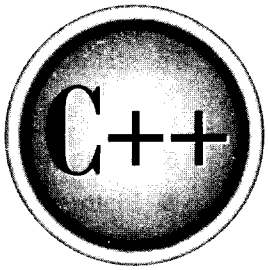
The Complete Reference



Part III

The Standard Function Library

C++ defines two types of libraries. The first is the standard function library. This library consists of general-purpose, stand-alone functions that are not part of any class. The function library is inherited from C. The second library is the object- oriented class library. Part Three of the book provides a reference to the standard function library. Part Four describes the class library.

The standard function library is divided into the following categories:

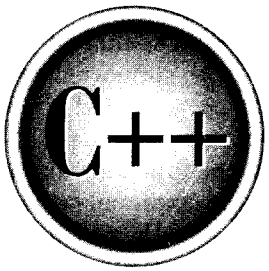
- I/O
- String and character handling
- Mathematical
- Time, date, and localization
- Dynamic allocation
- Miscellaneous
- Wide-character functions

The last category was added to Standard C in 1995 and was subsequently incorporated into C++. It provides wide-character (**wchar_t**) equivalents to several of the library functions. Frankly, the use of the wide-character library has been very limited, and C++ provides a better way of handling wide-character environments, but it is briefly described in Chapter 31 for completeness.

C99 added some new elements to the C function library. Several of these additions, such as support for complex arithmetic and type-generic macros for the mathematical functions, duplicate functionality already found in C++. Some provide new features that might be incorporated into C++ in the future. In all cases, the library elements added by C99 are incompatible with C++. Thus, the additions made to the Standard C library by C99 are not discussed in this book.

One last point: All compilers supply more functions than are defined by Standard C/C++. These additional functions typically provide for operating-system interfacing and other environment-dependent operations. You will want to check your compiler's documentation.

The Complete Reference



Chapter 25

The C-Based I/O Functions

This chapter describes the C-based I/O functions. These functions are also supported by Standard C++. While you will usually want to use C++'s object-oriented I/O system for new code, there is no fundamental reason that you cannot use the C I/O functions in a C++ program when you deem it appropriate. The functions in this chapter were first specified by the ANSI C standard, and they are commonly referred to collectively as the ANSI C I/O system.

The header associated with the C-based I/O functions is called **<cstdio>**. (A C program must use the header file **stdio.h**.) This header defines several macros and types used by the file system. The most important type is **FILE**, which is used to declare a file pointer. Two other types are **size_t** and **fpos_t**. The **size_t** type (usually some form of unsigned integer) defines an object that is capable of holding the size of the largest file allowed by the operating environment. The **fpos_t** type defines an object that can hold all information needed to uniquely specify every position within a file. The most commonly used macro defined by the headers is **EOF**, which is the value that indicates end-of-file.

Many of the I/O functions set the built-in global integer variable **errno** when an error occurs. Your program can check this variable when an error occurs to obtain more information about the error. The values that **errno** may take are implementation dependent.

For an overview of the C-based I/O system, see Chapters 8 and 9 in Part One.

Note *This chapter describes the character-based I/O functions. These are the functions that were originally defined for Standard C and C++ and are, by far, the most widely used. In 1995, several wide-character (**wchar_t**) functions were added, and they are briefly described in Chapter 31.*

clearerr

```
#include <cstdio>
void clearerr(FILE *stream);
```

The **clearerr( )** function resets (i.e., sets to zero) the error flag associated with the stream pointed to by *stream*. The end-of-file indicator is also reset.

The error flags for each stream are initially set to zero by a successful call to **fopen( )**.

File errors can occur for a wide variety of reasons, many of which are system dependent. The exact nature of the error can be determined by calling **perror( )**, which displays what error has occurred (see **perror( )**).

Related functions are **feof( )**, **ferror( )**, and **perror( )**.

fclose

```
#include <cstdio>
int fclose(FILE *stream);
```

The **fclose( )** function closes the file associated with *stream* and flushes its buffer. After an **fclose( )**, *stream* is no longer connected with the file, and any automatically allocated buffers are deallocated.

If **fclose( )** is successful, zero is returned; otherwise **EOF** is returned. Trying to close a file that has already been closed is an error. Removing the storage media before closing a file will also generate an error, as will lack of sufficient free disk space.

Related functions are **fopen( )**, **freopen( )**, and **fflush( )**.

feof

```
#include <cstdio>
int feof(FILE *stream);
```

The **feof( )** function checks the file position indicator to determine if the end of the file associated with *stream* has been reached. A nonzero value is returned if the file position indicator is at end-of-file; zero is returned otherwise.

Once the end of the file has been reached, subsequent read operations will return **EOF** until either **rewind( )** is called or the file position indicator is moved using **fseek( )**.

The **feof( )** function is particularly useful when working with binary files because the end-of-file marker is also a valid binary integer. Explicit calls must be made to **feof( )** rather than simply testing the return value of **getc( )**, for example, to determine when the end of a binary file has been reached.

Related functions are **clearerr( )**, **ferror( )**, **perror( )**, **putc( )**, and **getc( )**.

ferror

```
#include <cstdio>
int ferror(FILE *stream);
```

The **ferror( )** function checks for a file error on the given *stream*. A return value of zero indicates that no error has occurred, while a nonzero value means an error.

To determine the exact nature of the error, use the **perror( )** function.

Related functions are **clearerr( )**, **feof( )**, and **perror( )**.

fflush

```
#include <cstdio>
int fflush(FILE *stream);
```

If *stream* is associated with a file opened for writing, a call to **fflush( )** causes the contents of the output buffer to be physically written to the file. The file remains open.

A return value of zero indicates success; **EOF** indicates that a write error has occurred.

All buffers are automatically flushed upon normal termination of the program or when they are full. Also, closing a file flushes its buffer.

Related functions are **fclose( )**, **fopen( )**, **fread( )**, **fwrite( )**, **getc( )**, and **putc( )**.

fgetc

```
#include <cstdio>
int fgetc(FILE *stream);
```

The **fgetc( )** function returns the next character from the input *stream* from the current position and increments the file position indicator. The character is read as an **unsigned char** that is converted to an integer.

If the end of the file is reached, **fgetc( )** returns **EOF**. However, since **EOF** is a valid integer value, when working with binary files you must use **feof( )** to check for the end of the file. If **fgetc( )** encounters an error, **EOF** is also returned. If working with binary files, you must use **ferror( )** to check for file errors.

Related functions are **fputc( )**, **getc( )**, **putc( )**, and **fopen( )**.

fgetpos

```
#include <cstdio>
int fgetpos(FILE *stream, fpos_t *position);
```

The **fgetpos( )** function stores the current value of the file position indicator in the object pointed to by *position*. The object pointed to by *position* must be of type **fpos_t**. The value stored there is useful only in a subsequent call to **fsetpos( )**.

If an error occurs, **fgetpos( )** returns nonzero; otherwise it returns zero.

Related functions are **fsetpos( )**, **fseek( )**, and **ftell( )**.

fgets

```
#include <cstdio>
char *fgets(char *str, int num, FILE *stream);
```

The **fgets( )** function reads up to *num*-1 characters from *stream* and places them into the character array pointed to by *str*. Characters are read until either a newline or an **EOF** is received or until the specified limit is reached. After the characters have been read, a null is placed in the array immediately after the last character read. A newline character will be retained and will be part of the array pointed to by *str*.

If successful, **fgets( )** returns *str*; a null pointer is returned upon failure. If a read error occurs, the contents of the array pointed to by *str* are indeterminate. Because a null pointer will be returned when either an error has occurred or when the end of the file is reached, you should use **feof( )** or **ferror( )** to determine what has actually happened.

Related functions are **fputs( )**, **fgetc( )**, **gets( )**, and **puts( )**.

fopen

```
#include <cstdio>
FILE *fopen(const char *fname, const char *mode);
```

The **fopen( )** function opens a file whose name is pointed to by *fname* and returns the stream that is associated with it. The type of operations that will be allowed on the file are defined by the value of *mode*. The legal values for *mode* are shown in Table 25-1. The filename must be a string of characters comprising a valid filename as defined by the operating system and may include a path specification if the environment supports it.

If **fopen( )** is successful in opening the specified file, a **FILE** pointer is returned. If the file cannot be opened, a null pointer is returned.

Mode	Meaning
"r"	Open text file for reading.
"w"	Create a text file for writing.
"a"	Append to text file.
"rb"	Open binary file for reading.
"wb"	Create binary file for writing.
"ab"	Append to a binary file.
"r+"	Open text file for read/write.
"w+"	Create text file for read/write.
"a+"	Open text file for read/write.
"rb+" or "r+b"	Open binary file for read/write.
"wb+" or "w+b"	Create binary file for read/write.
"ab+" or "a+b"	Open binary file for read/write.

Table 25-1. *The Legal Values for the mode Parameter of* ***fopen( )***

As the table shows, a file may be opened in either text or binary mode. In text mode, some character translations may occur. For example, newlines may be converted into carriage return/linefeed sequences. No such translations occur on binary files.

The correct method of opening a file is illustrated by this code fragment:

```
FILE *fp;

if ((fp = fopen("test", "w"))==NULL) {
  printf("Cannot open file.\n");
  exit(1);
}
```

This method detects any error in opening a file, such as a write-protected or a full disk, before attempting to write to it.

If you use **fopen( )** to open a file for output, any preexisting file by that name will be erased and a new file started. If no file by that name exists, one will be created.

Opening a file for read operations requires that the file exists. If it does not exist, an error will be returned. If you want to add to the end of the file, you must use mode "a." If the file does not exist, it will be created.

When accessing a file opened for read/write operations, you cannot follow an output operation with an input operation without an intervening call to either **fflush( )**, **fseek( )**, **fsetpos( )**, or **rewind( )**. Also, you cannot follow an input operation with an output operation without an intervening call to one of the previously mentioned functions, except when the end of the file is reached during input. That is, output can directly follow input at the end of the file.

Related functions are **fclose( )**, **fread( )**, **fwrite( )**, **putc( )**, and **getc( )**.

fprintf

```
#include <cstdio>
int fprintf(FILE *stream, const char *format, ...);
```

The **fprintf( )** function outputs the values of the arguments that comprise the argument list as specified in the *format* string to the stream pointed to by *stream*. The return value is the number of characters actually printed. If an error occurs, a negative number is returned.

There may be from zero to several arguments, with the maximum number being system dependent.

The operations of the format control string and commands are identical to those in **printf( )**; see **printf( )** for a complete description.

Related functions are **printf( )** and **fscanf( )**.

fputc

```
#include <cstdio>
int fputc(int ch, FILE *stream);
```

The **fputc( )** function writes the character *ch* to the specified stream at the current file position and then advances the file position indicator. Even though *ch* is declared to be an **int** for historical reasons, it is converted by **fputc( )** into an **unsigned char**. Because all character arguments are elevated to integers at the time of the call, you will generally see character values used as arguments. If an integer were used, the high-order byte(s) would simply be discarded.

The value returned by **fputc( )** is the value of the character written. If an error occurs, **EOF** is returned. For files opened for binary operations, an **EOF** may be a valid character, and the function **ferror( )** will need to be used to determine whether an error has actually occurred.

Related functions are **fgetc( )**, **fopen( )**, **fprintf( )**, **fread( )**, and **fwrite( )**.

fputs

```
#include <cstdio>
int fputs(const char *str, FILE *stream);
```

The **fputs( )** function writes the contents of the string pointed to by *str* to the specified stream. The null terminator is not written.

The **fputs( )** function returns nonnegative on success and **EOF** on failure.

If the stream is opened in text mode, certain character translations may take place. This means that there may not be a one-to-one mapping of the string onto the file. However, if the stream is opened in binary mode, no character translations will occur, and a one-to-one mapping between the string and the file will exist.

Related functions are **fgets( )**, **gets( )**, **puts( )**, **fprintf( )**, and **fscanf( )**.

fread

```
#include <cstdio>
size_t fread(void *buf, size_t size, size_t count,
             FILE *stream);
```

The **fread( )** function reads *count* number of objects, each object being *size* bytes in length, from the stream pointed to by *stream* and places them in the array pointed to by *buf*. The file position indicator is advanced by the number of characters read.

The **fread( )** function returns the number of items actually read. If fewer items are read than are requested in the call, either an error has occurred or the end of the file has been reached. You must use **feof( )** or **ferror( )** to determine what has taken place.

If the stream is opened for text operations, certain character translations, such as carriage return/linefeed sequences being transformed into newlines, may occur.

Related functions are **fwrite( )**, **fopen( )**, **fscanf( )**, **fgetc( )**, and **getc( )**.

freopen

```
#include <cstdio>
FILE *freopen(const char *fname, const char *mode,
              FILE *stream);
```

The **freopen( )** function associates an existing stream with a different file. The new file's name is pointed to by *fname*, the access mode is pointed to by *mode*, and the stream to be reassigned is pointed to by *stream*. The string *mode* uses the same format as **fopen( )**; a complete discussion is found in the **fopen( )** description.

When called, **freopen( )** first tries to close a file that may currently be associated with *stream*. However, if the attempt to close the file fails, the **freopen( )** function still continues to open the other file.

The **freopen( )** function returns a pointer to *stream* on success and a null pointer otherwise.

The main use of **freopen( )** is to redirect the system defined streams **stdin**, **stdout**, and **stderr** to some other file.

Related functions are **fopen( )** and **fclose( )**.

fscanf

```
#include <cstdio>
int fscanf(FILE *stream, const char *format, ...);
```

The **fscanf( )** function works exactly like the **scanf( )** function, except that it reads the information from the stream specified by *stream* instead of **stdin**. See **scanf( )** for details.

The **fscanf( )** function returns the number of arguments actually assigned values. This number does not include skipped fields. A return value of **EOF** means that a failure occurred before the first assignment was made.

Related functions are **scanf( )** and **fprintf( )**.

fseek

```
#include <cstdio>
int fseek(FILE *stream, long offset, int origin);
```

The **fseek( )** function sets the file position indicator associated with stream according to the values of *offset* and *origin*. Its purpose is to support random-access I/O operations. The *offset* is the number of bytes from *origin* to seek to. The values for *origin* must be one of these macros (defined in **<cstdio>**).

Name	Meaning
SEEK_SET	Seek from start of file
SEEK_CUR	Seek from current location
SEEK_END	Seek from end of file

A return value of zero means that **fseek( )** succeeded. A nonzero value indicates failure.

You may use **fseek( )** to move the position indicator anywhere in the file, even beyond the end. However, it is an error to attempt to set the position indicator before the beginning of the file.

The **fseek( )** function clears the end-of-file flag associated with the specified stream. Furthermore, it nullifies any prior **ungetc( )** on the same stream (see **ungetc( )**).

Related functions are **ftell( )**, **rewind( )**, **fopen( )**, **fgetpos( )**, and **fsetpos( )**.

fsetpos

```
#include <cstdio>
int fsetpos(FILE *stream, const fpos_t *position);
```

The **fsetpos( )** function moves the file position indicator to the point specified by the object pointed to by *position*. This value must have been previously obtained through a call to **fgetpos( )**. After **fsetpos( )** is executed, the end-of-file indicator is reset. Also, any previous call to **ungetc( )** is nullified.

If **fsetpos( )** fails, it returns nonzero. If it is successful, it returns zero.

Related functions are **fgetpos( )**, **fseek( )**, and **ftell( )**.

ftell

```
#include <cstdio>
long ftell(FILE *stream);
```

The **ftell( )** function returns the current value of the file position indicator for the specified *stream*. In the case of binary streams, the value is the number of bytes the indicator is from the beginning of the file. For text streams, the return value may not be meaningful except as an argument to **fseek( )** because of possible character translations, such as carriage return/linefeeds being substituted for newlines, which affect the apparent size of the file.

The **ftell( )** function returns –1 when an error occurs.

Related functions are **fseek( )** and **fgetpos( )**.

fwrite

```
#include <cstdio>
size_t fwrite(const void *buf, size_t size,
              size_t count, FILE *stream);
```

The **fwrite( )** function writes *count* number of objects, each object being *size* bytes in length, to the stream pointed to by *stream* from the character array pointed to by *buf*. The file position indicator is advanced by the number of characters written.

The **fwrite( )** function returns the number of items actually written, which, if the function is successful, will equal the number requested. If fewer items are written than are requested, an error has occurred.

Related functions are **fread( )**, **fscanf( )**, **getc( )**, and **fgetc( )**.

getc

```
#include <cstdio>
int getc(FILE *stream);
```

The **getc( )** function returns the next character from the input stream and increments the file position indicator. The character is read as an **unsigned char** that is converted to an integer.

If the end of the file is reached, **getc( )** returns **EOF**. However, since **EOF** is a valid integer value, when working with binary files you must use **feof( )** to check for the end-of-file character. If **getc( )** encounters an error, **EOF** is also returned. If working with binary files, you must use **ferror( )** to check for file errors.

The functions **getc( )** and **fgetc( )** are identical, and in most implementations **getc( )** is simply defined as the macro shown here.

```
#define getc(fp) fgetc(fp)
```

This causes the **fgetc( )** function to be substituted for the **getc( )** macro.

Related functions are **fputc( )**, **fgetc( )**, **putc( )**, and **fopen( )**.

getchar

```
#include <cstdio>
int getchar(void);
```

The **getchar( )** function returns the next character from **stdin**. The character is read as an **unsigned char** that is converted to an integer.

If the end of the file is reached, **getchar( )** returns **EOF**. If **getchar( )** encounters an error, **EOF** is also returned.

The **getchar( )** function is often implemented as a macro.

Related functions are **fputc( )**, **fgetc( )**, **putc( )**, and **fopen( )**.

gets

```
#include <cstdio>
char *gets(char *str);
```

The **gets( )** function reads characters from **stdin** and places them into the character array pointed to by *str*. Characters are read until a newline or an **EOF** is received. The newline character is not made part of the string; instead, it is translated into a null to terminate the string.

If successful, **gets( )** returns *str*; a null pointer is returned upon failure. If a read error occurs, the contents of the array pointed to by *str* are indeterminate. Because a null pointer will be returned when either an error has occurred or when the end of the file is reached, you should use **feof( )** or **ferror( )** to determine what has actually happened.

There is no way to limit the number of characters that **gets( )** will read, and it is possible for the array pointed to by *str* to be overrun. Thus, **gets( )** is inherently dangerous.

Related functions are **fputs( )**, **fgetc( )**, **fgets( )**, and **puts( )**.

perror

```
#include <cstdio>
void perror(const char *str);
```

The **perror( )** function maps the value of the global variable **errno** onto a string and writes that string to **stderr**. If the value of *str* is not null, it is written first, followed by a colon, and then the implementation-defined error message.

printf

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

The **printf( )** function writes to **stdout** the arguments that comprise the argument list as specified by the string pointed to by *format*.

The string pointed to by *format* consists of two types of items. The first type is made up of characters that will be printed on the screen. The second type contains format specifiers that define the way the arguments are displayed. A format specifier begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order. For example, the following **printf( )** call displays "Hi c 10 there!".

```
printf("Hi %c %d %s", 'c', 10, "there!");
```

If there are insufficient arguments to match the format specifiers, the output is undefined. If there are more arguments than format specifiers, the remaining arguments are discarded. The format specifiers are shown in Table 25-2.

The **printf( )** function returns the number of characters actually printed. A negative return value indicates that an error has taken place.

The format codes may have modifiers that specify the field width, precision, and a left-justification flag. An integer placed between the % sign and the format code acts as a *minimum field-width specifier*. This pads the output with spaces or 0's to ensure that it is at least a certain minimum length. If the string or number is greater than that minimum, it will be printed in full, even if it overruns the minimum. The default padding is done with spaces. If you wish to pad with 0's, place a 0 before the field-width specifier. For example, **%05d** will pad a number of less than five digits with 0's so that its total length is 5.

The exact meaning of the *precision modifier* depends on the format code being modified. To add a precision modifier, place a decimal point followed by the precision after the field-width specifier. For **e**, **E**, and **f** formats, the precision modifier determines

Code	Format
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter (if %e, uses lowercase e)
%G	Uses %E or %f, whichever is shorter (if %E, uses uppercase E)
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument is a pointer to an integer into which is placed the number of characters written so far
%%	Prints a % sign

Table 25-2. *The **printf( )** Format Specifiers*

the number of decimal places printed. For example, **%10.4f** will display a number at least 10 characters wide with four decimal places. When the precision modifier is applied to the **g** or **G** format code, it determines the maximum number of significant digits displayed. When applied to integers, the precision modifier specifies the minimum number of digits that will be displayed. Leading zeros are added, if necessary.

When the precision modifier is applied to strings, the number following the period specifies the maximum field length. For example, **%5.7s** will display a string that will

be at least five characters long and will not exceed seven. If the string is longer than the maximum field width, the characters will be truncated off the end.

By default, all output is *right-justified*: if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force the information to be left-justified by placing a minus sign directly after the %. For example, **%-10.2f** will left-justify a floating-point number with two decimal places in a 10-character field.

There are two format modifiers that allow **printf( )** to display short and long integers. These modifiers may be applied to the **d**, **i**, **o**, **u**, and **x** type specifiers. The **l** modifier tells **printf( )** that a long data type follows. For example, **%ld** means that a long integer is to be displayed. The **h** modifier tells **printf( )** to display a short integer. Therefore, **%hu** indicates that the data is of type short unsigned integer.

If you are using a modern compiler that supports the wide-character features added in 1995, then you may use the **l** modifier with the **c** specifier to indicate a wide-character of type **wchar_t**. You may also use the **l** modifier with the **s** format command to indicate a wide-character string.

An **L** modifier may prefix the floating-point commands of **e**, **f**, and **g** and indicates that a **long double** follows.

The **%n** command causes the number of characters that have been written at the time the **%n** is encountered to be placed in an integer variable whose pointer is specified in the argument list. For example, this code fragment displays the number 14 after the line "This is a test":

```
int i;

printf("This is a test%n", &i);
printf("%d", i);
```

You can apply the **l** or **h** modifer to the **n** specifier to indicate that the corresponding argument points to a long or short integer, respectively.

The **#** has a special meaning when used with some **printf( )** format codes. Preceding a **g**, **G**, **f**, **e**, or **E** code with a **#** ensures that the decimal point will be present, even if there are no decimal digits. If you precede the **x** or **X** format code with a #, the hexadecimal number will be printed with a **0x** prefix. If you precede the **o** format with a **#**, the octal value will be printed with a **0** prefix. The **#** cannot be applied to any other format specifiers.

The minimum field-width and precision specifiers may be provided by arguments to **printf( )** instead of by constants. To accomplish this, use an * as a placeholder. When the format string is scanned, **printf( )** will match each * to an argument in the order in which they occur.

Related functions are **scanf( )** and **fprintf( )**

putc

```
#include <cstdio>
int putc(int ch, FILE *stream);
```

The **putc( )** function writes the character contained in the least significant byte of *ch* to the output stream pointed to by *stream*. Because character arguments are elevated to integer at the time of the call, you may use character values as arguments to **putc( )**.

The **putc( )** function returns the character written on success or **EOF** if an error occurs. If the output stream has been opened in binary mode, **EOF** is a valid value for *ch*. This means that you must use **ferror( )** to determine if an error has occurred.

Related functions are **fgetc( )**, **fputc( )**, **getchar( )**, and **putchar( )**.

putchar

```
#include <cstdio>
int putchar(int ch);
```

The **putchar( )** function writes the character contained in the least significant byte of *ch* to **stdout**. It is functionally equivalent to **putc(ch, stdout)**. Because character arguments are elevated to integer at the time of the call, you may use character values as arguments to **putchar( )**.

The **putchar( )** function returns the character written on success or **EOF** if an error occurs.

A related function is **putc( )**.

puts

```
#include <cstdio>
int puts(const char *str);
```

The **puts( )** function writes the string pointed to by *str* to the standard output device. The null terminator is translated to a newline.

The **puts( )** function returns a nonnegative value if successful and an **EOF** upon failure.

Related functions are **putc( )**, **gets( )**, and **printf( )**.

remove

```
#include <cstdio>
int remove(const char *fname);
```

The **remove( )** function erases the file specified by *fname*. It returns zero if the file was successfully deleted and nonzero if an error occurred.

A related function is **rename( )**.

rename

```
#include <cstdio>
int rename(const char *oldfname, const char *newfname);
```

The **rename( )** function changes the name of the file specified by *oldfname* to *newfname*. The *newfname* must not match any existing directory entry.

The **rename( )** function returns zero if successful and nonzero if an error has occurred.

A related function is **remove( )**.

rewind

```
#include <cstdio>
void rewind(FILE *stream);
```

The **rewind( )** function moves the file position indicator to the start of the specified stream. It also clears the end-of-file and error flags associated with *stream*. It has no return value.

A related function is **fseek( )**.

scanf

```
#include <cstdio>
int scanf(const char *format, ...);
```

The **scanf( )** function is a general-purpose input routine that reads the stream **stdin** and stores the information in the variables pointed to in its argument list. It can read all the built-in data types and automatically convert them into the proper internal format.

The control string pointed to by *format* consists of three classifications of characters:

Format specifiers
White-space characters
Non–white-space characters

The input format specifiers begin with a % sign and tell **scanf( )** what type of data is to be read next. The format specifiers are listed in Table 25-3. For example, **%s** reads a string while **%d** reads an integer. The format string is read left to right and the format specifiers are matched, in order, with the arguments that comprise the argument list.

Code	Meaning
%c	Reads a single character.
%d	Reads a decimal integer.
%i	Reads an integer.
%e	Reads a floating-point number.
%f	Reads a floating-point number.
%g	Reads a floating-point number.
%o	Reads an octal number.
%s	Reads a string.
%x	Reads a hexadecimal number.
%p	Reads a pointer.
%n	Receives an integer value equal to the number of characters read so far.
%u	Reads an unsigned integer.
%[ ]	Scans for a set of characters.
%%	Reads a percent sign.

Table 25-3. *The **scanf( )** Format Specifiers*

To read a long integer, put an **l** (*ell*) in front of the format specifier. To read a short integer, put an **h** in front of the format specifier. These modifiers can be used with the **d**, **i**, **o**, **u**, and **x** format codes.

By default, the **f**, **e**, and **g** specifiers instruct **scanf( )** to assign data to a **float**. If you put an **l** (*ell*) in front of one of these specifiers, **scanf( )** assigns the data to a **double**. Using an **L** tells **scanf( )** that the variable receiving the data is a **long double**.

If you are using a modern compiler that supports wide-character features added in 1995, you may use the **l** modifier with the **c** format code to indicate a pointer to a wide character of type **wchar_t**. You may also use the **l** modifier with the **s** format code to indicate a pointer to a wide-character string. The **l** may also be used to modify a scanset to indicate wide characters.

A white-space character in the format string causes **scanf( )** to skip over one or more white-space characters in the input stream. A white-space character is either a space, a tab character, or a newline. In essence, one white-space character in the control string will cause **scanf( )** to read, but not store, any number (including zero) of white-space characters up to the first non–white-space character.

A non–white-space character in the format string causes **scanf( )** to read and discard a matching character. For example, **%d,%d** causes **scanf( )** to first read an integer, then read and discard a comma, and finally read another integer. If the specified character is not found, **scanf( )** will terminate.

All the variables used to receive values through **scanf( )** must be passed by their addresses. This means that all arguments must be pointers.

The input data items must be separated by spaces, tabs, or newlines. Punctuation such as commas, semicolons, and the like do not count as separators. This means that

```
scanf("%d%d", &r, &c);
```

will accept an input of **10 20** but fail with **10,20**.

An * placed after the % and before the format code will read data of the specified type but suppress its assignment. Thus, the command

```
scanf("%d%*c%d", &x, &y);
```

given the input **10/20**, will place the value 10 into **x**, discard the divide sign, and give **y** the value 20.

The format commands can specify a maximum field-length modifier. This is an integer number placed between the % and the format code that limits the number of characters read for any field. For example, if you wish to read no more than 20 characters into **address**, you would write the following.

```
scanf("%20s", address);
```

If the input stream were greater than 20 characters, a subsequent call to input would begin where this call left off. Input for a field may terminate before the maximum field length is reached if a white space is encountered. In this case, **scanf( )** moves on to the next field.

Although spaces, tabs, and newlines are used as field separators, when reading a single character, these are read like any other character. For example, with an input stream of **x y**,

```
scanf("%c%c%c", &a, &b, &c);
```

will return with the character x in **a**, a space in **b** and the character y in **c**.

Beware: Any other characters in the control string—including spaces, tabs, and newlines—will be used to match and discard characters from the input stream. Any character that matches is discarded. For example, given the input stream **10t20**,

```
scanf("%dt%d", &x, &y);
```

will place 10 into **x** and 20 into **y**. The **t** is discarded because of the **t** in the control string.

Another feature of **scanf( )** is called a *scanset*. A scanset defines a set of characters that will be read by **scanf( )** and assigned to the corresponding character array. A scanset is defined by putting the characters you want to scan for inside square brackets. The beginning square bracket must be prefixed by a percent sign. For example, this scanset tells **scanf( )** to read only the characters A, B, and C:

```
%[ABC]
```

When a scanset is used, **scanf( )** continues to read characters and put them into the corresponding character array until a character that is not in the scanset is encountered. The corresponding variable must be a pointer to a character array. Upon return from **scanf( )**, the array will contain a null-terminated string comprised of the characters read.

You can specify an inverted set if the first character in the set is a ^. When the ^ is present, it instructs **scanf( )** to accept any character that *is not* defined by the scanset.

For many implementations, you can specify a range using a hyphen. For example, this tells **scanf( )** to accept the characters A through Z.

```
%[A-Z]
```

One important point to remember is that the scanset is case sensitive. Therefore, if you want to scan for both upper- and lowercase letters, they must be specified individually.

The **scanf( )** function returns a number equal to the number of fields that were successfully assigned values. This number will not include fields that were read but not assigned because the * modifier was used to suppress the assignment. **EOF** is returned if an error occurs before the first field is assigned.

Related functions are **printf( )** and **fscanf( )**.

setbuf

```
#include <cstdio>
void setbuf(FILE *stream, char *buf);
```

The **setbuf( )** function is used either to specify the buffer that *stream* will use or, if called with *buf* set to null, to turn off buffering. If a programmer-defined buffer is to be specified, it must be **BUFSIZ** characters long. **BUFSIZ** is defined in **<cstdio>**.

The **setbuf( )** function returns no value.

Related functions are **fopen( )**, **fclose( )**, and **setvbuf( )**.

setvbuf

```
#include <cstdio>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The **setvbuf( )** function allows the programmer to specify the buffer, its size, and its mode for the specified stream. The character array pointed to by *buf* is used as the stream buffer for I/O operations. The size of the buffer is set by *size*, and *mode* determines how buffering will be handled. If *buf* is null, **setvbuf( )** will allocate its own buffer.

The legal values of *mode* are **_IOFBF**, **_IONBF**, and **_IOLBF**. These are defined in **<cstdio>**. When *mode* is set to **_IOFBF**, full buffering will take place. If *mode* is **_IOLBF**, the stream will be line buffered, which means that the buffer will be flushed each time a newline character is written for output streams; for input streams, input is buffered until a newline character is read. If mode is **_IONBF**, no buffering takes place.

The **setvbuf( )** function returns zero on success, nonzero on failure.

A related function is **setbuf( )**.

sprintf

```
#include <cstdio>
int sprintf(char *buf, const char *format, ...);
```

The **sprintf( )** function is identical to **printf( )** except that the output is put into the array pointed to by *buf* instead of being written to the console. See **printf( )** for details.

The return value is equal to the number of characters actually placed into the array.

Related functions are **printf( )** and **fsprintf( )**.

sscanf

```
#include <cstdio>
int sscanf(const char *buf, const char *format, ...);
```

The **sscanf( )** function is identical to **scanf( )** except that data is read from the array pointed to by *buf* rather than **stdin**. See **scanf( )** for details.

The return value is equal to the number of variables that were actually assigned values. This number does not include fields that were skipped through the use of the * format command modifier. A value of zero means that no fields were assigned, and **EOF** indicates that an error occurred prior to the first assignment.

Related functions are **scanf( )** and **fscanf( )**.

tmpfile

```
#include <cstdio>
FILE *tmpfile(void);
```

The **tmpfile( )** function opens a temporary file for update and returns a pointer to the stream. The function automatically uses a unique filename to avoid conflicts with existing files.

The **tmpfile( )** function returns a null pointer on failure; otherwise it returns a pointer to the stream.

The temporary file created by **tmpfile( )** is automatically removed when the file is closed or when the program terminates.

A related function is **tmpnam( )**.

tmpnam

```
#include <cstdio>
char *tmpnam(char *name);
```

The **tmpnam( )** function generates a unique filename and stores it in the array pointed to by *name*. This array must be at least **L_tmpnam** characters long. (**L_tmpnam** is defined in **<cstdio>**.) The main purpose of **tmpnam( )** is to generate a temporary filename that is different from any other file in the current disk directory.

The function may be called up to **TMP_MAX** times. **TMP_MAX** is defined in **<cstdio>**, and it will be at least 25. Each time **tmpnam( )** is called, it will generate a new temporary filename.

A pointer to *name* is returned on success; otherwise a null pointer is returned. If *name* is null, the temporary filename is held in a static array owned by **tmpnam( )**, and a pointer to this array is returned. This array is overwritten by a subsequent call.

A related function is **tmpfile( )**.

ungetc

```
#include <cstdio>
int ungetc(int ch, FILE *stream);
```

The **ungetc( )** function returns the character specified by the low-order byte of *ch* to the input stream *stream*. This character will then be obtained by the next read operation on *stream*. A call to **fflush( )**, **fseek( )**, or **rewind( )** undoes an **ungetc( )** operation and discards the character.

A one-character pushback is guaranteed; however, some implementations will accept more.

You may not unget an **EOF**.

A call to **ungetc( )** clears the end-of-file flag associated with the specified stream. The value of the file position indicator for a text stream is undefined until all pushed- back characters are read, in which case it will be the same as it was prior to the first **ungetc( )** call. For binary streams, each **ungetc( )** call decrements the file position indicator.

The return value is equal to *ch* on success and **EOF** on failure.

A related function is **getc( )**.

vprintf, vfprintf, and vsprintf

```
#include <cstdarg>
#include <cstdio>
int vprintf(char *format, va_list arg_ptr);
int vfprintf(FILE *stream, const char *format,
             va_list arg_ptr);
int vsprintf(char *buf, const char *format,
             va_list arg_ptr);
```

The functions **vprintf( )**, **vfprintf( )**, and **vsprintf( )** are functionally equivalent to **printf( )**, **fprintf( )**, and **sprintf( )**, respectively, except that the argument list has been replaced by a pointer to a list of arguments. This pointer must be of type **va_list**, which is defined in the header **<cstdarg>** (or the C header file **stdarg.h**).

Related functions are **va_arg( )**, **va_start( )**, and **va_end( )**.