

Chapter 22

Input/Output

Introduction

- C's input/output library is the biggest and most important part of the standard library.
- The `<stdio.h>` header is the primary repository of input/output functions, including `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`.
- This chapter provides more information about these six functions.
- It also introduces many new functions, most of which deal with files.

Introduction

- Topics to be covered:
 - Streams, the `FILE` type, input and output redirection, and the difference between text files and binary files
 - Functions designed specifically for use with files, including functions that open and close files
 - Functions that perform “formatted” input/output
 - Functions that read and write unformatted data (characters, lines, and blocks)
 - Random access operations on files
 - Functions that write to a string or read from a string

Introduction

- In C99, some I/O functions belong to the `<wchar.h>` header.
- The `<wchar.h>` functions deal with wide characters rather than ordinary characters.
- Functions in `<stdio.h>` that read or write data are known as ***byte input/output functions***.
- Similar functions in `<wchar.h>` are called ***wide-character input/output functions***.

Streams

- In C, the term *stream* means any source of input or any destination for output.
- Many small programs obtain all their input from one stream (the keyboard) and write all their output to another stream (the screen).
- Larger programs may need additional streams.
- Streams often represent files stored on various media.
- However, they could just as easily be associated with devices such as network ports and printers.

File Pointers

- Accessing a stream is done through a *file pointer*, which has type `FILE *`.
- The `FILE` type is declared in `<stdio.h>`.
- Certain streams are represented by file pointers with standard names.
- Additional file pointers can be declared as needed:
`FILE *fp1, *fp2;`

Standard Streams and Redirection

- `<stdio.h>` provides three standard streams:

<i>File Pointer</i>	<i>Stream</i>	<i>Default Meaning</i>
<code>stdin</code>	Standard input	Keyboard
<code>stdout</code>	Standard output	Screen
<code>stderr</code>	Standard error	Screen

- These streams are ready to use—we don't declare them, and we don't open or close them.

Standard Streams and Redirection

- The I/O functions discussed in previous chapters obtain input from `stdin` and send output to `stdout`.
- Many operating systems allow these default meanings to be changed via a mechanism known as *redirection*.

Standard Streams and Redirection

- A typical technique for forcing a program to obtain its input from a file instead of from the keyboard:
demo <in.dat
This technique is known as *input redirection*.
- *Output redirection* is similar:
demo >out.dat
All data written to stdout will now go into the out.dat file instead of appearing on the screen.

Standard Streams and Redirection

- Input redirection and output redirection can be combined:
demo <in.dat >out.dat
- The < and > characters don't have to be adjacent to file names, and the order in which the redirected files are listed doesn't matter:
demo < in.dat > out.dat
demo >out.dat <in.dat

Standard Streams and Redirection

- One problem with output redirection is that *everything* written to stdout is put into a file.
- Writing error messages to stderr instead of stdout guarantees that they will appear on the screen even when stdout has been redirected.

Text Files versus Binary Files

- <stdio.h> supports two kinds of files: text and binary.
- The bytes in a *text file* represent characters, allowing humans to examine or edit the file.
 - The source code for a C program is stored in a text file.
- In a *binary file*, bytes don't necessarily represent characters.
 - Groups of bytes might represent other types of data, such as integers and floating-point numbers.
 - An executable C program is stored in a binary file.

Text Files versus Binary Files

- Text files have two characteristics that binary files don't possess.
- Text files are divided into lines.** Each line in a text file normally ends with one or two special characters.
 - Windows: carriage-return character ('`\r`') followed by line-feed character ('`\n`')
 - UNIX and newer versions of Mac OS: line-feed character
 - Older versions of Mac OS: carriage-return character

Text Files versus Binary Files

- Text files may contain a special “end-of-file” marker.**
 - In Windows, the marker is '`\x1a`' (Ctrl-Z), but it is not required.
 - Most other operating systems, including UNIX, have no special end-of-file character.
- In a binary file, there are no end-of-line or end-of-file markers; all bytes are treated equally.

Text Files versus Binary Files

- When data is written to a file, it can be stored in text form or in binary form.
- One way to store the number 32767 in a file would be to write it in text form as the characters 3, 2, 7, 6, and 7:

00110011	00110010	00110111	00110110	00110111
'3'	'2'	'7'	'6'	'7'

Text Files versus Binary Files

- The other option is to store the number in binary, which would take as few as two bytes:

01111111	11111111
----------	----------

- Storing numbers in binary can often save space.

Text Files versus Binary Files

- Programs that read from a file or write to a file must take into account whether it's text or binary.
- A program that displays the contents of a file on the screen will probably assume it's a text file.
- A file-copying program, on the other hand, can't assume that the file to be copied is a text file.
 - If it does, binary files containing an end-of-file character won't be copied completely.
- When we can't say for sure whether a file is text or binary, it's safer to assume that it's binary.

File Operations

- Simplicity is one of the attractions of input and output redirection.
- Unfortunately, redirection is too limited for many applications.
 - When a program relies on redirection, it has no control over its files; it doesn't even know their names.
 - Redirection doesn't help if the program needs to read from two files or write to two files at the same time.
- When redirection isn't enough, we'll use the file operations that `<stdio.h>` provides.

Opening a File

- Opening a file for use as a stream requires a call of the `fopen` function.
- Prototype for `fopen`:


```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```
- `filename` is the name of the file to be opened.
 - This argument may include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a File

- The word `restrict` appears twice in the prototype for `fopen`.
- `restrict`, which is a C99 keyword, indicates that `filename` and `mode` should point to strings that don't share memory locations.
- The C89 prototype for `fopen` doesn't contain `restrict` but is otherwise identical.
- `restrict` has no effect on the behavior of `fopen`, so it can usually be ignored.

Opening a File

- In Windows, be careful when the file name in a call of `fopen` includes the `\` character.
- The call

```
fopen("c:\project\test1.dat", "r")
```

will fail, because `\t` is treated as a character escape.
- One way to avoid the problem is to use `\\` instead of `\`:

```
fopen("c:\\project\\test1.dat", "r")
```
- An alternative is to use the `/` character instead of `\`:

```
fopen("c:/project/test1.dat", "r")
```

Opening a File

- `fopen` returns a file pointer that the program can (and usually will) save in a variable:

```
fp = fopen("in.dat", "r");
/* opens in.dat for reading */
```
- When it can't open a file, `fopen` returns a null pointer.

Modes

- Factors that determine which mode string to pass to `fopen`:
 - Which operations are to be performed on the file
 - Whether the file contains text or binary data

Modes

- Mode strings for text files:

<i>String</i>	<i>Meaning</i>
"r"	Open for reading
"w"	Open for writing (file need not exist)
"a"	Open for appending (file need not exist)
"r+"	Open for reading and writing, starting at beginning
"w+"	Open for reading and writing (truncate if file exists)
"a+"	Open for reading and writing (append if file exists)

Modes

- Mode strings for binary files:

<i>String</i>	<i>Meaning</i>
"r"	Open for reading
"w"	Open for writing (file need not exist)
"a"	Open for appending (file need not exist)
"r+" or "r+b"	Open for reading and writing, starting at beginning
"w+" or "w+b"	Open for reading and writing (truncate if file exists)
"a+" or "a+b"	Open for reading and writing (append if file exists)

Modes

- Note that there are different mode strings for *writing* data and *appending* data.
- When data is written to a file, it normally overwrites what was previously there.
- When a file is opened for appending, data written to the file is added at the end.

Modes

- Special rules apply when a file is opened for both reading and writing.
 - Can't switch from reading to writing without first calling a file-positioning function unless the reading operation encountered the end of the file.
 - Can't switch from writing to reading without either calling `fflush` or calling a file-positioning function.

Closing a File

- The `fclose` function allows a program to close a file that it's no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of `fopen` or `freopen`.
- `fclose` returns zero if the file was closed successfully.
- Otherwise, it returns the error code EOF (a macro defined in `<stdio.h>`).

Closing a File

- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
    FILE *fp;

    fp = fopen(FILE_NAME, "r");
    if (fp == NULL) {
        printf("Can't open %s\n", FILE_NAME);
        exit(EXIT_FAILURE);
    }
    ...
    fclose(fp);
    return 0;
}
```

Closing a File

- It's not unusual to see the call of `fopen` combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against `NULL`:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) ...
```

Attaching a File to an Open Stream

- `freopen` attaches a different file to a stream that's already open.
- The most common use of `freopen` is to associate a file with one of the standard streams (`stdin`, `stdout`, or `stderr`).
- A call of `freopen` that causes a program to begin writing to the file `foo`:

```
if (freopen("foo", "w", stdout) == NULL)
{
    /* error; foo can't be opened */
}
```

Attaching a File to an Open Stream

- `freopen`'s normal return value is its third argument (a file pointer).
- If it can't open the new file, `freopen` returns a null pointer.

Attaching a File to an Open Stream

- C99 adds a new twist: if `filename` is a null pointer, `freopen` attempts to change the stream's mode to that specified by the mode parameter.
- Implementations aren't required to support this feature.
- If they do, they may place restrictions on which mode changes are permitted.

Obtaining File Names from the Command Line

- There are several ways to supply file names to a program.
 - Building file names into the program doesn't provide much flexibility.
 - Prompting the user to enter file names can be awkward.
 - Having the program obtain file names from the command line is often the best solution.
- An example that uses the command line to supply two file names to a program named `demo`:

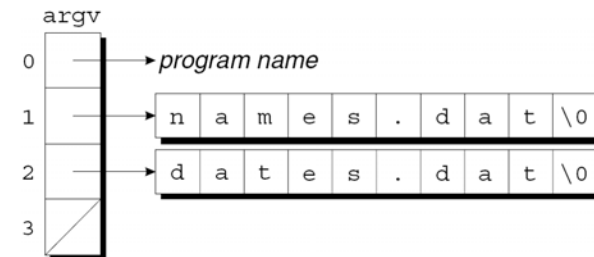
```
demo names.dat dates.dat
```

Obtaining File Names from the Command Line

- Chapter 13 showed how to access command-line arguments by defining `main` as a function with two parameters:
- ```
int main(int argc, char *argv[])
{
 ...
}
```
- `argc` is the number of command-line arguments.
  - `argv` is an array of pointers to the argument strings.

## Obtaining File Names from the Command Line

- `argv[0]` points to the program name, `argv[1]` through `argv[argc-1]` point to the remaining arguments, and `argv[argc]` is a null pointer.
- In the demo example, `argc` is 3 and `argv` has the following appearance:



## Program: Checking Whether a File Can Be Opened

- The `canopen.c` program determines if a file exists and can be opened for reading.
- The user will give the program a file name to check:  
`canopen file`
- The program will then print either *file* can be opened or *file* can't be opened.
- If the user enters the wrong number of arguments on the command line, the program will print the message `usage: canopen filename`.

### canopen.c

```
/* Checks whether a file can be opened for reading */
#include <stdio.h>
#include <stdlib.h>

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

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

 if ((fp = fopen(argv[1], "r")) == NULL) {
 printf("%s can't be opened\n", argv[1]);
 exit(EXIT_FAILURE);
 }

 printf("%s can be opened\n", argv[1]);
 fclose(fp);
 return 0;
}
```

## Temporary Files

- Programs often need to create temporary files—files that exist only as long as the program is running.
- `<stdio.h>` provides two functions, `tmpfile` and `tmpnam`, for working with temporary files.

## Temporary Files

- `tmpfile` creates a temporary file (opened in "wb+" mode) that will exist until it's closed or the program ends.
- A call of `tmpfile` returns a file pointer that can be used to access the file later:  
`FILE *temp_ptr;`  
...  
`temp_ptr = tmpfile();`  
/\* creates a temporary file \*/
- If it fails to create a file, `tmpfile` returns a null pointer.

## Temporary Files

- Drawbacks of using `tmpfile`:
  - Don't know the name of the file that `tmpfile` creates.
  - Can't decide later to make the file permanent.
- The alternative is to create a temporary file using `fopen`.
- The `tmpnam` function is useful for ensuring that this file doesn't have the same name as an existing file.

## Temporary Files

- `tmpnam` generates a name for a temporary file.
- If its argument is a null pointer, `tmpnam` stores the file name in a static variable and returns a pointer to it:

```
char *filename;
...
filename = tmpnam(NULL);
/* creates a temporary file name */
```

## Temporary Files

- Otherwise, `tmpnam` copies the file name into a character array provided by the programmer:
 

```
char filename[L_tmpnam];
...
tmpnam(filename);
/* creates a temporary file name */
```
- In this case, `tmpnam` also returns a pointer to the first character of this array.
- `L_tmpnam` is a macro in `<stdio.h>` that specifies how long to make a character array that will hold a temporary file name.

## Temporary Files

- The `TMP_MAX` macro (defined in `<stdio.h>`) specifies the maximum number of temporary file names that can be generated by `tmpnam`.
- If it fails to generate a file name, `tmpnam` returns a null pointer.

## File Buffering

- Transferring data to or from a disk drive is a relatively slow operation.
- The secret to achieving acceptable performance is *buffering*.
- Data written to a stream is actually stored in a buffer area in memory; when it's full (or the stream is closed), the buffer is "flushed."
- Input streams can be buffered in a similar way: the buffer contains data from the input device; input is read from this buffer instead of the device itself.

## File Buffering

- Buffering can result in enormous gains in efficiency, since reading a byte from a buffer or storing a byte in a buffer is very fast.
- It takes time to transfer the buffer contents to or from disk, but one large "block move" is much faster than many tiny byte moves.
- The functions in `<stdio.h>` perform buffering automatically when it seems advantageous.
- On rare occasions, we may need to use the functions `fflush`, `setbuf`, and `setvbuf`.

## File Buffering

- By calling `fflush`, a program can flush a file's buffer as often as it wishes.
- A call that flushes the buffer for the file associated with `fp`:  

```
fflush(fp); /* flushes buffer for fp */
```
- A call that flushes *all* output streams:  

```
fflush(NULL); /* flushes all buffers */
```
- `fflush` returns zero if it's successful and EOF if an error occurs.

## File Buffering

- `setvbuf` allows us to change the way a stream is buffered and to control the size and location of the buffer.
- The function's third argument specifies the kind of buffering desired:  
`_IOFBF` (full buffering)  
`_IOLBF` (line buffering)  
`_IONBF` (no buffering)
- Full buffering is the default for streams that aren't connected to interactive devices.

## File Buffering

- `setvbuf`'s second argument (if it's not a null pointer) is the address of the desired buffer.
- The buffer might have static storage duration, automatic storage duration, or even be allocated dynamically.
- `setvbuf`'s last argument is the number of bytes in the buffer.

## File Buffering

- A call of `setvbuf` that changes the buffering of stream to full buffering, using the N bytes in the buffer array as the buffer:

```
char buffer[N];
...
setvbuf(stream, buffer, _IOFBF, N);
```

- `setvbuf` must be called after `stream` is opened but before any other operations are performed on it.

## File Buffering

- It's also legal to call `setvbuf` with a null pointer as the second argument, which requests that `setvbuf` create a buffer with the specified size.
- `setvbuf` returns zero if it's successful.
- It returns a nonzero value if the mode argument is invalid or the request can't be honored.

## File Buffering

- `setbuf` is an older function that assumes default values for the buffering mode and buffer size.
- If `buf` is a null pointer, the call `setbuf(stream, buf)` is equivalent to `(void) setvbuf(stream, NULL, _IONBF, 0);`
- Otherwise, it's equivalent to `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);` where `BUFSIZ` is a macro defined in `<stdio.h>`.
- `setbuf` is considered to be obsolete.

## Miscellaneous File Operations

- The `remove` and `rename` functions allow a program to perform basic file management operations.
- Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*.
- Both functions return zero if they succeed and a nonzero value if they fail.

## Miscellaneous File Operations

- `remove` deletes a file:  

```
remove("foo");
```

*/\* deletes the file named "foo" \*/*
- If a program uses `fopen` (instead of `tmpfile`) to create a temporary file, it can use `remove` to delete the file before the program terminates.
- The effect of removing a file that's currently open is implementation-defined.

## Miscellaneous File Operations

- `rename` changes the name of a file:  

```
rename("foo", "bar");
```

*/\* renames "foo" to "bar" \*/*
- `rename` is handy for renaming a temporary file created using `fopen` if a program should decide to make it permanent.
  - If a file with the new name already exists, the effect is implementation-defined.
- `rename` may fail if asked to rename an open file.

## Formatted I/O

- The next group of library functions use format strings to control reading and writing.
- `printf` and related functions are able to convert data from numeric form to character form during output.
- `scanf` and related functions are able to convert data from character form to numeric form during input.

## The ...printf Functions

- The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.
- The prototypes for both functions end with the ... symbol (an *ellipsis*), which indicates a variable number of additional arguments:  

```
int fprintf(FILE * restrict stream,
 const char * restrict format, ...);
int printf(const char * restrict format, ...);
```
- Both functions return the number of characters written; a negative return value indicates that an error occurred.

## The ...printf Functions

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument:  

```
printf("Total: %d\n", total);
/* writes to stdout */
fprintf(fp, "Total: %d\n", total);
/* writes to fp */
```
- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

## The ...printf Functions

- `fprintf` works with any output stream.
- One of its most common uses is to write error messages to `stderr`:  

```
fprintf(stderr, "Error: data file can't be opened.\n");
```
- Writing a message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

## The ...printf Functions

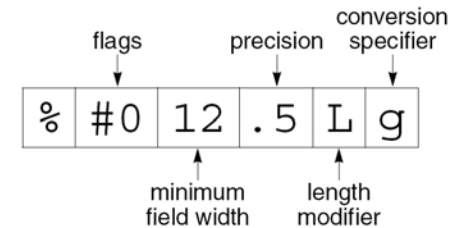
- Two other functions in `<stdio.h>` can write formatted output to a stream.
- These functions, named `vfprintf` and `vprintf`, are fairly obscure.
- Both rely on the `va_list` type, which is declared in `<stdarg.h>`, so they're discussed along with that header.

## ...printf Conversion Specifications

- Both `printf` and `fprintf` require a format string containing ordinary characters and/or conversion specifications.
  - Ordinary characters are printed as is.
  - Conversion specifications describe how the remaining arguments are to be converted to character form for display.

## ...printf Conversion Specifications

- A `...printf` conversion specification consists of the `%` character, followed by as many as five distinct items:



## ...printf Conversion Specifications

- Flags** (optional; more than one permitted):
 

| Flag  | Meaning                                                                                                                                                                                                                              |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -     | Left-justify within field.                                                                                                                                                                                                           |
| +     | Numbers produced by signed conversions always begin with + or -.                                                                                                                                                                     |
| space | Nonnegative numbers produced by signed conversions are preceded by a space.                                                                                                                                                          |
| #     | Octal numbers begin with 0, nonzero hexadecimal numbers with 0x or 0X. Floating-point numbers always have a decimal point. Trailing zeros aren't removed from numbers printed with the <code>g</code> or <code>G</code> conversions. |
| 0     | Numbers are padded with leading zeros up to the field width. (zero)                                                                                                                                                                  |

## ...printf Conversion Specifications

- Minimum field width** (optional). An item that's too small to occupy the field will be padded.
  - By default, spaces are added to the left of the item.
- An item that's too large for the field width will still be displayed in its entirety.
- The field width is either an integer or the character `*`.
  - If `*` is present, the field width is obtained from the next argument.



**...printf Conversion Specifications**

- **Precision** (optional). The meaning of the precision depends on the conversion:
  - d, i, o, u, x, X: minimum number of digits (leading zeros are added if the number has fewer digits)
  - a, A, e, E, f, F: number of digits after the decimal point
  - g, G: number of significant digits
  - s: maximum number of bytes
- The precision is a period (.) followed by an integer or the character \*.
  - If \* is present, the precision is obtained from the next argument.

**...printf Conversion Specifications**

- **Length modifier** (optional). Indicates that the item to be displayed has a type that's longer or shorter than normal.
  - %d normally refers to an int value; %hd is used to display a short int and %ld is used to display a long int.

**...printf Conversion Specifications**

| <i>Length Modifier</i> | <i>Conversion Specifiers</i> | <i>Meaning</i>                |
|------------------------|------------------------------|-------------------------------|
| hh <sup>†</sup>        | d, i, o, u, x, X             | signed char, unsigned char    |
|                        | n                            | signed char *                 |
| h                      | d, i, o, u, x, X             | short int, unsigned short int |
|                        | n                            | short int *                   |
| l                      | d, i, o, u, x, X             | long int, unsigned long int   |
| (ell)                  | n                            | long int *                    |
|                        | c                            | wint_t                        |
|                        | s                            | wchar_t *                     |
|                        | a, A, e, E, f, F, g, G       | no effect                     |

<sup>†</sup>C99 only**...printf Conversion Specifications**

| <i>Length Modifier</i> | <i>Conversion Specifiers</i> | <i>Meaning</i>         |
|------------------------|------------------------------|------------------------|
| ll <sup>†</sup>        | d, i, o, u, x, X             | long long int,         |
| (ell-ell)              |                              | unsigned long long int |
|                        | n                            | long long int *        |
| j <sup>†</sup>         | d, i, o, u, x, X             | intmax_t, uintmax_t    |
|                        | n                            | intmax_t *             |
| z <sup>†</sup>         | d, i, o, u, x, X             | size_t                 |
|                        | n                            | size_t *               |
| t <sup>†</sup>         | d, i, o, u, x, X             | ptrdiff_t              |
|                        | n                            | ptrdiff_t *            |
| L                      | a, A, e, E, f, F, g, G       | long double            |

<sup>†</sup>C99 only

## ...printf Conversion Specifications

- **Conversion specifier.** Must be one of the characters in the following table.

| Conversion Specifier | Meaning                                                                                                                                                                             |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| d, i                 | Converts an <code>int</code> value to decimal form.                                                                                                                                 |
| o, u, x, X           | Converts an unsigned <code>int</code> value to base 8 (o), base 10 (u), or base 16 (x, X). x displays the hexadecimal digits a–f in lower case; X displays them in upper case.      |
| f, F <sup>†</sup>    | Converts a <code>double</code> value to decimal form, putting the decimal point in the correct position. If no precision is specified, displays six digits after the decimal point. |

<sup>†</sup>C99 only

## ...printf Conversion Specifications

| Conversion Specifier            | Meaning                                                                                                                                                                                                                                                             |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| e, E                            | Converts a <code>double</code> value to scientific notation. If no precision is specified, displays six digits after the decimal point. If e is chosen, the exponent is preceded by the letter e; if E is chosen, the exponent is preceded by E.                    |
| g, G                            | g converts a <code>double</code> value to either f form or e form. G chooses between F and E forms.                                                                                                                                                                 |
| a <sup>†</sup> , A <sup>†</sup> | Converts a <code>double</code> value to hexadecimal scientific notation using the form <code>[-]0xh.hhhhp±d</code> . a displays the hex digits a–f in lower case; A displays them in upper case. The choice of a or A also affects the case of the letters x and p. |

<sup>†</sup>C99 only

## ...printf Conversion Specifications

| Conversion Specifier | Meaning                                                                                                                                                                                                  |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| c                    | Displays an <code>int</code> value as an unsigned character.                                                                                                                                             |
| s                    | Writes the characters pointed to by the argument. Stops writing when the number of bytes specified by the precision (if present) is reached or a null character is encountered.                          |
| p                    | Converts a <code>void *</code> value to printable form.                                                                                                                                                  |
| n                    | The corresponding argument must point to an object of type <code>int</code> . Stores in this object the number of characters written so far by this call of <code>...printf</code> ; produces no output. |
| %                    | Writes the character %.                                                                                                                                                                                  |

## C99 Changes to ...printf Conversion Specifications

- C99 changes to the conversion specifications for `printf` and `fprintf`:
  - Additional length modifiers
  - Additional conversion specifiers
  - Ability to write infinity and NaN
  - Support for wide characters
  - Previously undefined conversion specifications now allowed

## Examples of ...printf Conversion Specifications

- Examples showing the effect of flags on the %d conversion:

| <i>Conversion Specification</i> | <i>Result of Applying Conversion to 123</i> | <i>Result of Applying Conversion to -123</i> |
|---------------------------------|---------------------------------------------|----------------------------------------------|
| %d                              | .....123                                    | .....-123                                    |
| %-d                             | 123.....                                    | -123.....                                    |
| %+d                             | .....+123                                   | .....-123                                    |
| % 8d                            | .....123                                    | .....-123                                    |
| %08d                            | 00000123                                    | -0000123                                     |
| %-+8d                           | +123.....                                   | -123.....                                    |
| %- 8d                           | •123.....                                   | -123.....                                    |
| %+08d                           | +0000123                                    | -0000123                                     |
| % 08d                           | •0000123                                    | -0000123                                     |

## Examples of ...printf Conversion Specifications

- Examples showing the effect of the # flag on the o, x, X, g, and G conversions:

| <i>Conversion Specification</i> | <i>Result of Applying Conversion to 123</i> | <i>Result of Applying Conversion to 123.0</i> |
|---------------------------------|---------------------------------------------|-----------------------------------------------|
| %8o                             | .....173                                    |                                               |
| ##8o                            | .....0173                                   |                                               |
| %8x                             | .....7b                                     |                                               |
| ##8x                            | .....0x7b                                   |                                               |
| %8X                             | .....7B                                     |                                               |
| ##8X                            | .....0X7B                                   |                                               |
| %8g                             |                                             | .....123                                      |
| ##8g                            |                                             | •123.000                                      |
| %8G                             |                                             | .....123                                      |
| ##8G                            |                                             | •123.000                                      |

## Examples of ...printf Conversion Specifications

- Examples showing the effect of the minimum field width and precision on the %s conversion:

| <i>Conversion Specification</i> | <i>Result of Applying Conversion to "bogus"</i> | <i>Result of Applying Conversion to "buzzword"</i> |
|---------------------------------|-------------------------------------------------|----------------------------------------------------|
| %6s                             | •bogus                                          | buzzword                                           |
| %-6s                            | bogus•                                          | buzzword                                           |
| %.4s                            | bogu                                            | buzz                                               |
| %6.4s                           | ••bogu                                          | ••buzz                                             |
| %-6.4s                          | bogu••                                          | buzz••                                             |

## Examples of ...printf Conversion Specifications

- Examples showing how the %g conversion displays some numbers in %e form and others in %f form:

| <i>Number</i> | <i>Result of Applying %.4g Conversion to Number</i> |
|---------------|-----------------------------------------------------|
| 123456.       | 1.235e+05                                           |
| 12345.6       | 1.235e+04                                           |
| 1234.56       | 1235                                                |
| 123.456       | 123.5                                               |
| 12.3456       | 12.35                                               |
| 1.23456       | 1.235                                               |
| .123456       | 0.1235                                              |
| .0123456      | 0.01235                                             |
| .00123456     | 0.001235                                            |
| .000123456    | 0.0001235                                           |
| .0000123456   | 1.235e-05                                           |
| .00000123456  | 1.235e-06                                           |

## Examples of ...printf Conversion Specifications

- The minimum field width and precision are usually embedded in the format string.
- Putting the \* character where either number would normally go allows us to specify it as an argument *after* the format string.
- Calls of printf that produce the same output:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

## Examples of ...printf Conversion Specifications

- A major advantage of \* is that it allows us to use a macro to specify the width or precision:
- The width or precision can even be computed during program execution:

```
printf("%*d", WIDTH, i);

printf("%*d", page_width / num_cols, i);
```

## Examples of ...printf Conversion Specifications

- The %p conversion is used to print the value of a pointer:
- The pointer is likely to be shown as an octal or hexadecimal number.

```
printf("%p", (void *) ptr);
/* displays value of ptr */
```

## Examples of ...printf Conversion Specifications

- The %n conversion is used to find out how many characters have been printed so far by a call of ...printf.
- After the following call, the value of len will be 3:

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

## The ...scanf Functions

- `fscanf` and `scanf` read data items from an input stream, using a format string to indicate the layout of the input.
- After the format string, any number of pointers—each pointing to an object—follow as additional arguments.
- Input items are converted (according to conversion specifications in the format string) and stored in these objects.

## The ...scanf Functions

- `scanf` always reads from `stdin`, whereas `fscanf` reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);
/* reads from stdin */
fscanf(fp, "%d%d", &i, &j);
/* reads from fp */
```

- A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first argument.

## The ...scanf Functions

- Errors that cause the ...scanf functions to return prematurely:
  - *Input failure* (no more input characters could be read)
  - *Matching failure* (the input characters didn't match the format string)
- In C99, an input failure can also occur because of an *encoding error*.

## The ...scanf Functions

- The ...scanf functions return the number of data items that were read and assigned to objects.
- They return EOF if an input failure occurs before any data items can be read.
- Loops that test `scanf`'s return value are common.
- A loop that reads a series of integers one by one, stopping at the first sign of trouble:
 

```
while (scanf("%d", &i) == 1) {
 ...
}
```

## ...scanf Format Strings

- Calls of the ...scanf functions resemble those of the ...printf functions.
- However, the ...scanf functions work differently.
- The format string represents a pattern that a ...scanf function attempts to match as it reads input.
  - If the input doesn't match the format string, the function returns.
  - The input character that didn't match is "pushed back" to be read in the future.

## ...scanf Format Strings

- A ...scanf format string may contain three things:
  - Conversion specifications
  - White-space characters
  - Non-white-space characters

## ...scanf Format Strings

- **Conversion specifications.** Conversion specifications in a ...scanf format string resemble those in a ...printf format string.
- Most conversion specifications skip white-space characters at the beginning of an input item (the exceptions are %[ , %c, and %n).
- Conversion specifications never skip *trailing* white-space characters, however.

## ...scanf Format Strings

- **White-space characters.** One or more white-space characters in a format string match zero or more white-space characters in the input stream.
- **Non-white-space characters.** A non-white-space character other than % matches the same character in the input stream.

## ...scanf Format Strings

- The format string "ISBN %d-%d-%ld-%d" specifies that the input will consist of:
  - the letters ISBN
  - possibly some white-space characters
  - an integer
  - the - character
  - an integer (possibly preceded by white-space characters)
  - the - character
  - a long integer (possibly preceded by white-space characters)
  - the - character
  - an integer (possibly preceded by white-space characters)

## ...scanf Conversion Specifications

- A ...scanf conversion specification consists of the character % followed by:
  - \*
  - Maximum field width
  - Length modifier
  - Conversion specifier
- \* (optional). Signifies **assignment suppression**: an input item is read but not assigned to an object.
  - Items matched using \* aren't included in the count that ...scanf returns.

## ...scanf Conversion Specifications

- Maximum field width** (optional). Limits the number of characters in an input item.
  - White-space characters skipped at the beginning of a conversion don't count.
- Length modifier** (optional). Indicates that the object in which the input item will be stored has a type that's longer or shorter than normal.
- The table on the next slide lists each length modifier and the type indicated when it is combined with a conversion specifier.

## ...scanf Conversion Specifications

| Length Modifier | Conversion Specifiers                | Meaning                                      |
|-----------------|--------------------------------------|----------------------------------------------|
| hh <sup>†</sup> | d, i, o, u, x, X, n                  | signed char *, unsigned char *               |
| h               | d, i, o, u, x, X, n                  | short int *, unsigned short int *            |
| l               | d, i, o, u, x, X, n                  | long int *, unsigned long int *              |
| (ell)           | a, A, e, E, f, F, g, G<br>c, s, or [ | double *<br>wchar_t *                        |
| ll <sup>†</sup> | d, i, o, u, x, X, n                  | long long int *,<br>unsigned long long int * |
| (ell-ell)       | j <sup>†</sup>                       | intmax_t *, uintmax_t *                      |
|                 | z <sup>†</sup>                       | size_t *                                     |
|                 | t <sup>†</sup>                       | ptrdiff_t *                                  |
|                 | L                                    | long double *                                |

<sup>†</sup>C99 only



...scanf Conversion Specifications

- **Conversion specifier.** Must be one of the characters in the following table.

| Conversion Specifier | Meaning                                                                                                                                                                                                        |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| d                    | Matches a decimal integer; the corresponding argument is assumed to have type <code>int *</code> .                                                                                                             |
| i                    | Matches an integer; the corresponding argument is assumed to have type <code>int *</code> . The integer is assumed to be in base 10 unless it begins with 0 (indicating octal) or with 0x or 0X (hexadecimal). |
| o                    | Matches an octal integer; the corresponding argument is assumed to have type <code>unsigned int *</code> .                                                                                                     |
| u                    | Matches a decimal integer; the corresponding argument is assumed to have type <code>unsigned int *</code> .                                                                                                    |

...scanf Conversion Specifications

| Conversion Specifier                                             | Meaning                                                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| x, X                                                             | Matches a hexadecimal integer; the corresponding argument is assumed to have type <code>unsigned int *</code> .                                                                                                                                                                                       |
| a <sup>†</sup> , A <sup>†</sup> , e, E, f, F <sup>†</sup> , g, G | Matches a floating-point number; the corresponding argument is assumed to have type <code>float *</code> .                                                                                                                                                                                            |
| c                                                                | Matches <i>n</i> characters, where <i>n</i> is the maximum field width, or one character if no field width is specified. The corresponding argument is assumed to be a pointer to a character array (or a character object, if no field width is specified). Doesn't add a null character at the end. |
| s                                                                | Matches a sequence of non-white-space characters, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.                                                                                                                                  |

<sup>†</sup>C99 only

...scanf Conversion Specifications

| Conversion Specifier | Meaning                                                                                                                                                                                                                                                             |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [                    | Matches a nonempty sequence of characters from a scanset, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.                                                                                        |
| p                    | Matches a pointer value in the form that <code>...printf</code> would have written it. The corresponding argument is assumed to be a pointer to a <code>void *</code> object.                                                                                       |
| n                    | The corresponding argument must point to an object of type <code>int</code> . Stores in this object the number of characters read so far by this call of <code>...scanf</code> . No input is consumed and the return value of <code>...scanf</code> isn't affected. |
| %                    | Matches the character <code>%</code> .                                                                                                                                                                                                                              |

...scanf Conversion Specifications

- Numeric data items can always begin with a sign (+ or -).
- The `o`, `u`, `x`, and `X` specifiers convert the item to unsigned form, however, so they're not normally used to read negative numbers.



## ...scanf Conversion Specifications

- The `[` specifier is a more complicated (and more flexible) version of the `s` specifier.
- A conversion specification using `[` has the form `%[set]` or `%[^set]`, where *set* can be any set of characters.
- `%[set]` matches any sequence of characters in *set* (the *scanset*).
- `%[^set]` matches any sequence of characters not in *set*.
- Examples:
  - `%[abc]` matches any string containing only a, b, and c.
  - `%[^abc]` matches any string that doesn't contain a, b, or c.

## ...scanf Conversion Specifications

- Many of the `...scanf` conversion specifiers are closely related to the numeric conversion functions in `<stdlib.h>`.
- These functions convert strings (like `"-297"`) to their equivalent numeric values (`-297`).
- The `d` specifier, for example, looks for an optional `+` or `-` sign, followed by decimal digits; this is the same form that the `strtol` function requires.

## ...scanf Conversion Specifications

- Correspondence between `...scanf` conversion specifiers and numeric conversion functions:

| <i>Conversion Specifier</i>         | <i>Numeric Conversion Function</i>       |
|-------------------------------------|------------------------------------------|
| <code>d</code>                      | <code>strtol</code> with 10 as the base  |
| <code>i</code>                      | <code>strtol</code> with 0 as the base   |
| <code>o</code>                      | <code>strtoul</code> with 8 as the base  |
| <code>u</code>                      | <code>strtoul</code> with 10 as the base |
| <code>x, X</code>                   | <code>strtoul</code> with 16 as the base |
| <code>a, A, e, E, f, F, g, G</code> | <code>strtod</code>                      |

## C99 Changes to ...scanf Conversion Specifications

- C99 changes to the conversion specifications for `scanf` and `fscanf`:
  - Additional length modifiers
  - Additional conversion specifiers
  - Ability to read infinity and NaN
  - Support for wide characters

scanf Examples

- The next three tables contain sample calls of scanf.
- Characters printed in ~~strikeout~~ are consumed by the call.

scanf Examples

- Examples that combine conversion specifications, white-space characters, and non-white-space characters:

| scanf Call                   | Input                | Variables                     |
|------------------------------|----------------------|-------------------------------|
| n = scanf("%d%d", &i, &j);   | <del>12</del> •,•34□ | n: 1<br>i: 12<br>j: unchanged |
| n = scanf("%d,%d", &i, &j);  | <del>12</del> •,•34□ | n: 1<br>i: 12<br>j: unchanged |
| n = scanf("%d ,%d", &i, &j); | <del>12</del> •,•34□ | n: 2<br>i: 12<br>j: 34        |
| n = scanf("%d, %d", &i, &j); | <del>12</del> •,•34□ | n: 1<br>i: 12<br>j: unchanged |

scanf Examples

- Examples showing the effect of assignment suppression and specifying a field width:

| scanf Call                           | Input                     | Variables                           |
|--------------------------------------|---------------------------|-------------------------------------|
| n = scanf("%*d%d", &i);              | <del>12</del> •34□        | n: 1<br>i: 34                       |
| n = scanf("%*s%s", str);             | <del>My</del> •Fair•Lady□ | n: 1<br>str: "Fair"                 |
| n = scanf("%1d%2d%3d", &i, &j, &k);  | <del>123</del> 45□        | n: 3<br>i: 1<br>j: 23<br>k: 45      |
| n = scanf("%2d%2s%2d", &i, str, &j); | <del>12</del> 3456□       | n: 3<br>i: 12<br>str: "34"<br>j: 56 |

scanf Examples

- Examples that illustrate the i, [, and n conversion specifiers:

| scanf Call                        | Input                    | Variables                       |
|-----------------------------------|--------------------------|---------------------------------|
| n = scanf("%i%i%i", &i, &j, &k);  | <del>12</del> •012•0x12□ | n: 3<br>i: 12<br>j: 10<br>k: 18 |
| n = scanf("%[0123456789]", str);  | <del>123</del> abc□      | n: 1<br>str: "123"              |
| n = scanf("%[0123456789]", str);  | abc123□                  | n: 0<br>str: unchanged          |
| n = scanf("%[^0123456789]", str); | <del>abc</del> 123□      | n: 1<br>str: "abc"              |
| n = scanf("%*d%d%n", &i, &j);     | <del>10</del> •20•30□    | n: 1<br>i: 20<br>j: 5           |

## Detecting End-of-File and Error Conditions

- If we ask a `...scanf` function to read and store  $n$  data items, we expect its return value to be  $n$ .
- If the return value is less than  $n$ , something went wrong:
  - **End-of-file.** The function encountered end-of-file before matching the format string completely.
  - **Read error.** The function was unable to read characters from the stream.
  - **Matching failure.** A data item was in the wrong format.

## Detecting End-of-File and Error Conditions

- Every stream has two indicators associated with it: an **error indicator** and an **end-of-file indicator**.
- These indicators are cleared when the stream is opened.
- Encountering end-of-file sets the end-of-file indicator, and a read error sets the error indicator.
  - The error indicator is also set when a write error occurs on an output stream.
- A matching failure doesn't change either indicator.

## Detecting End-of-File and Error Conditions

- Once the error or end-of-file indicator is set, it remains in that state until it's explicitly cleared, perhaps by a call of the `clearerr` function.
- `clearerr` clears both the end-of-file and error indicators:
 

```
clearerr(fp);
/* clears eof and error indicators for fp */
```
- `clearerr` isn't needed often, since some of the other library functions clear one or both indicators as a side effect.

## Detecting End-of-File and Error Conditions

- The `feof` and `ferror` functions can be used to test a stream's indicators to determine why a prior operation on the stream failed.
- The call `feof ( fp )` returns a nonzero value if the end-of-file indicator is set for the stream associated with `fp`.
- The call `ferror ( fp )` returns a nonzero value if the error indicator is set.

## Detecting End-of-File and Error Conditions

- When `scanf` returns a smaller-than-expected value, `feof` and `ferror` can be used to determine the reason.
  - If `feof` returns a nonzero value, the end of the input file has been reached.
  - If `ferror` returns a nonzero value, a read error occurred during input.
  - If neither returns a nonzero value, a matching failure must have occurred.
- The return value of `scanf` indicates how many data items were read before the problem occurred.

## Detecting End-of-File and Error Conditions

- The `find_int` function is an example that shows how `feof` and `ferror` might be used.
- `find_int` searches a file for a line that begins with an integer:
 

```
n = find_int("foo");
```
- `find_int` returns the value of the integer that it finds or an error code:
  - 1 File can't be opened
  - 2 Read error
  - 3 No line begins with an integer

```
int find_int(const char *filename)
{
 FILE *fp = fopen(filename, "r");
 int n;

 if (fp == NULL)
 return -1; /* can't open file */

 while (fscanf(fp, "%d", &n) != 1) {
 if (ferror(fp)) {
 fclose(fp);
 return -2; /* read error */
 }
 if (feof(fp)) {
 fclose(fp);
 return -3; /* integer not found */
 }
 fscanf(fp, "%*[^\\n]"); /* skips rest of line */
 }

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

## Character I/O

- The next group of library functions can read and write single characters.
- These functions work equally well with text streams and binary streams.
- The functions treat characters as values of type `int`, not `char`.
- One reason is that the input functions indicate an end-of-file (or error) condition by returning EOF, which is a negative integer constant.

## Output Functions

- `putchar` writes one character to the `stdout` stream:  

```
putchar(ch); /* writes ch to stdout */
```
- `fputc` and `putc` write a character to an arbitrary stream:  

```
fputc(ch, fp); /* writes ch to fp */
putc(ch, fp); /* writes ch to fp */
```
- `putc` is usually implemented as a macro (as well as a function), while `fputc` is implemented only as a function.

## Output Functions

- `putchar` itself is usually a macro:  

```
#define putchar(c) putc((c), stdout)
```
- The C standard allows the `putc` macro to evaluate the stream argument more than once, which `fputc` isn't permitted to do.
- Programmers usually prefer `putc`, which gives a faster program.
- If a write error occurs, all three functions set the error indicator for the stream and return EOF.
- Otherwise, they return the character that was written.

## Input Functions

- `getchar` reads a character from `stdin`:  

```
ch = getchar();
```
- `fgetc` and `getc` read a character from an arbitrary stream:  

```
ch = fgetc(fp);
ch = getc(fp);
```
- All three functions treat the character as an unsigned char value (which is then converted to int type before it's returned).
- As a result, they never return a negative value other than EOF.

## Input Functions

- `getc` is usually implemented as a macro (as well as a function), while `fgetc` is implemented only as a function.
- `getchar` is normally a macro as well:  

```
#define getchar() getc(stdin)
```
- Programmers usually prefer `getc` over `fgetc`.

## Input Functions

- The `fgetc`, `getc`, and `getchar` functions behave the same if a problem occurs.
- At end-of-file, they set the stream's end-of-file indicator and return EOF.
- If a read error occurs, they set the stream's error indicator and return EOF.
- To differentiate between the two situations, we can call either `feof` or `ferror`.

## Input Functions

- One of the most common uses of `fgetc`, `getc`, and `getchar` is to read characters from a file.
- A typical while loop for that purpose:  

```
while ((ch = getc(fp)) != EOF) {
 ...
}
```
- Always store the return value in an `int` variable, not a `char` variable.
- Testing a `char` variable against EOF may give the wrong result.

## Input Functions

- The `ungetc` function “pushes back” a character read from a stream and clears the stream's end-of-file indicator.
- A loop that reads a series of digits, stopping at the first nondigit:

```
while (isdigit(ch = getc(fp))) {
 ...
}
ungetc(ch, fp);
/* pushes back last character read */
```

## Input Functions

- The number of characters that can be pushed back by consecutive calls of `ungetc` varies; only the first call is guaranteed to succeed.
- Calling a file-positioning function (`fseek`, `fsetpos`, or `rewind`) causes the pushed-back characters to be lost.
- `ungetc` returns the character it was asked to push back.
  - It returns EOF if an attempt is made to push back EOF or to push back more characters than allowed.

## Program: Copying a File

- The `fcopy.c` program makes a copy of a file.
- The names of the original file and the new file will be specified on the command line when the program is executed.
- An example that uses `fcopy` to copy the file `f1.c` to `f2.c`:  
`fcopy f1.c f2.c`
- `fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

## Program: Copying a File

- Using `"rb"` and `"wb"` as the file modes enables `fcopy` to copy both text and binary files.
- If we used `"r"` and `"w"` instead, the program wouldn't necessarily be able to copy binary files.

### `fcopy.c`

```
/* Copies a file */

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

int main(int argc, char *argv[])
{
 FILE *source_fp, *dest_fp;
 int ch;

 if (argc != 3) {
 fprintf(stderr, "usage: fcopy source dest\n");
 exit(EXIT_FAILURE);
 }
}
```

```
if ((source_fp = fopen(argv[1], "rb")) == NULL) {
 fprintf(stderr, "Can't open %s\n", argv[1]);
 exit(EXIT_FAILURE);
}

if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
 fprintf(stderr, "Can't open %s\n", argv[2]);
 fclose(source_fp);
 exit(EXIT_FAILURE);
}

while ((ch = getc(source_fp)) != EOF)
 putc(ch, dest_fp);

fclose(source_fp);
fclose(dest_fp);
return 0;
}
```

## Line I/O

- Library functions in the next group are able to read and write lines.
- These functions are used mostly with text streams, although it's legal to use them with binary streams as well.

## Output Functions

- The `puts` function writes a string of characters to `stdout`:  

```
puts("Hi, there!"); /* writes to stdout */
```
- After it writes the characters in the string, `puts` always adds a new-line character.

## Output Functions

- `fputs` is a more general version of `puts`.
- Its second argument indicates the stream to which the output should be written:  

```
fputs("Hi, there!", fp); /* writes to fp */
```
- Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.
- Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

## Input Functions

- The `gets` function reads a line of input from `stdin`:  

```
gets(str); /* reads a line from stdin */
```
- `gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).
- `fgets` is a more general version of `gets` that can read from any stream.
- `fgets` is also safer than `gets`, since it limits the number of characters that it will store.



## Input Functions

- A call of `fgets` that reads a line into a character array named `str`:  

```
fgets(str, sizeof(str), fp);
```
- `fgets` will read characters until it reaches the first new-line character or `sizeof(str) - 1` characters have been read.
- If it reads the new-line character, `fgets` stores it along with the other characters.

## Input Functions

- Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters.
- Otherwise, both return their first argument, which points to the array in which the input was stored.
- Both functions store a null character at the end of the string.

## Input Functions

- `fgets` should be used instead of `gets` in most situations.
- `gets` is safe to use only when the string being read is *guaranteed* to fit into the array.
- When there's no guarantee (and there usually isn't), it's much safer to use `fgets`.
- `fgets` will read from the standard input stream if passed `stdin` as its third argument:  

```
fgets(str, sizeof(str), stdin);
```

## Block I/O

- The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step.
- `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

## Block I/O

- `fwrite` is designed to copy an array from memory to a stream.
- Arguments in a call of `fwrite`:
  - Address of array
  - Size of each array element (in bytes)
  - Number of elements to write
  - File pointer
- A call of `fwrite` that writes the entire contents of the array `a`:

```
fwrite(a, sizeof(a[0]),
 sizeof(a) / sizeof(a[0]), fp);
```

## Block I/O

- `fwrite` returns the number of elements actually written.
- This number will be less than the third argument if a write error occurs.

## Block I/O

- `fread` will read the elements of an array from a stream.
- A call of `fread` that reads the contents of a file into the array `a`:
 

```
n = fread(a, sizeof(a[0]),
 sizeof(a) / sizeof(a[0]), fp);
```
- `fread`'s return value indicates the actual number of elements read.
- This number should equal the third argument unless the end of the input file was reached or a read error occurred.

## Block I/O

- `fwrite` is convenient for a program that needs to store data in a file before terminating.
- Later, the program (or another program) can use `fread` to read the data back into memory.
- The data doesn't need to be in array form.
- A call of `fwrite` that writes a structure variable `s` to a file:
 

```
fwrite(&s, sizeof(s), 1, fp);
```

## File Positioning

- Every stream has an associated *file position*.
- When a file is opened, the file position is set at the beginning of the file.
  - In “append” mode, the initial file position may be at the beginning or end, depending on the implementation.
- When a read or write operation is performed, the file position advances automatically, providing sequential access to data.

## File Positioning

- Although sequential access is fine for many applications, some programs need the ability to jump around within a file.
- If a file contains a series of records, we might want to jump directly to a particular record.
- `<stdio.h>` provides five functions that allow a program to determine the current file position or to change it.

## File Positioning

- The `fseek` function changes the file position associated with the first argument (a file pointer).
- The third argument is one of three macros:
 

|                       |                       |
|-----------------------|-----------------------|
| <code>SEEK_SET</code> | Beginning of file     |
| <code>SEEK_CUR</code> | Current file position |
| <code>SEEK_END</code> | End of file           |
- The second argument, which has type `long int`, is a (possibly negative) byte count.

## File Positioning

- Using `fseek` to move to the beginning of a file:
 

```
fseek(fp, 0L, SEEK_SET);
```
- Using `fseek` to move to the end of a file:
 

```
fseek(fp, 0L, SEEK_END);
```
- Using `fseek` to move back 10 bytes:
 

```
fseek(fp, -10L, SEEK_CUR);
```
- If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

## File Positioning

- The file-positioning functions are best used with binary streams.
- C doesn't prohibit programs from using them with text streams, but certain restrictions apply.
- For text streams, `fseek` can be used only to move to the beginning or end of a text stream or to return to a place that was visited previously.
- For binary streams, `fseek` isn't required to support calls in which the third argument is `SEEK_END`.

## File Positioning

- The `ftell` function returns the current file position as a long integer.
- The value returned by `ftell` may be saved and later supplied to a call of `fseek`:

```
long file_pos;
...
file_pos = ftell(fp);
/* saves current position */
...
fseek(fp, file_pos, SEEK_SET);
/* returns to old position */
```

## File Positioning

- If `fp` is a binary stream, the call `ftell(fp)` returns the current file position as a byte count, where zero represents the beginning of the file.
- If `fp` is a text stream, `ftell(fp)` isn't necessarily a byte count.
- As a result, it's best not to perform arithmetic on values returned by `ftell`.

## File Positioning

- The `rewind` function sets the file position at the beginning.
- The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`.
  - The difference? `rewind` doesn't return a value but does clear the error indicator for `fp`.

## File Positioning

- `fseek` and `ftell` are limited to files whose positions can be stored in a long integer.
- For working with very large files, C provides two additional functions: `fgetpos` and `fsetpos`.
- These functions can handle large files because they use values of type `fpos_t` to represent file positions.
  - An `fpos_t` value isn't necessarily an integer; it could be a structure, for instance.

## File Positioning

- The call `fgetpos(fp, &file_pos)` stores the file position associated with `fp` in the `file_pos` variable.
- The call `fsetpos(fp, &file_pos)` sets the file position for `fp` to be the value stored in `file_pos`.
- If a call of `fgetpos` or `fsetpos` fails, it stores an error code in `errno`.
- Both functions return zero when they succeed and a nonzero value when they fail.

## File Positioning

- An example that uses `fgetpos` and `fsetpos` to save a file position and return to it later:

```
fpos_t file_pos;
...
fgetpos(fp, &file_pos);
 /* saves current position */
...
fsetpos(fp, &file_pos);
 /* returns to old position */
```

## Program: Modifying a File of Part Records

- Actions performed by the `invclear.c` program:
  - Opens a binary file containing part structures.
  - Reads the structures into an array.
  - Sets the `on_hand` member of each structure to 0.
  - Writes the structures back to the file.
- The program opens the file in "`rb+`" mode, allowing both reading and writing.

**invclear.c**

```
/* Modifies a file of part records by setting the quantity
 on hand to zero for all records */
```

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

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
} inventory[MAX_PARTS];

int num_parts;
```

```
int main(void)
{
 FILE *fp;
 int i;

 if ((fp = fopen("inventory.dat", "rb+")) == NULL) {
 fprintf(stderr, "Can't open inventory file\n");
 exit(EXIT_FAILURE);
 }

 num_parts = fread(inventory, sizeof(struct part),
 MAX_PARTS, fp);

 for (i = 0; i < num_parts; i++)
 inventory[i].on_hand = 0;

 rewind(fp);
 fwrite(inventory, sizeof(struct part), num_parts, fp);
 fclose(fp);

 return 0;
}
```

**String I/O**

- The functions described in this section can read and write data using a string as though it were a stream.
- `sprintf` and `snprintf` write characters into a string.
- `sscanf` reads characters from a string.

**String I/O**

- Three similar functions (`vsprintf`, `vsnprintf`, and `vsscanf`) also belong to `<stdio.h>`.
- These functions rely on the `va_list` type, which is declared in `<stdarg.h>`, so they are discussed in Chapter 26.

## Output Functions

- The `sprintf` function writes output into a character array (pointed to by its first argument) instead of a stream.
- A call that writes "9/20/2010" into `date`:  

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```
- `sprintf` adds a null character at the end of the string.
- It returns the number of characters stored (not counting the null character).

## Output Functions

- `sprintf` can be used to format data, with the result saved in a string until it's time to produce output.
- `sprintf` is also convenient for converting numbers to character form.

## Output Functions

- The `snprintf` function (new in C99) is the same as `sprintf`, except for an additional second parameter named `n`.
- No more than `n - 1` characters will be written to the string, not counting the terminating null character, which is always written unless `n` is zero.
- Example:  

```
snprintf(name, 13, "%s, %s", "Einstein", "Albert");
```

 The string "Einstein, Al" is written into `name`.

## Output Functions

- `snprintf` returns the number of characters that would have been written (not including the null character) had there been no length restriction.
- If an encoding error occurs, `snprintf` returns a negative number.
- To see if `snprintf` had room to write all the requested characters, we can test whether its return value was nonnegative and less than `n`.

## Input Functions

- The `sscanf` function is similar to `scanf` and `fscanf`.
- `sscanf` reads from a string (pointed to by its first argument) instead of reading from a stream.
- `sscanf`'s second argument is a format string identical to that used by `scanf` and `fscanf`.

## Input Functions

- `sscanf` is handy for extracting data from a string that was read by another input function.
- An example that uses `fgets` to obtain a line of input, then passes the line to `sscanf` for further processing:

```
fgets(str, sizeof(str), stdin);
/* reads a line of input */
sscanf(str, "%d%d", &i, &j);
/* extracts two integers */
```

## Input Functions

- One advantage of using `sscanf` is that we can examine an input line as many times as needed.
- This makes it easier to recognize alternate input forms and to recover from errors.
- Consider the problem of reading a date that's written either in the form *month / day / year* or *month-day-year*:

```
if (sscanf(str, "%d /%d /%d", &month, &day, &year) == 3)
 printf("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf(str, "%d-%d-%d", &month, &day, &year) == 3)
 printf("Month: %d, day: %d, year: %d\n", month, day, year);
else
 printf("Date not in the proper form\n");
```

## Input Functions

- Like the `scanf` and `fscanf` functions, `sscanf` returns the number of data items successfully read and stored.
- `sscanf` returns EOF if it reaches the end of the string (marked by a null character) before finding the first item.