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.
Chapter 22: Input/Output

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
Chapter 22: Input/Output

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:

<table>
<thead>
<tr>
<th>File Pointer</th>
<th>Stream</th>
<th>Default Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>Standard input</td>
<td>Keyboard</td>
</tr>
<tr>
<td>stdout</td>
<td>Standard output</td>
<td>Screen</td>
</tr>
<tr>
<td>stderr</td>
<td>Standard error</td>
<td>Screen</td>
</tr>
</tbody>
</table>

• 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.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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 (’\x0d’) followed by line-feed character (’\x0a’) 
  – 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.
Chapter 22: Input/Output

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'
```
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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`:

```c
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 \texttt{fopen} includes the \ character.

• The call

\begin{verbatim}
fopen("c:\project\test1.dat", "r")
\end{verbatim}

will fail, because \texttt{\t} is treated as a character escape.

• One way to avoid the problem is to use \texttt{\\} instead of \texttt{\}:

\begin{verbatim}
fopen("c:\\project\\test1.dat", "r")
\end{verbatim}

• An alternative is to use the / character instead of \texttt{\}:

\begin{verbatim}
fopen("c:/project/test1.dat", "r")
\end{verbatim}
Opening a File

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

  ```c
  fp = fopen("in.dat", "r"); /* opens in.dat for reading */
  ```

- When it can’t open a file, `fopen` returns a null pointer.
Chapter 22: Input/Output

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:

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;r&quot;</td>
<td>Open for reading</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>Open for writing (file need not exist)</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>Open for appending (file need not exist)</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>Open for reading and writing, starting at beginning</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>Open for reading and writing (truncate if file exists)</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>Open for reading and writing (append if file exists)</td>
</tr>
</tbody>
</table>
Modes

- Mode strings for binary files:

<table>
<thead>
<tr>
<th>String</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;rb&quot;</td>
<td>Open for reading</td>
</tr>
<tr>
<td>&quot;wb&quot;</td>
<td>Open for writing (file need not exist)</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>Open for appending (file need not exist)</td>
</tr>
<tr>
<td>&quot;r+b&quot; or &quot;rb+&quot;</td>
<td>Open for reading and writing, starting at beginning</td>
</tr>
<tr>
<td>&quot;w+b&quot; or &quot;wb+&quot;</td>
<td>Open for reading and writing (truncate if file exists)</td>
</tr>
<tr>
<td>&quot;a+b&quot; or &quot;ab+&quot;</td>
<td>Open for reading and writing (append if file exists)</td>
</tr>
</tbody>
</table>
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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:

```c
#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;
}
```
• 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`:

```c
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
Chapter 13 showed how to access command-line arguments by defining `main` as a function with two parameters:

```c
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:

```
0: program name
1: names.dat
2: dates.dat
```
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.
Chapter 22: Input/Output

**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:

```c
FILE *tempptr;
...
tempptr = tmpfile();
/* creates a temporary file */
```

- If it fails to create a file, `tmpfile` returns a null pointer.
Chapter 22: Input/Output

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:

```c
char *filename;
...
filename = tmpnam(NULL);
/* creates a temporary file name */
```
Chapter 22: Input/Output

Temporary Files

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

```c
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.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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`:
  ```c
  fflush(fp);   /* flushes buffer for fp */
  ```

• A call that flushes all output streams:
  ```c
  fflush(NULL);  /* flushes all buffers */
  ```

• `fflush` returns zero if it’s successful and `EOF` if an error occurs.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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:

```c
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
  
  ```c
  (void) setvbuf(stream, NULL, _IOMBF, 0);
  ```

• Otherwise, it’s equivalent to
  
  ```c
  (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.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

Miscellaneous File Operations

• `rename` changes the name of a file:
  ```c
  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.
Chapter 22: Input/Output

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:

  ```c
  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:

  ```c
  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.
Chapter 22: Input/Output

The ...printf Functions

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>Left-justify within field.</td>
</tr>
<tr>
<td>+</td>
<td>Numbers produced by signed conversions always begin with + or −.</td>
</tr>
<tr>
<td>space</td>
<td>Nonnegative numbers produced by signed conversions are preceded by a space.</td>
</tr>
<tr>
<td>#</td>
<td>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 g or G conversions.</td>
</tr>
<tr>
<td>0</td>
<td>Numbers are padded with leading zeros up to the field width.</td>
</tr>
</tbody>
</table>

(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.
...\texttt{printf} Conversion Specifications

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

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

†C99 only
## \texttt{printf} Conversion Specifications

<table>
<thead>
<tr>
<th>Length Modifier</th>
<th>Conversion Specifiers</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ll}†</td>
<td>\texttt{d, i, o, u, x, X}</td>
<td>long long int,</td>
</tr>
<tr>
<td>\texttt{(ell-ell)}</td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td></td>
<td>long long int *</td>
</tr>
<tr>
<td>\texttt{j}†</td>
<td>\texttt{d, i, o, u, x, X}</td>
<td>intmax_t, uintmax_t</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td></td>
<td>intmax_t *</td>
</tr>
<tr>
<td>\texttt{z}†</td>
<td>\texttt{d, i, o, u, x, X}</td>
<td>size_t</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td></td>
<td>size_t *</td>
</tr>
<tr>
<td>\texttt{t}†</td>
<td>\texttt{d, i, o, u, x, X}</td>
<td>ptrdiff_t</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td></td>
<td>ptrdiff_t *</td>
</tr>
<tr>
<td>\texttt{L}</td>
<td>\texttt{a, A, e, E, f, F, g, G}</td>
<td>long double</td>
</tr>
</tbody>
</table>

†C99 only
...printf Conversion Specifications

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

<table>
<thead>
<tr>
<th>Conversion Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>Converts an int value to decimal form.</td>
</tr>
<tr>
<td>o, u, x, X</td>
<td>Converts an unsigned int 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.</td>
</tr>
<tr>
<td>f, F†</td>
<td>Converts a double value to decimal form, putting the decimal point in the correct position. If no precision is specified, displays six digits after the decimal point.</td>
</tr>
</tbody>
</table>

†C99 only
## Conversion Specifications

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

†C99 only
## ...printf Conversion Specifications

<table>
<thead>
<tr>
<th>Conversion Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Displays an int value as an unsigned character.</td>
</tr>
<tr>
<td>s</td>
<td>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.</td>
</tr>
<tr>
<td>p</td>
<td>Converts a void * value to printable form.</td>
</tr>
<tr>
<td>n</td>
<td>The corresponding argument must point to an object of type int. Stores in this object the number of characters written so far by this call of ...printf; produces no output.</td>
</tr>
<tr>
<td>%</td>
<td>Writes the character %.</td>
</tr>
</tbody>
</table>
C99 Changes to \textit{printf} Conversion Specifications

- C99 changes to the conversion specifications for \texttt{printf} and \texttt{fprintf}:
  - Additional length modifiers
  - Additional conversion specifiers
  - Ability to write infinity and NaN
  - Support for wide characters
  - Previously undefined conversion specifications now allowed
Chapter 22: Input/Output

Examples of ...printf Conversion Specifications

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

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Result of Applying Conversion to 123</th>
<th>Result of Applying Conversion to –123</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%8d</code></td>
<td>•••••123</td>
<td>•••••–123</td>
</tr>
<tr>
<td><code>%8d</code></td>
<td>123•••••</td>
<td>–123•••••</td>
</tr>
<tr>
<td><code>%+8d</code></td>
<td>••••+123</td>
<td>••••–123</td>
</tr>
<tr>
<td><code>%8d</code></td>
<td>••••123</td>
<td>••••–123</td>
</tr>
<tr>
<td><code>%08d</code></td>
<td>00000123</td>
<td>–0000123</td>
</tr>
<tr>
<td><code>%+08d</code></td>
<td>+0000123</td>
<td>–0000123</td>
</tr>
<tr>
<td><code>%08d</code></td>
<td>0000123</td>
<td>–0000123</td>
</tr>
</tbody>
</table>
Chapter 22: Input/Output

Examples of \texttt{printf} Conversion Specifications

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

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Result of Applying Conversion to 123</th>
<th>Result of Applying Conversion to 123.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%8o</td>
<td>\ldots\ldots173</td>
<td></td>
</tr>
<tr>
<td>%#8o</td>
<td>\ldots\ldots0173</td>
<td></td>
</tr>
<tr>
<td>%8x</td>
<td>\ldots\ldots7b</td>
<td></td>
</tr>
<tr>
<td>%#8x</td>
<td>\ldots\ldots0x7b</td>
<td></td>
</tr>
<tr>
<td>%8X</td>
<td>\ldots\ldots7B</td>
<td></td>
</tr>
<tr>
<td>%#8X</td>
<td>\ldots\ldots0X7B</td>
<td></td>
</tr>
<tr>
<td>%8g</td>
<td>\ldots\ldots123</td>
<td>\ldots\ldots123</td>
</tr>
<tr>
<td>%#8g</td>
<td>\ldots\ldots123.000</td>
<td>\ldots\ldots123</td>
</tr>
<tr>
<td>%8G</td>
<td>\ldots\ldots123</td>
<td>\ldots\ldots123.000</td>
</tr>
<tr>
<td>%#8G</td>
<td>\ldots\ldots123</td>
<td>\ldots\ldots123.000</td>
</tr>
</tbody>
</table>
Examples of `printf` Conversion Specifications

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

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Result of Applying Conversion to &quot;bogus&quot;</th>
<th>Result of Applying Conversion to &quot;buzzword&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>%6s</td>
<td>•bogus</td>
<td>buzzword</td>
</tr>
<tr>
<td>%-6s</td>
<td>bogus•</td>
<td>buzzword</td>
</tr>
<tr>
<td>%.4s</td>
<td>bogu</td>
<td>buzz</td>
</tr>
<tr>
<td>%6.4s</td>
<td>••bogu</td>
<td>••buzz</td>
</tr>
<tr>
<td>%-6.4s</td>
<td>bogu••</td>
<td>buzz••</td>
</tr>
</tbody>
</table>
Examples of \texttt{printf} Conversion Specifications

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

<table>
<thead>
<tr>
<th>Number</th>
<th>Result of Applying % . 4g Conversion to Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456.</td>
<td>1.235e+05</td>
</tr>
<tr>
<td>12345.6</td>
<td>1.235e+04</td>
</tr>
<tr>
<td>1234.56</td>
<td>1235</td>
</tr>
<tr>
<td>123.456</td>
<td>123.5</td>
</tr>
<tr>
<td>12.3456</td>
<td>12.35</td>
</tr>
<tr>
<td>1.23456</td>
<td>1.235</td>
</tr>
<tr>
<td>.123456</td>
<td>0.1235</td>
</tr>
<tr>
<td>.0123456</td>
<td>0.01235</td>
</tr>
<tr>
<td>.00123456</td>
<td>0.001235</td>
</tr>
<tr>
<td>.000123456</td>
<td>0.0001235</td>
</tr>
<tr>
<td>.0000123456</td>
<td>1.235e−05</td>
</tr>
<tr>
<td>.00000123456</td>
<td>1.235e−06</td>
</tr>
</tbody>
</table>

Copyright © 2008 W. W. Norton & Company. All rights reserved.
Examples of \texttt{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 \textit{after} the format string.
- Calls of \texttt{printf} that produce the same output:

```c
printf("\%6.4d", i);
printf("\%*.4d", 6, i);
printf("\%6.*d", 4, i);
printf("\%*.d", 6, 4, i);
```
Examples of \textit{printf} Conversion Specifications

- A major advantage of \* is that it allows us to use a macro to specify the width or precision:
  
  \begin{verbatim}
  printf("%*d", WIDTH, i);
  \end{verbatim}

- The width or precision can even be computed during program execution:

  \begin{verbatim}
  printf("%*d", page_width / num_cols, i);
  \end{verbatim}
Chapter 22: Input/Output

Examples of ... `printf`
Conversion Specifications

- The `%p` conversion is used to print the value of a pointer:

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

- The pointer is likely to be shown as an octal or hexadecimal number.
Examples of \texttt{printf} Conversion Specifications

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

\begin{verbatim}
printf("%d%n\n", 123, &len);
\end{verbatim}
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 \texttt{...scanf} Functions

- \texttt{scanf} always reads from \texttt{stdin}, whereas \texttt{fscanf} reads from the stream indicated by its first argument:

  \begin{verbatim}
  scanf("\%d\%d", &i, &j);
  /* reads from stdin */
  fscanf(fp, "\%d\%d", &i, &j);
  /* reads from fp */
  \end{verbatim}

- A call of \texttt{scanf} is equivalent to a call of \texttt{fscanf} with \texttt{stdin} as the first argument.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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:

```c
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
Chapter 22: Input/Output

...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

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

†C99 only
...scanf Conversion Specifications

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

<table>
<thead>
<tr>
<th>Conversion Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Matches a decimal integer; the corresponding argument is assumed to have type <code>int *</code>.</td>
</tr>
<tr>
<td>i</td>
<td>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).</td>
</tr>
<tr>
<td>o</td>
<td>Matches an octal integer; the corresponding argument is assumed to have type <code>unsigned int *</code>.</td>
</tr>
<tr>
<td>u</td>
<td>Matches a decimal integer; the corresponding argument is assumed to have type <code>unsigned int *</code>.</td>
</tr>
</tbody>
</table>
...scanf Conversion Specifications

<table>
<thead>
<tr>
<th>Conversion Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, X</td>
<td>Matches a hexadecimal integer; the corresponding argument is assumed to have type unsigned int*.</td>
</tr>
<tr>
<td>a†, A†, e, E, f, F†, g, G</td>
<td>Matches a floating-point number; the corresponding argument is assumed to have type float*.</td>
</tr>
<tr>
<td>c</td>
<td>Matches $n$ characters, where $n$ 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.</td>
</tr>
<tr>
<td>s</td>
<td>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.</td>
</tr>
</tbody>
</table>

†C99 only
## …scanf Conversion Specifications

<table>
<thead>
<tr>
<th>Conversion Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
<td>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.</td>
</tr>
<tr>
<td>p</td>
<td>Matches a pointer value in the form that …printf would have written it. The corresponding argument is assumed to be a pointer to a <code>void *</code> object.</td>
</tr>
<tr>
<td>n</td>
<td>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 …scanf. No input is consumed and the return value of …scanf isn’t affected.</td>
</tr>
<tr>
<td>%</td>
<td>Matches the character <code>%</code>.</td>
</tr>
</tbody>
</table>
...`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.
...\texttt{scanf} Conversion Specifications

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

<table>
<thead>
<tr>
<th>Conversion Specifier</th>
<th>Numeric Conversion Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td><code>strtol</code> with 10 as the base</td>
</tr>
<tr>
<td>i</td>
<td><code>strtol</code> with 0 as the base</td>
</tr>
<tr>
<td>o</td>
<td><code>strtoul</code> with 8 as the base</td>
</tr>
<tr>
<td>u</td>
<td><code>strtoul</code> with 10 as the base</td>
</tr>
<tr>
<td>x, X</td>
<td><code>strtoul</code> with 16 as the base</td>
</tr>
<tr>
<td>a, A, e, E, f, F, g, G</td>
<td><code>strtod</code></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>scanf Call</th>
<th>Input</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n = scanf(&quot;%d%d&quot;, &amp;i, &amp;j);</code></td>
<td><code>12 </code>, , <code>34</code></td>
<td><code>n: 1</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;%d,%d&quot;, &amp;i, &amp;j);</code></td>
<td><code>12 </code>, , <code>34</code></td>
<td><code>n: 1</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;%d ,%d&quot;, &amp;i, &amp;j);</code></td>
<td><code>12 -, ,34</code></td>
<td><code>n: 2</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;%d ,%d&quot;, &amp;i, &amp;j);</code></td>
<td><code>12 -, ,34</code></td>
<td><code>n: 1</code></td>
</tr>
</tbody>
</table>
Chapter 22: Input/Output

scanf Examples

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

```
scanf Call                  Input             Variables
n = scanf("%*d%d", &i);    12•34             n: 1
                           i: 34
n = scanf("%*s%s", str);  My•Fair•Lady      n: 1
                           str: "Fair"
```

```
n = scanf("%1d%2d%3d", &i, &j, &k); 12345     n: 3
                                     i: 1
                                     j: 23
                                     k: 45
```

```
n = scanf("%2d%2s%2d", &i, str, &j); 123456   n: 3
                                       i: 12
                                       str: "34"
                                       j: 56
```
### `scanf` Examples

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

<table>
<thead>
<tr>
<th><code>scanf</code> Call</th>
<th>Input</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n = scanf(&quot;%i%i%i&quot;, &amp;i, &amp;j, &amp;k);</code></td>
<td>12-012-0x12☆</td>
<td><code>n: 3</code></td>
</tr>
<tr>
<td><code>i: 12</code></td>
<td><code>j: 10</code></td>
<td><code>k: 18</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;[%0123456789]&quot;, str);</code></td>
<td>123abc☆</td>
<td><code>n: 1</code></td>
</tr>
<tr>
<td><code>str: &quot;123&quot;</code></td>
<td><code>n: 0</code></td>
<td><code>str: unchanged</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;[%0123456789]&quot;, str);</code></td>
<td>abc123☆</td>
<td><code>str: &quot;abc&quot;</code></td>
</tr>
<tr>
<td><code>n: 1</code></td>
<td><code>str: &quot;abc&quot;</code></td>
<td><code>n: 1</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;[^0123456789]&quot;, str);</code></td>
<td>abc123☆</td>
<td><code>str: &quot;abc&quot;</code></td>
</tr>
<tr>
<td><code>n: 1</code></td>
<td><code>str: &quot;abc&quot;</code></td>
<td><code>n: 1</code></td>
</tr>
<tr>
<td><code>n = scanf(&quot;%*d%d%n&quot;, &amp;i, &amp;j);</code></td>
<td>10-20-30☆</td>
<td><code>n: 1</code></td>
</tr>
<tr>
<td><code>i: 20</code></td>
<td><code>j: 5</code></td>
<td></td>
</tr>
</tbody>
</table>
Detecting End-of-File and Error Conditions

- If we ask a \( \text{scanf} \) 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 \textit{error indicator} and an \textit{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:

```c
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
Chapter 22: Input/Output

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:

  ```c
  putchar(ch);  /* writes ch to stdout */
  ```

- **fputc** and **putc** write a character to an arbitrary stream:

  ```c
  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:
  ```c
  #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`:
  \[
  \text{ch} = \text{getchar}();
  \]

• `fgetc` and `getc` read a character from an arbitrary stream:
  \[
  \text{ch} = \text{fgetc}(\text{fp});
  \]
  \[
  \text{ch} = \text{getc}(\text{fp});
  \]

• 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:
  ```c
  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.
Chapter 22: Input/Output

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:

```c
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.
Chapter 22: Input/Output

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.
```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);
    }
```
Chapter 22: Input/Output

```c
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`:
  ```c
  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:
  
  ```c
  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`:
  ```c
  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 newline 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`:
  \[
  \text{fgets(str, sizeof(str), fp)};
  \]
• `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:
  ```c
  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.
Chapter 22: Input/Output

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`:
  ```c
  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.
Chapter 22: Input/Output

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`:
  
  ```c
  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:
  ```c
  fwrite(&s, sizeof(s), 1, fp);
  ```
Chapter 22: Input/Output

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:
  SEEK_SET    Beginning of file
  SEEK_CUR    Current file position
  SEEK_END    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`. 
Chapter 22: Input/Output

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`:

```c
long file_pos;
...
file_pos = ftell(fp);
    /* saves current position */
...
    fseek(fp, file_pos, SEEK_SET);
    /* returns to old position */
```
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

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.
Chapter 22: Input/Output

File Positioning

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

  ```c
  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.
Chapter 22: Input/Output

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`:
  ```c
  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.
Chapter 22: Input/Output

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:

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

  The string "Einstein, Al" is written into `name`. 
Chapter 22: Input/Output

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:

```c
fgets(str, sizeof(str), stdin); /* reads a line of input */
sscanf(str, "%d%d", &i, &j); /* extracts two integers */
```
Chapter 22: Input/Output

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`:

```c
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.