**, and an array of **char**s.

If there are fewer arguments than conversion specifications, then **fscanf**'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 **fscanf** returns.

fscanf returns the number of input elements it scanned and formatted. If an error occurs while **fscanf** is reading its input, it returns **EOF**.

Example

This example reads and displays data from a file of strings with the following format:

```
      ABORT    C        312   1-24-88  11:03a
      ABS      C        239   1-24-88  11:03a
```

This is the output of the MS-DOS command **dir**.

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

main(int argc, char *argv[])
{
      int count;
      long size;
      char fname[8], ext[3];
      FILE *ifp;

      if(argc != 2) {
            printf("usage: fscanf inputfile\n");
            exit(EXIT_FAILURE);
      }

      if((ifp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      while((count = fscanf(ifp, "%8s %3s %ld %*[^\n]",
            fname, ext, &size)) != EOF)
            if(count == 3)
                  printf("%s.%s %ld\n", fname, ext, size);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.6.2
The C Programming Language, ed. 2, p. 245

See Also

scanf, sscanf, STDIO

Notes

fscanf is best used to read data you are certain are in the correct format, such as strings previously written out with **fprintf**.

The character that **fscanf** recognizes as representing the decimal point is affected by the program's locale, as set by the function **setlocale**. For more information, see **localization**.

fseek() — STDIO (libc)

Set file-position indicator
#include <stdio.h>
int fseek(FILE ***fp*, **long int *offset*, **int** *whence***);**

fseek sets the file-position indicator for stream *fp*. This changes the point where the next read or write operation will occur.

offset and *whence* specify how the value of the file-position indicator should be re-set. *offset* is the amount to move it, in bytes; this is a signed quantity. *whence* is the point from which to move it, as follows:

SEEK_CUR	From the current position
SEEK_END	From the end of the file
SEEK_SET	From the beginning of the file

The values of these macros are set in the header **stdio.h**.

fseek clears the end-of-file indicator and undoes the effects of a previous call to **ungetc**; the next operation on *fp* may be input or output.

fseek returns a number other than zero for what the Standard calls an "improper request." Presumably, this means attempting to seek past the end or the beginning of a file, attempting to seek on an interactive device (such as a terminal), or attempting to seek on a file that does not exist.

Example

This example implements the UNIX game **fortune**. It randomly selects a line from a text file, and prints it. Multi-line fortunes, such as poems, should have '@'s embedded within them to mark line breaks.

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

main(int argc, char *argv[])
{
      FILE  *ifp;
      double  randomAdj;
      int c;

      if(argc != 2) {
            printf("usage: fseek inputfile\n");
            exit(EXIT_FAILURE);
      }

      if ((ifp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      fseek(ifp, 0L, SEEK_END);
      randomAdj = (double)ftell(ifp)/((double)RAND_MAX);
      /* Exercise rand() to make number more random */
      srand((unsigned int)time(NULL));
      for(c = 0; c < 100; c++)
            rand();

      fseek(ifp, (long)(randomAdj * (double)rand()), SEEK_SET);
      while('\n' != (c = fgetc(ifp)) && EOF != c)
            ;

      if(c == EOF) {
            printf("File does not end with newline\n");
            exit(EXIT_FAILURE);
      }

      while('\n' != (c = fgetc(ifp))) {
            if(EOF == c) {
                  fseek(ifp, 0L, SEEK_SET);
                  continue;
            }

            /* display multi-line fortunes */
            if('@' == c)
                  c = '\n';
            putchar(c);
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.9.9.2
The C Programming Language, ed. 2, p. 248

See Also

fsetpos, ftell, STDIO

Notes

Although the Standard does not describe the behavior of **fseek** if you attempt to seek beyond the end of a file, it does not result in an error condition until the corresponding read or write is attempted.

Note, too, that **fseek** allows a user to seek past the beginning of a binary file as well as past its end. *Caveat utilitor.*

fsetpos() — STDIO (libc)

Set file-position indicator
#include <stdio.h>
int fsetpos(FILE ****fp*, const fpos_t ****position***);**

fsetpos resets the file-position indicator. *fp* points to the file stream whose indicator is being reset. *position* is a value that had been returned by an earlier call to **fgetpos**; it is of type **fpos_t**, which is defined in the header **stdio.h**.

Like the related function **fseek**, **fsetpos** clears the end-of-file indicator and undoes the effects of a previous call to **ungetc**. The next operation on *fp* may read or write data.

fsetpos returns zero if all goes well. If an error occurs, it returns nonzero and sets the integer expression **errno** to the appropriate error number.

Example

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

Cross-references

Standard, §4.9.9.3
The C Programming Language, ed. 2, p. 248

See Also

fgetpos, fseek, ftell, rewind, STDIO

Notes

The Standard designed **fsetpos** to be used with files whose file position cannot be represented within a **long**. Under **Let's C**, it behaves the same as **fseek**.

Note, too, that there is no given way to obtain the value of the file-position indicator other than by a previous call to **fgetpos**.

ftell() — STDIO (libc)

Get value of file-position indicator
#include <stdio.h>
long int ftell(FILE ****fp*);**

ftell returns the value of the file-position indicator for the stream pointed to by *fp*.

The information returned by **ftell** varies, depending upon the run-time environment and whether the stream pointed to by *fp* was opened into text mode or binary mode. If *fp* was opened into binary mode, then **ftell** returns the number of characters from the beginning of the file to the current position. If *fp* was opened into text mode, however, **ftell** returns an implementation-defined number.

For example, in UNIX-style environments, **ftell** returns the number of characters the current position is from the beginning; whereas under MS-DOS, where lines are terminated by a carriage return-newline pair, **ftell** counts each carriage return and each newline as a character in its return value.

If an error occurs, **ftell** returns -1L and sets the integer expression **errno** to the appropriate value. An error will occur if, for example, you attempt to use **ftell** with a stream that is associated with a device that is not file-structured.

Example

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

Cross-references

Standard, §4.9.9.4
The C Programming Language, ed. 2, p. 248

See Also

errno, fgetpos, fseek, fsetpos, rewind, STDIO

function call library archive

function — Definition

A *function* is a construct that performs a task. It includes statements and related variables, including those passed to it as arguments. A C program commonly consists of many functions, each of which performs one or more tasks.

A function can be compiled and stored in a *library* or *archive*, from which it can be extracted by a linker.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, pp. 67*ff*

See Also

Definitions

function call — Definition

A *function call* invokes a function at a particular point in a program. A function call consists of an identifier followed by a pair of parentheses '()'; between the parentheses may appear a list of arguments.

The behavior of a function call is affected by the following: a *function declaration*, a *function prototype*, and a *function definition*. Some or all of these may be visible to the translator when it interprets the function call. The translator must respond appropriately to the presence or absence of each when it translates the function call. The following paragraphs describe how these elements affect the behavior of a function call.

Function Declaration

If a function declaration is visible when function is called, then the function is assumed to return the type and to have the linkage noted in the declaration.

For example, the following declaration

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

declares that the function **example** has static linkage and returns a pointer to **char**.

If no function declaration is visible to the translator when it reads the function call, then it assumes that the function has the declaration:

```
extern int example();
```

where **example** is the name of the function being called. This action is sometimes referred to as an *implicit declaration* of a function. It declares the function to have external linkage and return type **int**.

If a declaration, whether explicit or implict, does not match what the function actually returns, the behavior is undefined.

Consider a function call of the form:

```
char *value;
value = example(argument1, argument2);
```

If the translator sees the declaration for **example**, then it knows that **example** returns a pointer to **char** and reacts accordingly. If, however, it does not see the declaration for **example**, then it implicitly declares **example** to return an **int**, and generates code appropriate for that. What happens after this error occurs may vary from implementation to implementation.

A function declaration does not check the number or the type of arguments of the function call; to check arguments, you should use a function prototype (described below). If the number and the types of the arguments to a function call do not match those that the function requires, and if no prototype is visible when the function is called, then behavior is undefined.

Function Prototype

A function prototype is a more detailed form of function declaration. A function prototype lists not only the linkage and the return value of a function, but also its parameters and the type of each.

This allows the translator to check each function call to ensure that it has the correct number of arguments and that each argument has the correct type. See **function prototype** for a full description.

Function Definition

A *function definition* defines code for a function. In effect, the function definition is where the function “lives”.

A function definition begins with a *declarator*, which includes a list of the parameters the function needs. Behavior is undefined if a function call’s list of arguments does not match the function declaration’s list of parameters, both in number and in type, and no prototype is visible. A function call in the presence of a prototype-style function definition will be prototype-checked against this declaration.

Let’s C Calling Conventions

The following presents the calling conventions for **Let’s C**.

The design of the calling conventions had to take into account the fact that C does not require that the number of arguments passed to a function be the same as the number of arguments specified in the function’s declaration. Routines with a variable number of arguments are not uncommon; for example, **printf** and **scanf** can take a variable number of arguments. Another consideration was the availability of **register** variables.

Therefore, **Let’s C** uses the following calling sequence. The function arguments are pushed onto the stack from the first, or rightmost, through the last, or leftmost. **long**s are pushed high-half first. This makes the word order compatible with the **dd** instruction. **double**s are pushed so that the byte order on the stack is compatible with the i8087 co-processor. The function is then called with a NEAR call (either directly or indirectly) for SMALL model, or a FAR call for LARGE model. An **add** instruction after the call removes the arguments from the stack.

For example, the function call

```
int a;
long b;
char c;

foo(void)
{
      example(a, b, c);
}
```

generates the code

```
movb    al,c
cbw
push    ax
push    b+2
push    b
push    a
call    example_
add     sp,8
```

An underbar character ‘_’ has been appended to the function name. This serves two purposes. First, it makes it harder to accidentally call routines written in other languages. Second, it means that two routines with the same name can be called from C and another language in identical fashions.

The parameters and local variables in the called function are referenced as offsets from the BP register. In SMALL model, the arguments begin at offset 8 and continue toward higher addresses, whereas the local variables begin at offset -2 and continue toward lower addresses.

The SP register points the local variable with the lowest address. Thus, when **example_** is reached in the above model, the SMALL-model stack frame resembles the following:

```
       High          +-----------------------+
                     | c (widened to a word) |
                     +-----------------------+
                     |    high half of b     |
                     +-----------------------+
                     |    low half of b      |
                     +-----------------------+
                     |          a            |
       Low           +-----------------------+
```

In LARGE model, the return address occupy two words.

Functions return **int**s in the AX register, **long**s in the DX:AX register pair, pointers in the AX register for SMALL model and in DX:AX for LARGE model, and **double**s on the top of the i8087's stack. The following program

```
      example(int a, b, c)
      {
          return (a * b - c);
      }
```

when compiled with the **-VASM** option, produces the following assembly language program:

```
       .shri
       .globl example_

example_:
       push          si
       push          di
       push          bp
       mov           bp, sp
       mov           ax, 10(bp)
       imul          8(bp)
       sub           ax, 12(bp)
       pop           bp
       pop           di
       pop           si
       ret
```

In SMALL model, the runtime startup initializes the registers CS, DS, ES, and SS, and the segment registers remain unchanged. In LARGE model, the runtime startup initializes registers SS and SP. The generated code loads the other segment registers as needed. As noted above, a C function preserves registers SI, DI, BP, and SP, plus the segment registers in SMALL model; other registers may be overwritten.

Source code for some runtime startup routines is included with the sample programs that come with your copy of **Let's C**.

Let's C pushes function arguments as follows.

char	Widened to **int**, then pushed
double	Pushed in i8087 order
float	Widened to **double**, then pushed
int	Pushed in machine word order
long double	Same as **double**
double	Pushed in i8087 order
struct	Pushed in memory order
union	Pushed in memory order
pointer	SMALL: offset pushed LARGE: base pushed, then offset pushed

Functions return values as follows:

char	In AL
double	On i8087 stack
float	Same as **double**
int	In AX
long	In DX:AX
long double	Same as **double**
struct	SMALL: pointer in AX LARGE: pointer in DX:AX
union	SMALL: pointer in AX LARGE: pointer in DX:AX
pointer	SMALL: in AX LARGE: in DX:AX

A function that returns a **struct** or **union** actually returns a pointer. The code generated for the function call block-moves the result to its destination. Functions that return a **float** or **double** return it on the i8087 stack if your computer has an i8087 co-processor; otherwise, they return it in the global double **fpac_**.

For example, consider the call

```
example(int i, long l, char c, char *cp);
```

where **example** declares two automatic **int**s. After execution of the call and the prologue of **example**, the SMALL-model stack contains the following 11 words:

```
       High          +------------------+
                     |        cp        |
                     +------------------+
                     |        c         |
                     +------------------+
                     |  high word of l  |
                     +------------------+
                     |  low word of l   |
                     +------------------+
                     |        i         |
                     +------------------+
                     |  return address  |
                     +------------------+
                     |     saved SI     |
                     +------------------+
                     |     saved DI     |
                     +------------------+
                     |     saved BP     |
                     +------------------+
                     | space for auto 1 |
                     +------------------+
                     | space for auto 2 |
       Low           +------------------+
```

The following example performs a simple function call:

```
main(void)
{
      example(1, 2);    /* call sample routine */
}

example(int p1, int p2)
{
      int a, b;

      a = 3;
      b = 4;
}
```

When the function **example** is about to return, the stack appears as follows:

```
                                                               SMALL          LARGE
High +------------------+
     |        2         | parm 2                               10(bp)         12(bp)
     +------------------+
     |        1         | parm 1                               8(bp)          10(bp)
     +------------------+
     | Return Address:  |
     |   2 words in     |
     |  LARGE model,    |
     | 1 in SMALL model | ret.addr.                            6(bp)          6(bp)
     | 1 in SMALL model |
     +------------------+
     |    main's SI     |                                      4(bp)          4(bp)
     +------------------+
     |    main's DI     |                                      2(bp)          2(bp)
     +------------------+
     |    main's BP     |                                      (bp)           (bp)
     +------------------+
     |        3         | a                                    -2(bp)         -2(bp)
     +------------------+
     |        4         | SP b                                 -4(bp)         -4(bp)
Low  +------------------+
```

Cross-references

Standard, §3.3.2.2
The C Programming Language, ed. 2, p. 201

See Also

(), function declarators, function definition, function prototype, operators

Notes

C passes arguments by value; this is known as *call-by-value semantics*. This means that C always passes a *copy* of an argument to the called function. If the called function alters the value of its copy, the original argument will not change. The only way the called function can change the value of the original argument is if it is passed the address of that argument.

C does not specify the order of evaluation of arguments. Hence, for maximally portable code, you should not rely on any specific order of evaluation.

The Rationale notes that the original syntax for calling a function through a pointer to a function

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

has been augmented to allow the pointer to be automatically deferenced as:

```
      example();
```

This means that pointers to functions stored in structures may be called with the syntax

```
      example.funcmember();
```

instead of the more cluttered:

```
      (*structure.funcmember)();
```

Such an expression cannot be used as an lvalue.

The order of evaluation of a function's arguments is undefined.

function declarators — Definition

A *function declarator* declares a function.

A function declarator is marked by the use of parentheses '()' after the identifier. Function declarators come in two varieties.

In the first form, the parentheses enclose a list of parameters and their types. The list may end with an ellipsis '...'. This indicates that the function takes an indefinite number of arguments. The list may also consist merely of **void**, which indicates that the function takes no arguments.

This form of function declaration is called a *parameter type list*. It is also called a *function prototype*, because a succeeding call to the function can be checked against it to ensure that the call uses the correct number of arguments and that the type of each is correct. It is also referred to as a *new-style function declarator*. See **function prototype** for more information.

The second form of function declarator names the arguments to a function, but does not give their types. No prototype checking can be performed against a declarator of this sort. This form is called a *function identifier list*. It is also called an *old-style function declarator*, because the Standard states that this form is obsolescent.

Either style of function declaration will be checked against any prototype that had been declared previously and that is within scope.

Finally, a function declarator may consist simply of two parentheses with nothing between them. This indicates that the identifier names a function, but says nothing about the number or the type of arguments that the function takes.

Cross-references

Standard, §3.5.4.3
The C Programming Language, ed. 2, p. 218

See Also

(), declarators, function definition, function prototype

function definition — Definition

A *definition* is a declaration that reserves storage for the thing declared.

A program or its associated libraries must define exactly once each function it uses. A *compound-statement* is the code that forms the body of the function.

The *declaration-specifiers* give the function's storage class and return type. The storage class may be either **extern** or **static**. If no storage class is specified, then the function is **extern** by default. The return type may be any type except an array. This means that a function may return a structure, which was illegal under Kernighan and Ritchie's definition of C. If no return type is specified, the function is assumed to return type **int**.

The *declarator* names the function and its formal parameters. A function's parameters can be described in either of two ways. The first is to use *declaration-specifiers*. These name the function's parameters and give the type of each. For example, the function **fopen** has the following declaration:

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

Here, **const char *file** and **const char *mode** name **fopen**'s parameters and give the type of each.

Each declaration specifier must have both a type and an identifier. The only exception is when a function takes no parameters; then the type **void** may be used without an identifier. A declarator of this form serves as a function prototype for all subsequent calls to this function.

The second way to declare a function's parameters is to use a *declaration-list*. Here, the declarator contains only the parameter's name. Each formal parameter is then declared in a list that follows the declarator. For example, if **fopen** used a declaration list, it would appear as follows:

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

In this example, the declaration list gives the types of the identifiers **file** and **mode**. If an identifier appears in the declarator but is not named in the following identifier list, it is assumed to be of type **int**. A declaration list can contain no storage-class specifier except **register**, and no identifier may be initialized in the identifier list.

A declarator of this type cannot be used as a function prototype for subsequent calls. The Standard considers this type of function definition to be obsolete and expects that it will disappear over time.

With either manner of definition, all parameters have automatic storage (as indicated by the fact that the only storage-class specifier allowed is **register**). When an argument is read, it is converted to an object of the type of the corresponding parameter.

Finally, every parameter is considered to be an lvalue.

Cross-references

Standard, §3.7.1
The C Programming Language, ed. 2, p. 225

See Also

conversions, definition, external definitions, function calls, function declarators, function prototypes, object definition, prototype

Notes

If a function takes an indefinite number of parameters, and its function definition does not use a list of declaration specifiers that ends with the ellipsis operator '...', the behavior is undefined.

function designator — Definition

A *function designator* is any expression that has a function type.

A function designator whose type is "function that returns *type*" is normally converted to the type "pointer to function that returns *type*." One exception is when the function designator is the operand to the unary **&** operator. In this case, the use of **&** states explicitly that the address of the function designator is to be taken, so implicit conversion is not necessary.

Cross-references

Standard, §3.2.2.1
The C Programming Language, ed. 2, p. 201

See Also

conversions, implicit conversion

function prototype — Definition

A *function prototype* is a sophisticated form of function declaration. A function prototype lists not only the linkage and the return value of a function, but also lists its arguments and the types of each. This allows the translator to check each argument in a function call to see that it is of the correct type.

Function prototypes are normally kept in a header. The header must be explicitly included in the source module for the prototype to be visible to the translator as it translates the module. For example, consider the following function prototype:

```
extern char *example(int argument1, long argument2);
```

This declares that the function **example** has external linkage; that it returns a pointer to **char**; and that it takes two arguments, the first of which is an **int** and the second of which is a **long**. The names of the arguments given in the function prototype are used only in the prototype. They are not visible outside of it, and so will not affect any other use of those names in your program.

A function prototype may end with an ellipsis '...'. This indicates that the function takes a variable number of arguments. For example, consider the following prototype for the function **fprintf**:

```
int fprintf(FILE *fp, const char *format, ...);
```

The prototype declares that **fprintf** takes at least two arguments, one of which is a pointer to an object of type **FILE** and the other is a pointer to **char**. The ellipsis at the end of the list of arguments indicates that a variable number of arguments may follow.

When the translator reads a call to **fprintf**, it compares the first two arguments against their declared types. All further arguments in the function call are not checked. Every function that takes a variable number of arguments must have a function prototype; otherwise, its behavior is undefined.

Another advantage of function prototypes is that arguments do not undergo the *default argument promotions*. Normally, the translator promotes arguments as follows: **char** and **short int** are promoted to **int** (if it can hold the value encoded within the variable), or to **unsigned int** (if **int** cannot hold the value). **float** is always to **double**. This is discussed more fully below.

If a function takes no arguments, its prototype should be of the form:

```
extern char *example(void);
```

The type specifier **void** between the parentheses indicates that the function takes no arguments. This is *not* the same as:

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

This latter declaration says merely that you have nothing to say about the function's arguments.

When a function prototype is *not* visible where the function is called, then the following rules apply:

- The arguments of the function call undergo the default argument promotions. Behavior is undefined when the number of arguments does not match the number of parameters in the function definition, regardless of whether the prototype is visible where the function is defined.
- If the function prototype is *not* visible where the function is defined, then the parameters of the function definition also undergo default argument promotion. Behavior is undefined when the type of a promoted argument does not match that of its corresponding promoted parameter.
- If the function prototype *is* visible where the function is defined, then behavior is undefined either when the type of a promoted argument does not match that of its corresponding parameter, or when the function prototype ends with an ellipse '...'.

When, however, the function prototype is visible both where the function is defined and where it is called, each argument of the function call is implicitly converted to the type of its corresponding parameter. If the function prototype ends in an ellipsis, then such promotion of arguments ends with the last declared parameter; all arguments thereafter undergo default argument promotion.

For example, consider the following function call:

```
int fprintf(FILE *fp, const char *format, ...);
   . . .
```

```
float argument;
   . . .
fprintf(stderr, "%3.2f\n", argument);
```

The first two arguments in the function call are cast to the types given in the prototype. The third argument, which is indicated by the ellipsis in the function prototype, undergoes the usual promotion **double** before being passed to **fprintf**.

The last situation allows you to write code like:

```
#include <math.h>
   . . .
d = cos(2);
```

This works correctly, because the prototype

```
double cos(double d);
```

in the header tells the translator to promote the integer constant **2** to **double** rather than passing an **int** to the function, as it would do otherwise.

Cross-references

Standard, §3.1.2.1, §3.3.2.2, §3.5.4.3, §3.7.1
The C Programming Language, ed. 2, p. 202

See Also

function call, function declarators, function definition

fwrite() — STDIO (libc)

Write data into a stream
#include <stdio.h>
size_t fwrite(const void ****buffer*, size_t** *size***, size_t** *n***, FILE ****fp***);**

fwrite writes up to *n* items, each being *size* bytes long, from the area pointed to by *buffer* into the stream pointed to by *fp*. It increments the file-position indicator by the amount appropriate to the number of bytes written.

fwrite returns the number of items written. This will be equal to *n*, unless a write error occurs. If a write error occurs, the value of the file-position indicator is indeterminate.

Example

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

Cross-references

Standard, §4.9.8.2
The C Programming Language, ed. 2, p. 247

See Also

fread, STDIO