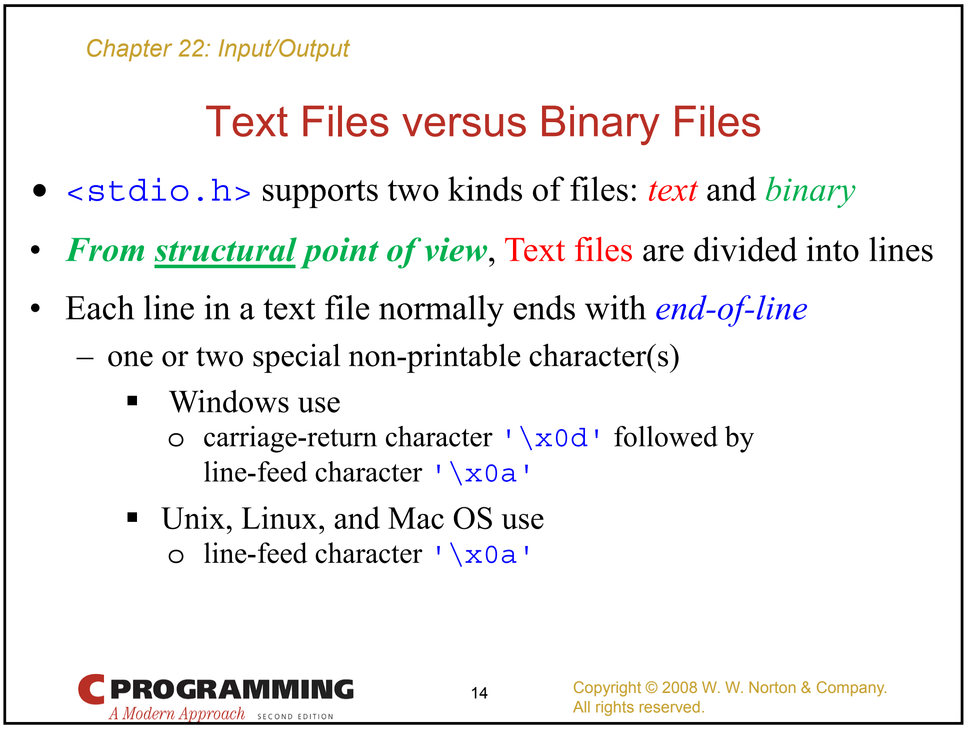
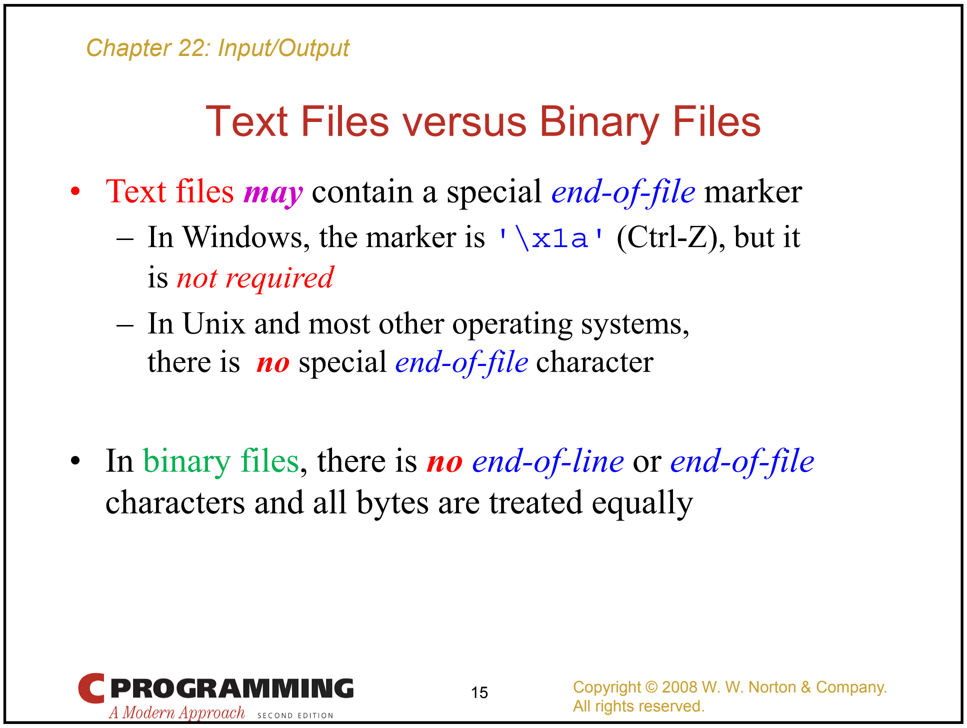


Text Files versus Binary Files

- `<stdio.h>` supports two kinds of files: *text* and *binary*
- ***From <u>structural</u> point of view***, Text files are divided into lines
- Each line in a text file normally ends with *end-of-line*
  - one or two special non-printable character(s)
    - Windows use
      - carriage-return character `'\x0d'` followed by line-feed character `'\x0a'`
    - Unix, Linux, and Mac OS use
      - line-feed character `'\x0a'`



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*
  - In Unix and most other operating systems, there is ***no*** special *end-of-file* character
- In binary files, there is ***no*** *end-of-line* or *end-of-file* characters and all bytes are treated equally

Text Files versus Binary Files

- When writing a program that reads from or write to a file, we need to take into account whether it is a text file or binary file
(i.e., how we will deal with `'\x0a'`, `'\x0d'`*, and* `'\x1a'`*)*
  - A program that displays the contents of a file on the screen will probably assume it is a text file
  - A file copying program on the other hand can not assume that the file to be copied is a text file
  - When we cannot say for sure whether a file is text or binary, it is safer to assume that it is binary

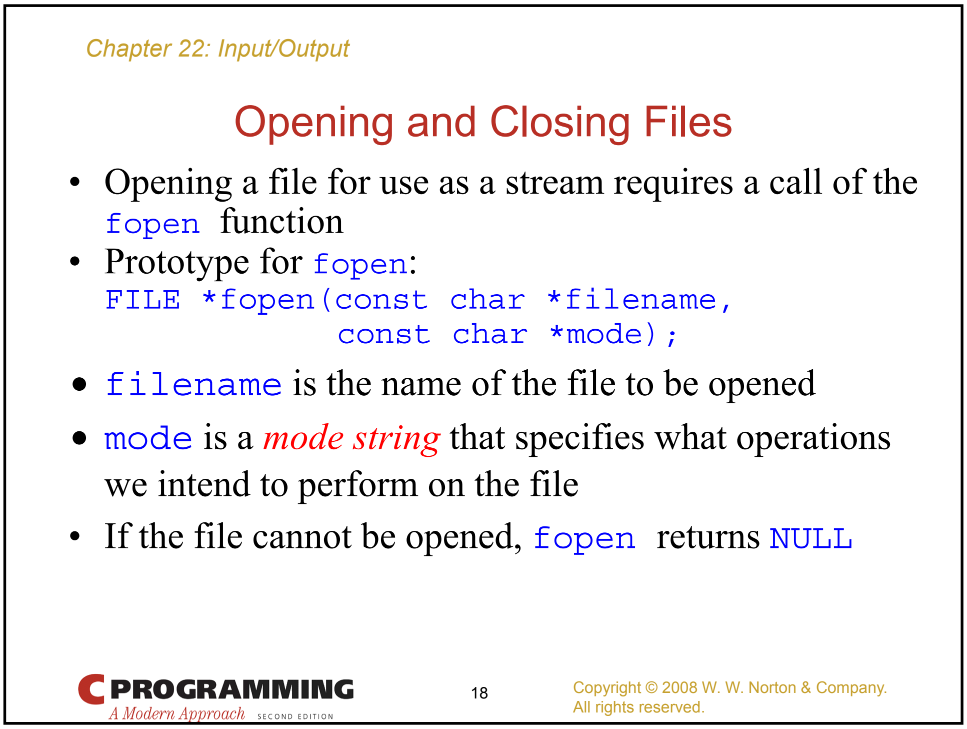
Text Files versus Binary Files

- When data is written to a file, it can be stored in *text form* or in *binary form*
- ***From encoding point of view***, In *text format*, data is represented as *characters*
For example, 32767 will be represented as:

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

- In *binary format*, data is ***not*** necessarily represented as *characters*; For example, 32767 will be represented as:

```
01111111 11111111
```

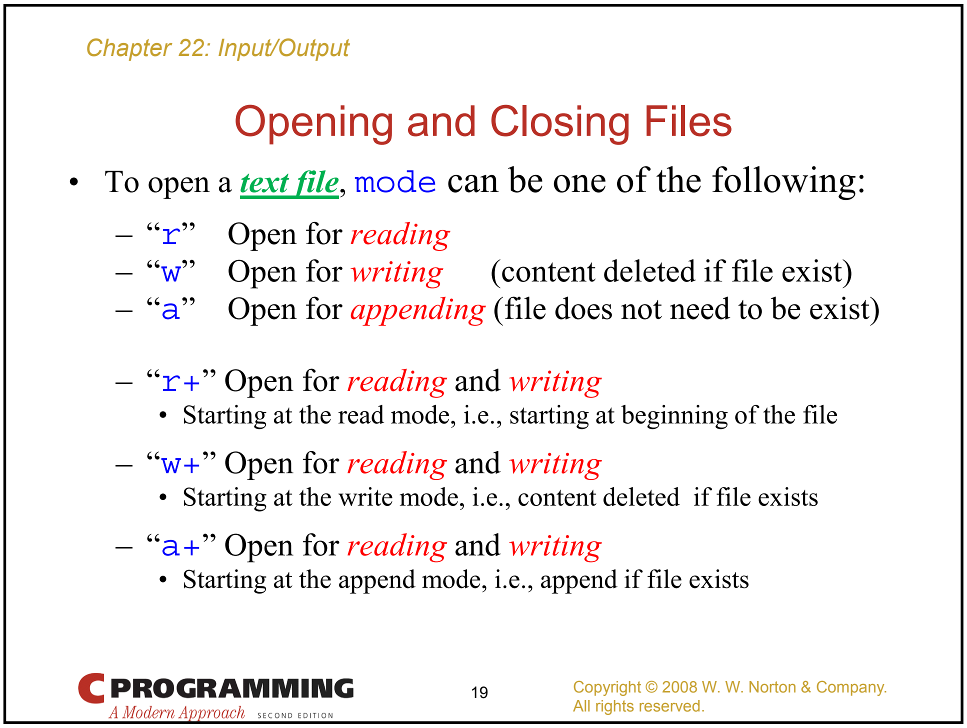


Opening and Closing Files

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

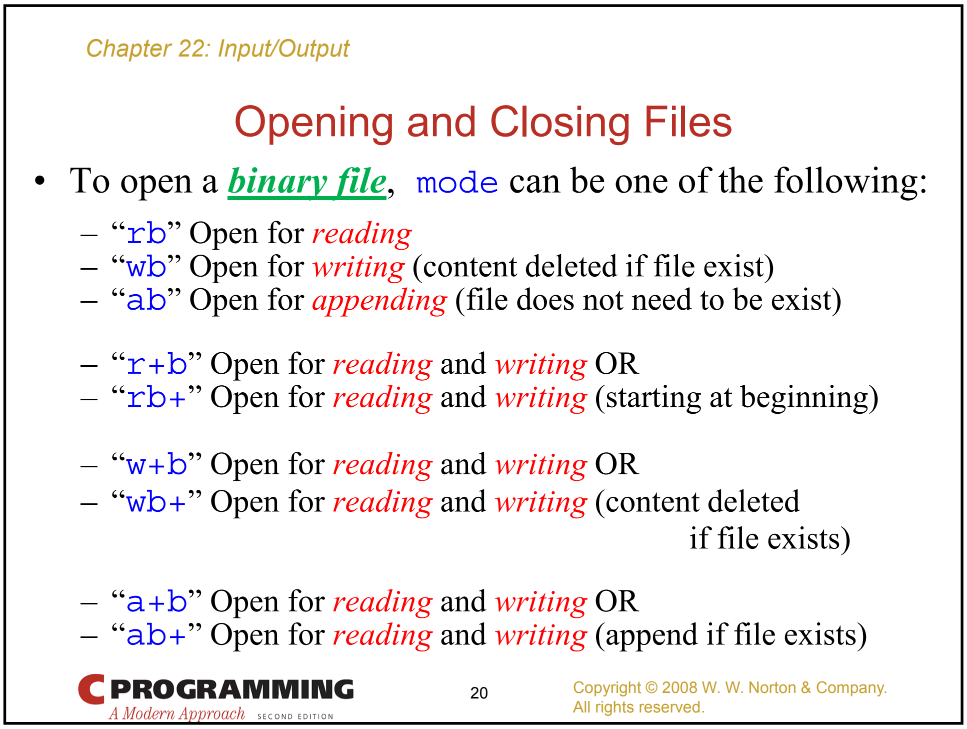
```
FILE *fopen(const char *filename,
            const char *mode);
```

- `filename` is the name of the file to be opened
- `mode` is a *mode string* that specifies what operations we intend to perform on the file
- If the file cannot be opened, `fopen` returns `NULL`



Opening and Closing Files

- To open a ***text file***, `mode` can be one of the following:
  - “`r`” Open for *reading*
  - “`w`” Open for *writing* (content deleted if file exist)
  - “`a`” Open for *appending* (file does not need to be exist)
  - “`r+`” Open for *reading* and *writing*
    - Starting at the read mode, i.e., starting at beginning of the file
  - “`w+`” Open for *reading* and *writing*
    - Starting at the write mode, i.e., content deleted if file exists
  - “`a+`” Open for *reading* and *writing*
    - Starting at the append mode, i.e., append if file exists



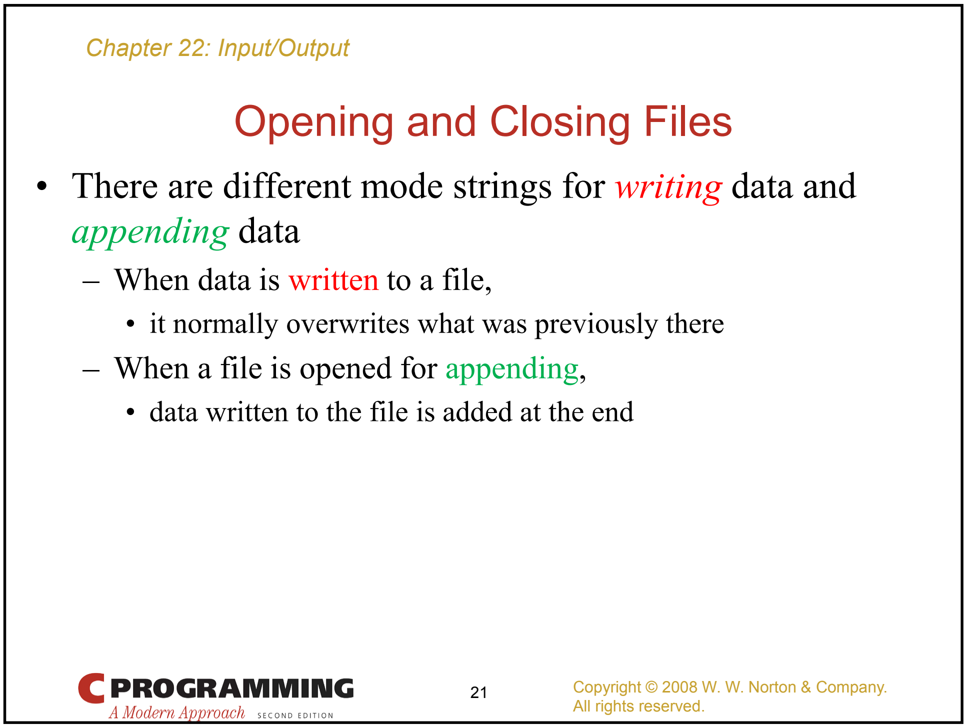
Opening and Closing Files

- To open a ***binary file***, `mode` can be one of the following:
  - “`rb`” Open for *reading*
  - “`wb`” Open for *writing* (content deleted if file exist)
  - “`ab`” Open for *appending* (file does not need to be exist)

  - “`r+b`” Open for *reading* and *writing* OR
  - “`rb+`” Open for *reading* and *writing* (starting at beginning)

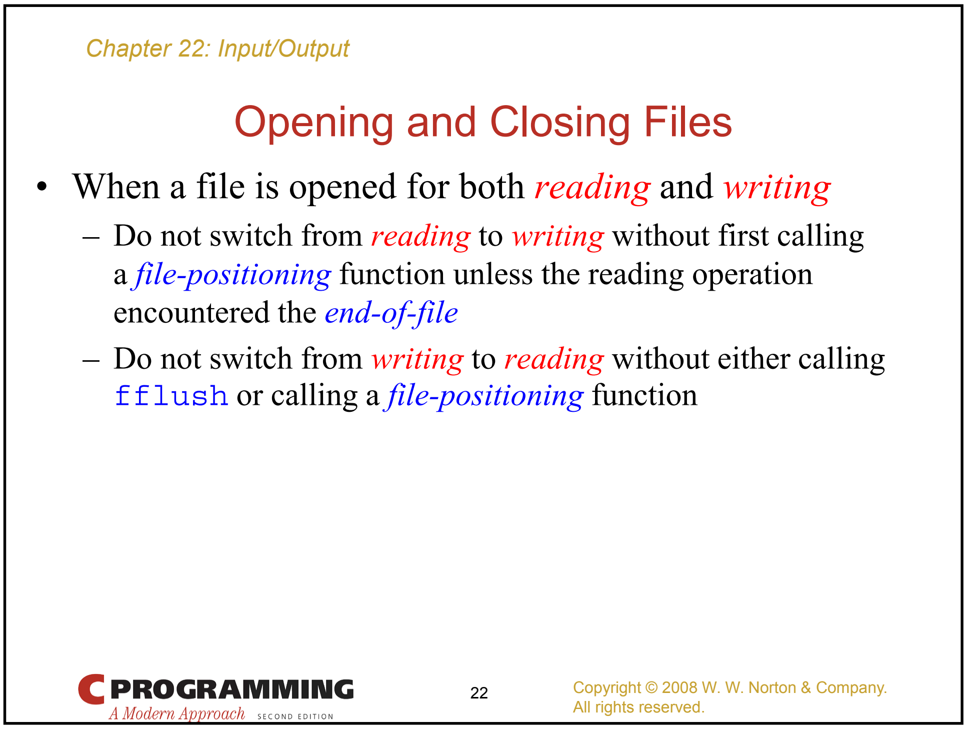
  - “`w+b`” Open for *reading* and *writing* OR
  - “`wb+`” Open for *reading* and *writing* (content deleted if file exists)

  - “`a+b`” Open for *reading* and *writing* OR
  - “`ab+`” Open for *reading* and *writing* (append if file exists)



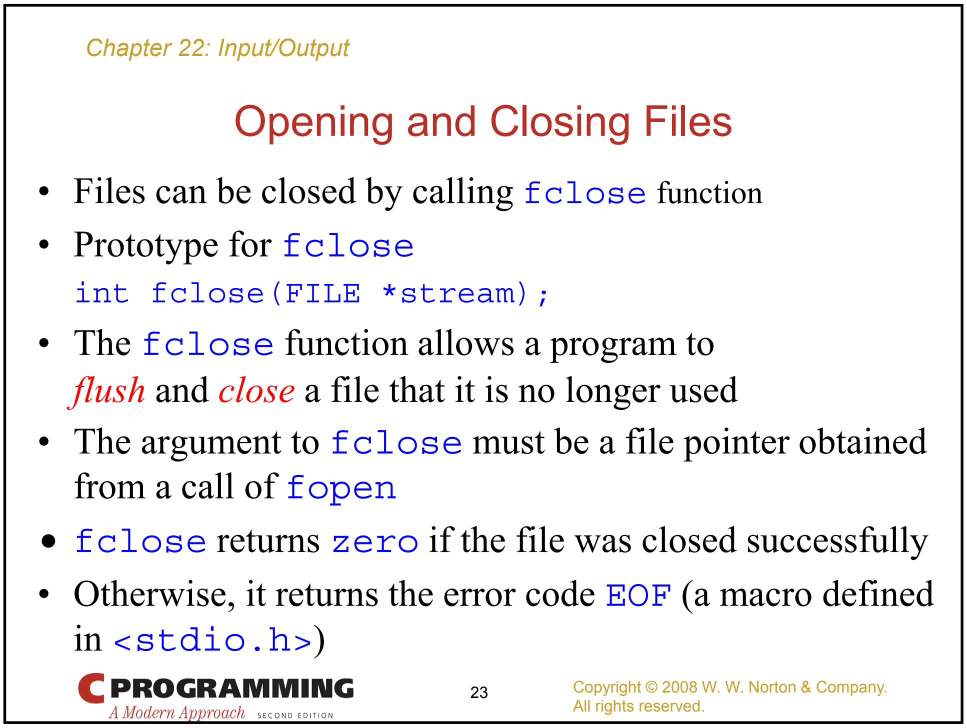
Opening and Closing Files

- 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



Opening and Closing Files

- When a file is opened for both *reading* and *writing*
  - Do not switch from *reading* to *writing* without first calling a *file-positioning* function unless the reading operation encountered the *end-of-file*
  - Do not switch from *writing* to *reading* without either calling `fflush` or calling a *file-positioning* function

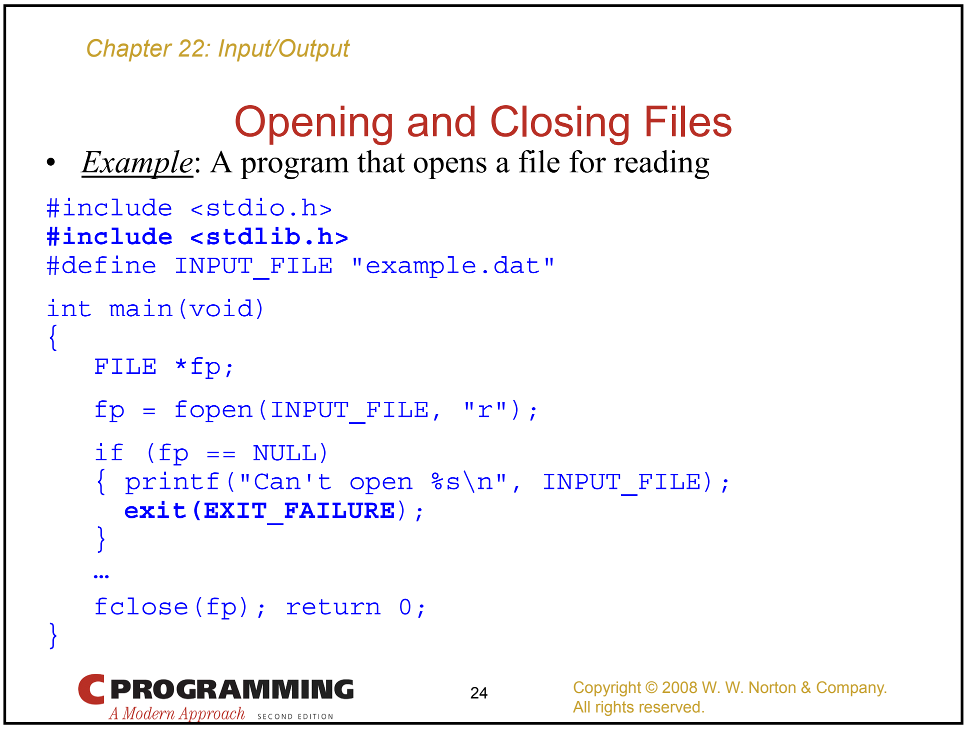


Opening and Closing Files

- Files can be closed by calling `fclose` function
- Prototype for `fclose`

```
int fclose(FILE *stream);
```

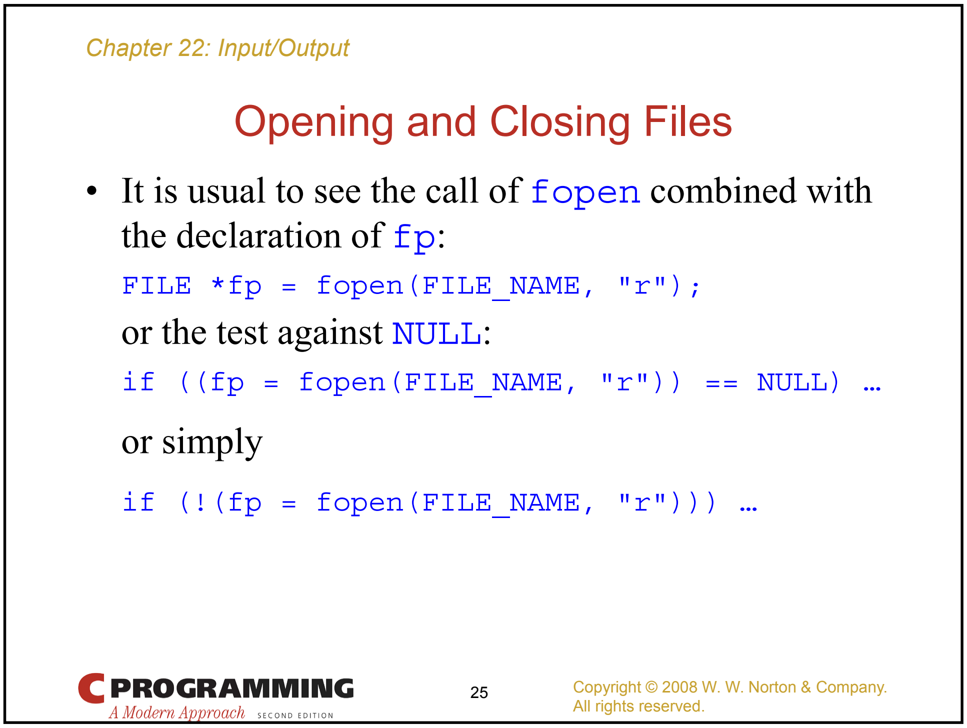
- The `fclose` function allows a program to *flush* and *close* a file that it is no longer used
- The argument to `fclose` must be a file pointer obtained from a call of `fopen`
- `fclose` returns `zero` if the file was closed successfully
- Otherwise, it returns the error code `EOF` (a macro defined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



Opening and Closing Files

- *Example*: A program that opens a file for reading

```
#include <stdio.h>
#include <stdlib.h>
#define INPUT_FILE "example.dat"

int main(void)
{
   FILE *fp;

   fp = fopen(INPUT_FILE, "r");

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



Opening and Closing Files

- It is usual to see the call of `fopen` combined with the declaration of `fp`:

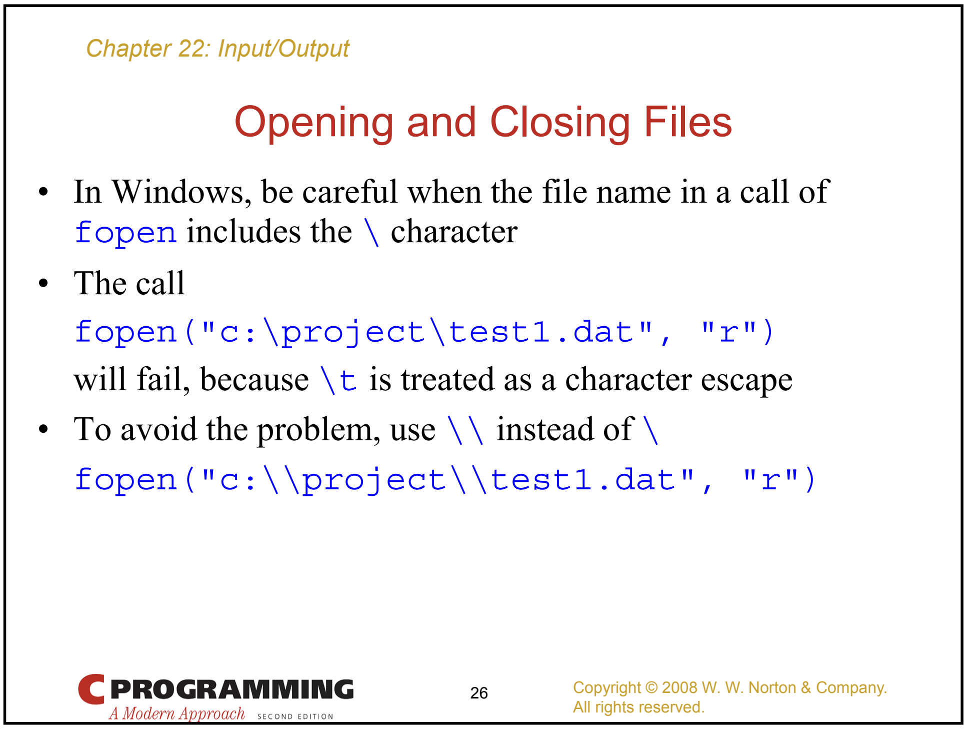
```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against `NULL`:

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

or simply

```
if (!(fp = fopen(FILE_NAME, "r"))) …
```



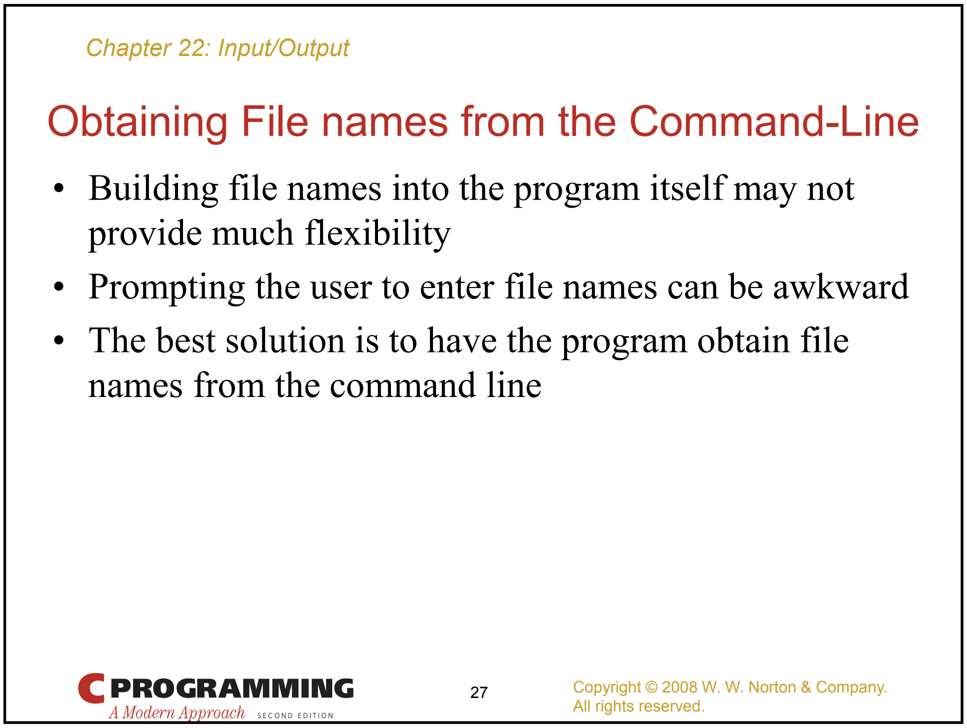
Opening and Closing Files

- 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
- To avoid the problem, use `\\` instead of `\`

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



Obtaining File names from the Command-Line

- Building file names into the program itself may not provide much flexibility
- Prompting the user to enter file names can be awkward
- The best solution is to have the program obtain file names from the command line

Obtaining File names from the Command-Line

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

```
int main(int argc, char *argv[])
{
  …
}
```

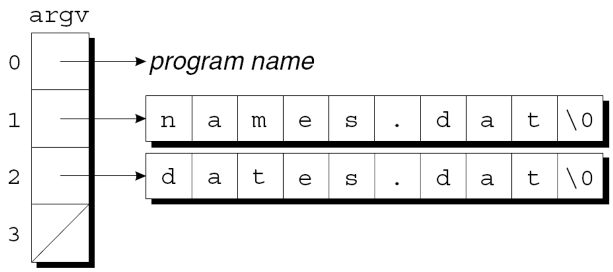
- `argc` is the number of command-line arguments
- `argv` is an array of pointers to the argument strings
  - `argv[0]` points to the program name
  - `argv[1]` through `argv[argc-1]` point to the remaining arguments
  - `argv[argc]` is a `NULL` pointer

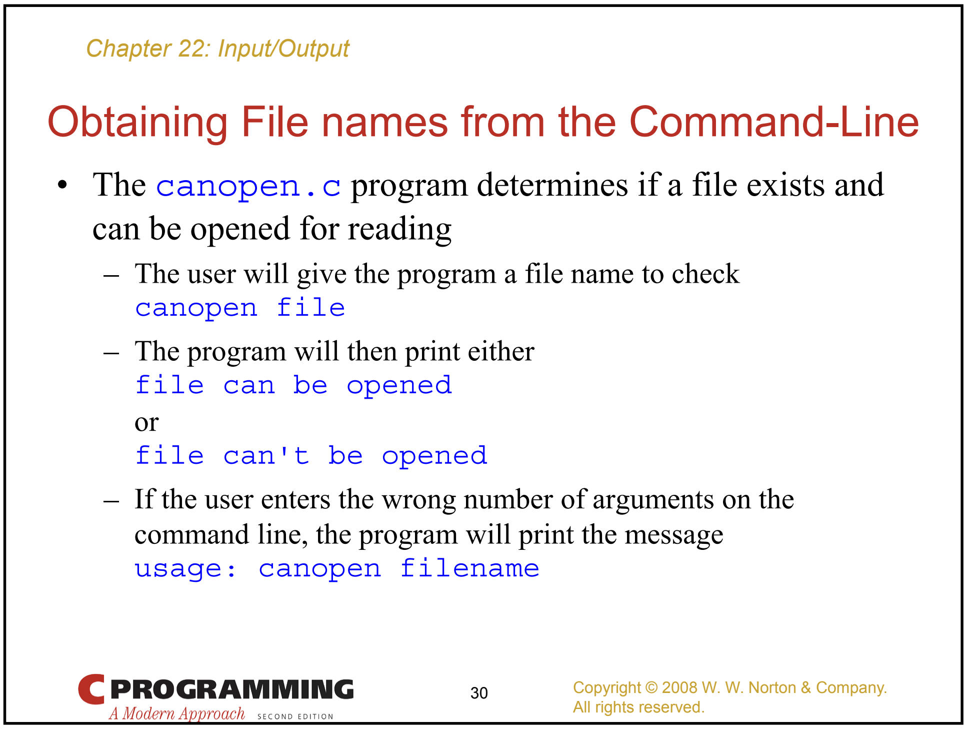
Obtaining File names from the Command-Line

- Example that uses the command line to supply two file names to a program named `demo`:

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

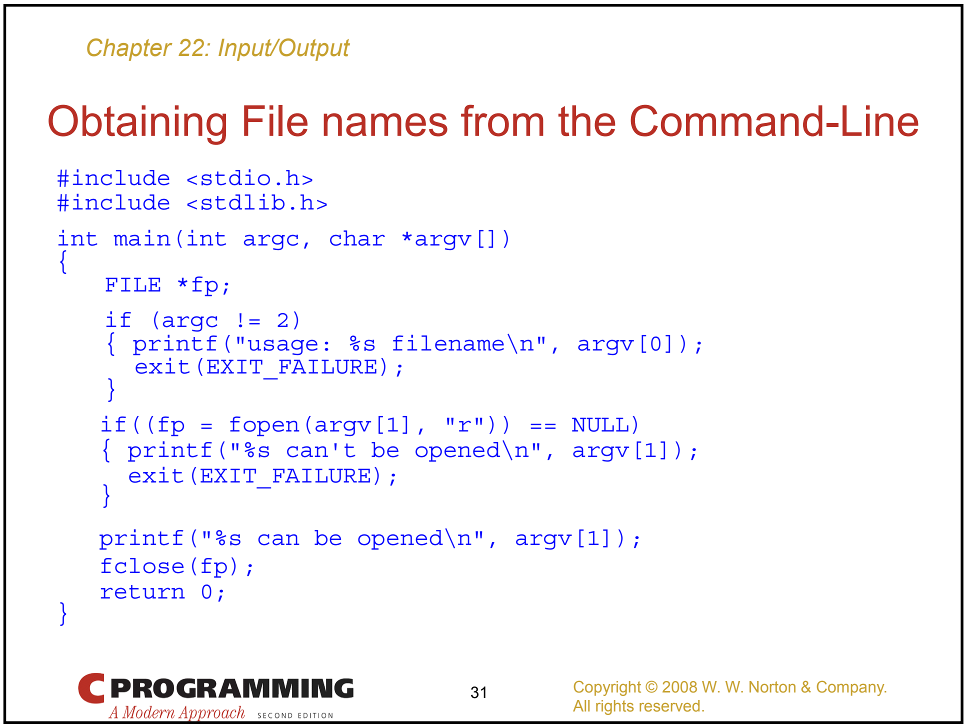
- In this example, `argc` is `3` and `argv` has the following appearance





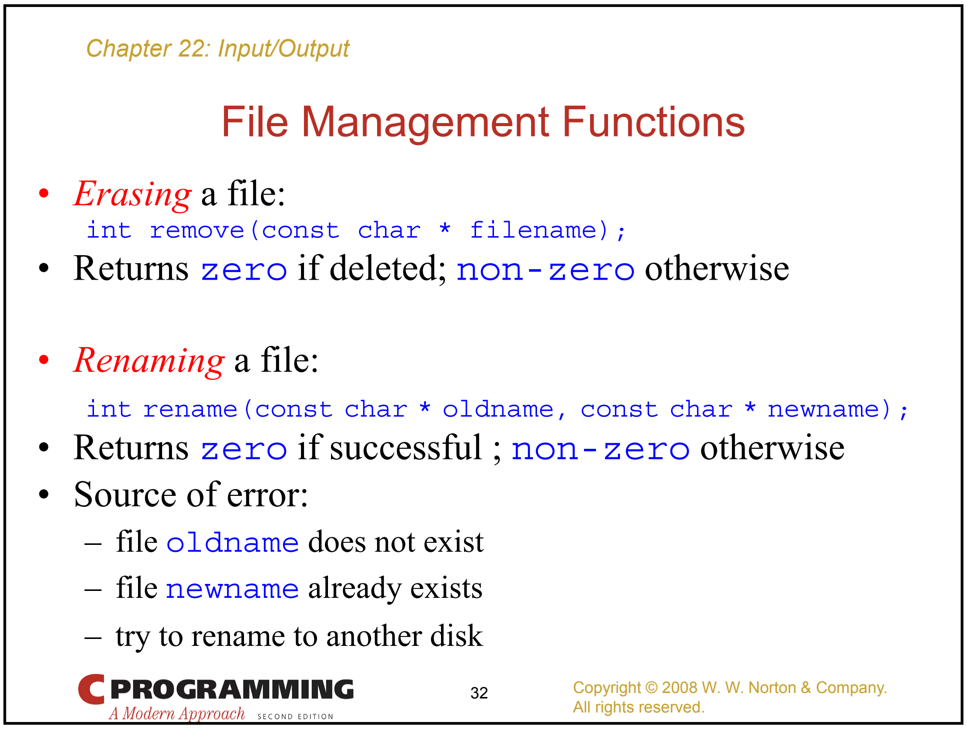
Obtaining File names from the Command-Line

- 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`



Obtaining File names from the Command-Line

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   FILE *fp;
   if (argc != 2)
   { printf("usage: %s filename\n", argv[0]);
     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;
}
```



File Management Functions

- *Erasing* a file:

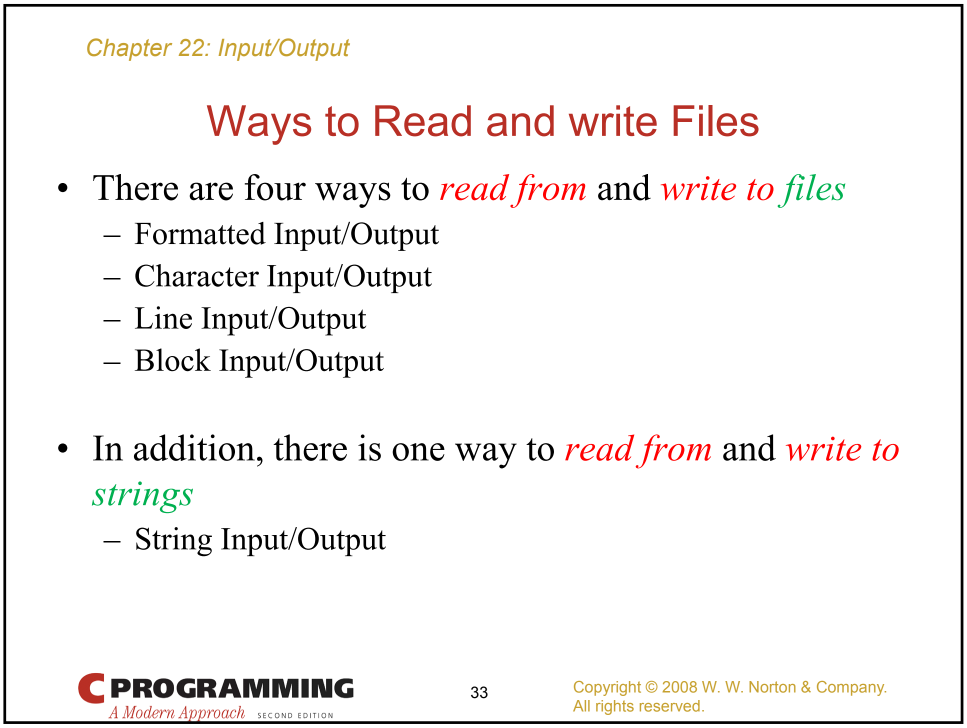
```
int remove(const char * filename);
```

- Returns `zero` if deleted; `non-zero` otherwise

- *Renaming* a file:

```
int rename(const char * oldname, const char * newname);
```

- Returns `zero` if successful ; `non-zero` otherwise
- Source of error:
  - file `oldname` does not exist
  - file `newname` already exists
  - try to rename to another disk



Ways to Read and write Files

- There are four ways to *read from* and *write to files*
  - Formatted Input/Output
  - Character Input/Output
  - Line Input/Output
  - Block Input/Output

- In addition, there is one way to *read from* and *write to strings*
  - String Input/Output

Formatted Input/Output

- Formatted input/output functions use format strings to control reading and writing
- `scanf` and `printf` (covered in Chapter 3 and Chapter 7) have the ability to
  - convert data from *character form* to *numeric (binary) form* during input

    32767 ➔ `01111111 11111111`
  - convert data from *numeric (binary) form* to *character form* during output

    `01111111 11111111` ➔ 32767

Formatted Input/Output

- `scanf` and `fscanf` read data items from an input stream
- Input items are converted (according to conversion specifications in the format string) and stored
- Prototypes for `scanf` and `fscanf`
  - `int scanf(const char *format, ...)`
    Reads formatted input from `stdin`
  - `int fscanf(FILE *fp, const char *format, ...)`
    Reads formatted input from `fp`

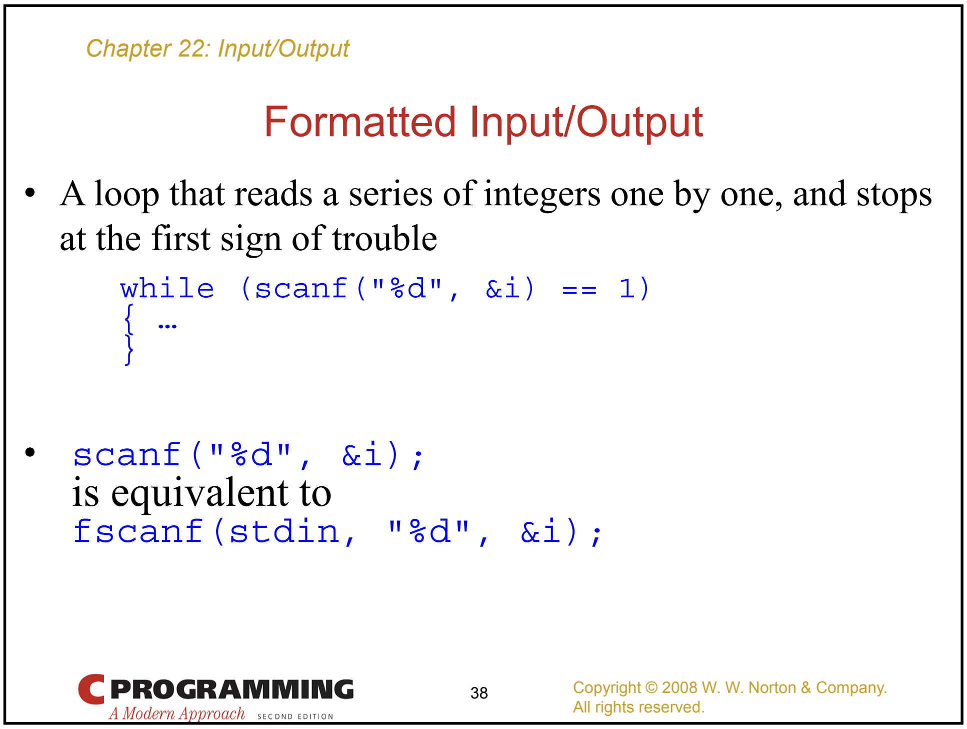
Formatted Input/Output

- The value returned by `scanf` and `fscanf` indicates the *actual number of input items* that were read
- If *end-of-file* occurred *before even one item could be read* by `scanf` and `fscanf`, the return value is `EOF`, otherwise the number of successfully read items will be returned
- Errors that cause the `scanf` and `fscanf` functions to return prematurely are:
  - ***Input failure***
    - no more input characters could be read
  - ***Matching failure***
    - the input characters didn't match the format string
    - The input character that didn't match is *pushed back* to be read in the future

Formatted Input/Output

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

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%d%d", &i, &j);`	`12•,•34¤`	`n`: 1 `i`: 12 `j`: unchanged
`n = scanf("%d,%d", &i, &j);`	`12•,•34¤`	`n`: 1 `i`: 12 `j`: unchanged
`n = scanf("%d, %d", &i, &j);`	`12•,•34¤`	`n`: 1 `i`: 12 `j`: unchanged
`n = scanf("%d ,%d", &i, &j);`	`12•,•34¤`	`n`: 2 `i`: 12 `j`: 34

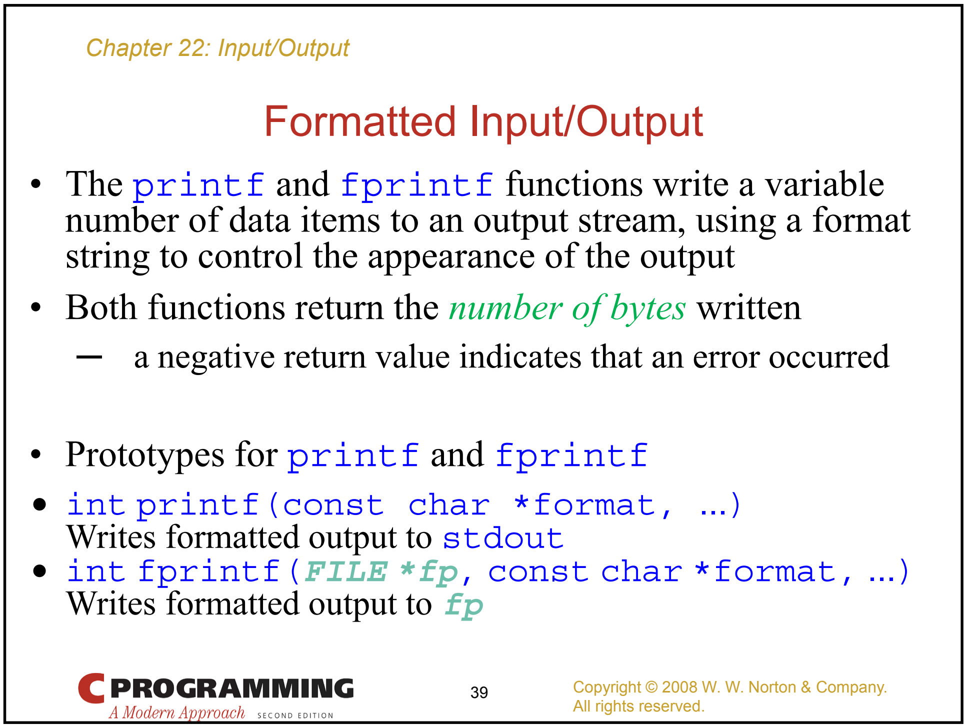


Formatted Input/Output

- A loop that reads a series of integers one by one, and stops at the first sign of trouble

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

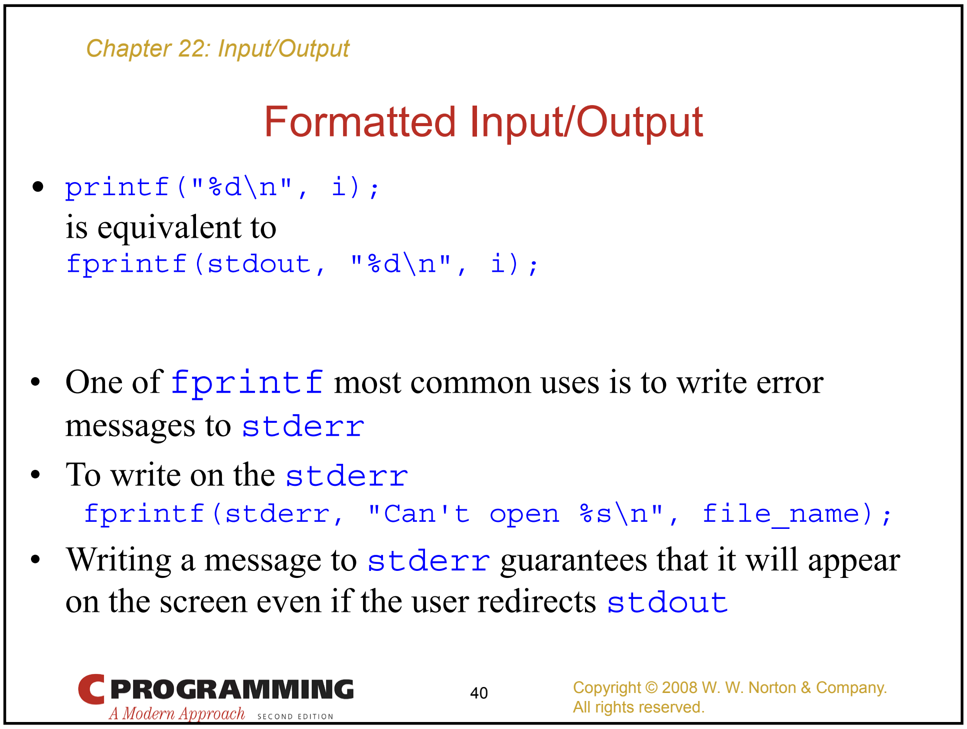
- `scanf("%d", &i);`
  is equivalent to
  `fscanf(stdin, "%d", &i);`



Formatted Input/Output

- The `printf` and `fprintf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output
- Both functions return the *number of bytes* written
  - a negative return value indicates that an error occurred

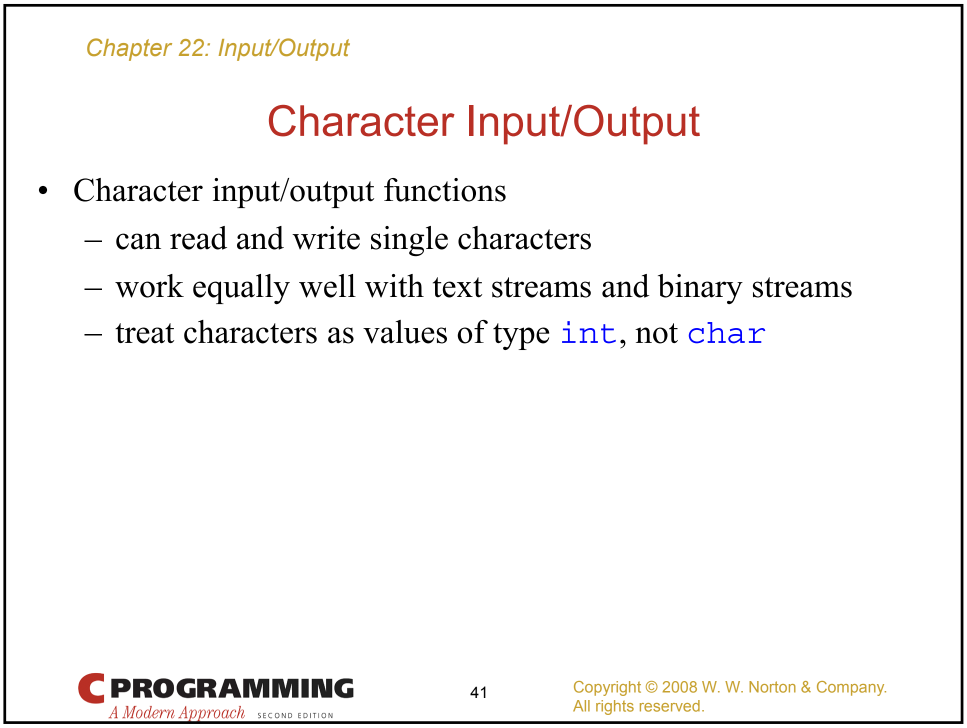
- Prototypes for `printf` and `fprintf`
- `int printf(const char *format, …)`
  Writes formatted output to `stdout`
- `int fprintf(`***`FILE *fp`***`, const char *format, …)`
  Writes formatted output to ***`fp`***



Formatted Input/Output

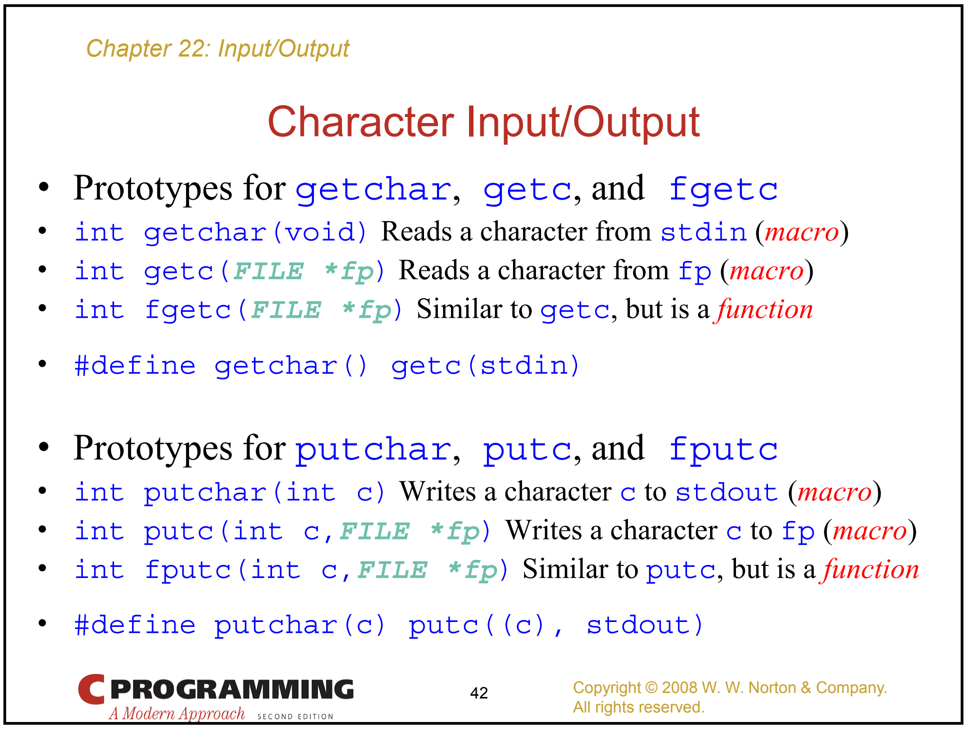
- `printf("%d\n", i);`
  is equivalent to
  `fprintf(stdout, "%d\n", i);`

- One of `fprintf` most common uses is to write error messages to `stderr`
- To write on the `stderr`
  `fprintf(stderr, "Can't open %s\n", file_name);`
- Writing a message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`



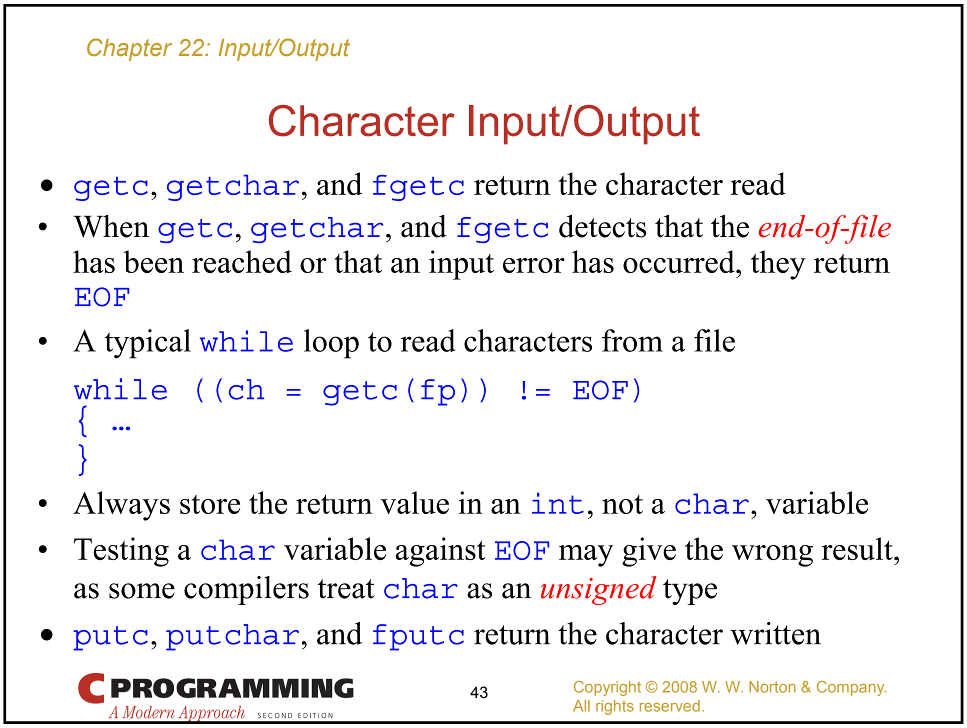
Character Input/Output

- Character input/output functions
  - can read and write single characters
  - work equally well with text streams and binary streams
  - treat characters as values of type `int`, not `char`



Character Input/Output

- Prototypes for `getchar`, `getc`, and `fgetc`
- `int getchar(void)` Reads a character from `stdin` (*macro*)
- `int getc(FILE *fp)` Reads a character from `fp` (*macro*)
- `int fgetc(FILE *fp)` Similar to `getc`, but is a *function*
- `#define getchar() getc(stdin)`
- Prototypes for `putchar`, `putc`, and `fputc`
- `int putchar(int c)` Writes a character `c` to `stdout` (*macro*)
- `int putc(int c,FILE *fp)` Writes a character `c` to `fp` (*macro*)
- `int fputc(int c,FILE *fp)` Similar to `putc`, but is a *function*
- `#define putchar(c) putc((c), stdout)`



Character Input/Output

- `getc`, `getchar`, and `fgetc` return the character read
- When `getc`, `getchar`, and `fgetc` detects that the *end-of-file* has been reached or that an input error has occurred, they return `EOF`
- A typical `while` loop to read characters from a file

```
while ((ch = getc(fp)) != EOF)
{ …
}
```

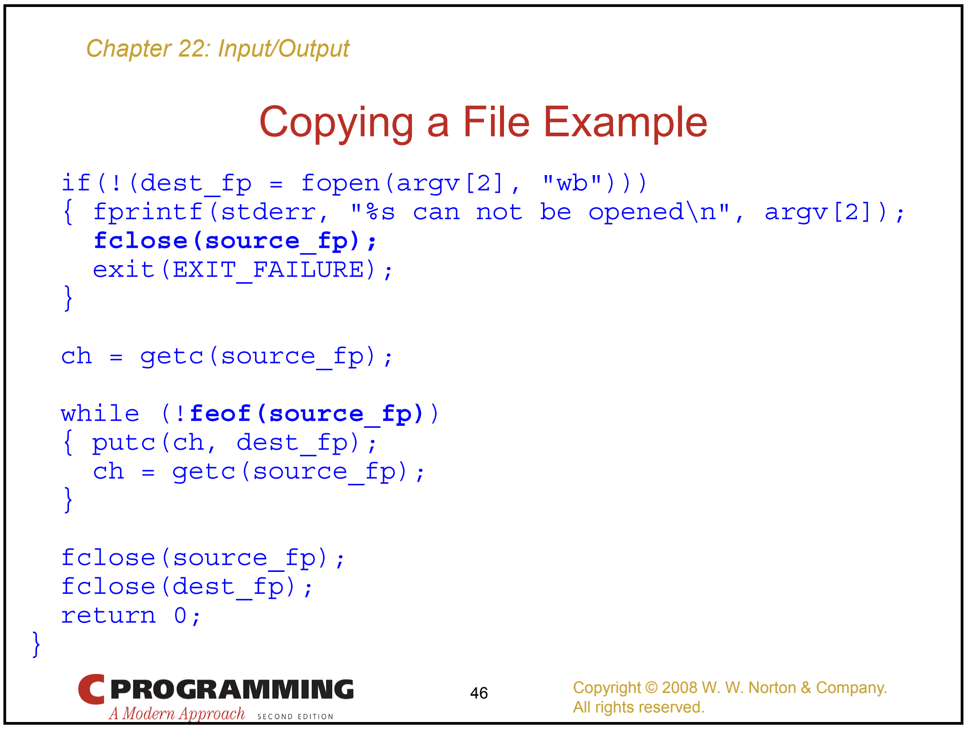
- Always store the return value in an `int`, not a `char`, variable
- Testing a `char` variable against `EOF` may give the wrong result, as some compilers treat `char` as an *unsigned* type
- `putc`, `putchar`, and `fputc` return the character written

Character Input/Output

- 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
- The call feof(source_fp) returns a nonzero value if the *end-of-file* indicator is set for the stream associated with source_fp

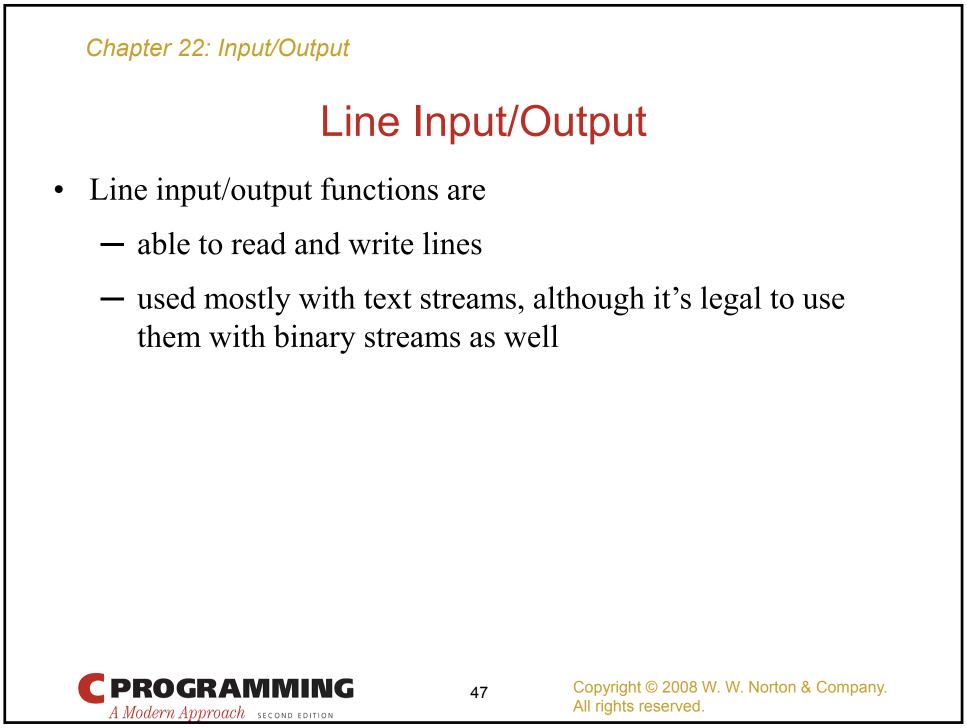
Copying a File Example

```
/* Copying 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: %s source dest\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  if (!(source_fp = fopen(argv[1], "rb")))
  { fprintf(stderr, "%s can not be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }
```



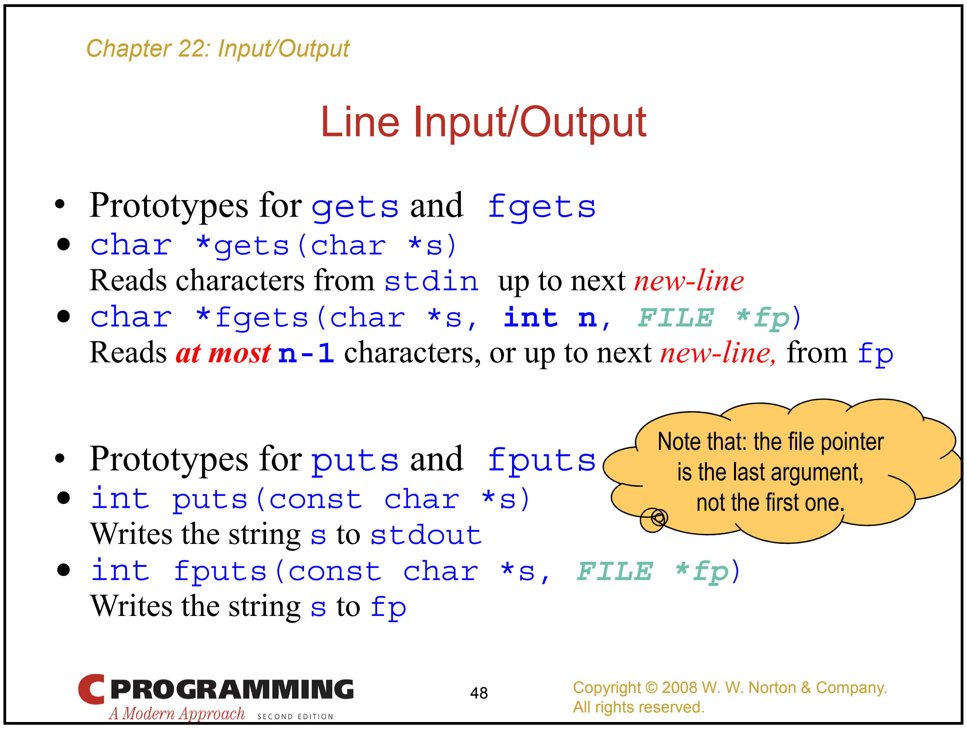
Copying a File Example

```
  if(!(dest_fp = fopen(argv[2], "wb")))
  { fprintf(stderr, "%s can not be opened\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  ch = getc(source_fp);

  while (!feof(source_fp))
  { putc(ch, dest_fp);
    ch = getc(source_fp);
  }

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



Line Input/Output

- Line input/output functions are
  - able to read and write lines
  - used mostly with text streams, although it’s legal to use them with binary streams as well

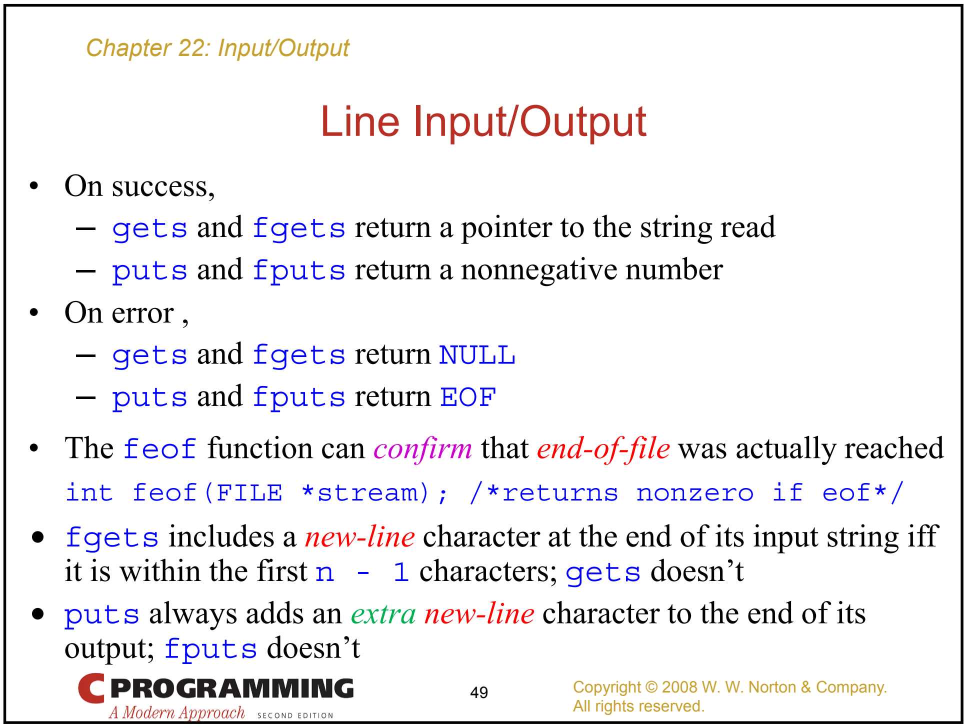


Line Input/Output

- Prototypes for `gets` and `fgets`
- `char *gets(char *s)`
  Reads characters from `stdin` up to next *new-line*
- `char *fgets(char *s, int n, FILE *fp)`
  Reads ***at most*** **`n-1`** characters, or up to next *new-line,* from `fp`

- Prototypes for `puts` and `fputs`
- `int puts(const char *s)`
  Writes the string `s` to `stdout`
- `int fputs(const char *s, FILE *fp)`
  Writes the string `s` to `fp`

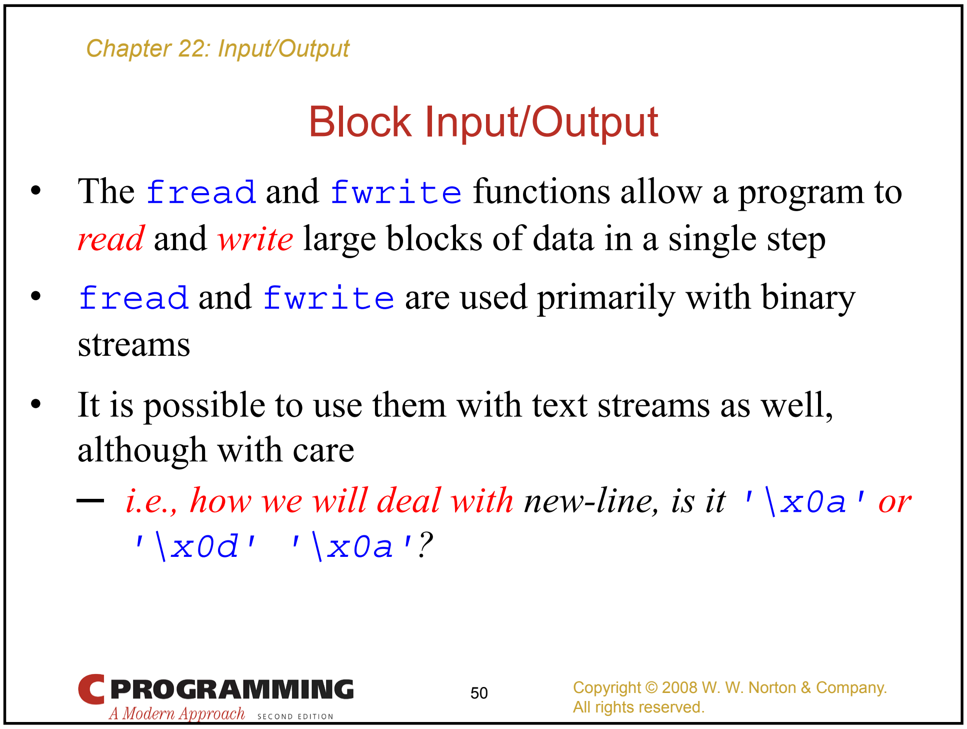
Note that: the file pointer
is the last argument,
not the first one.



Line Input/Output

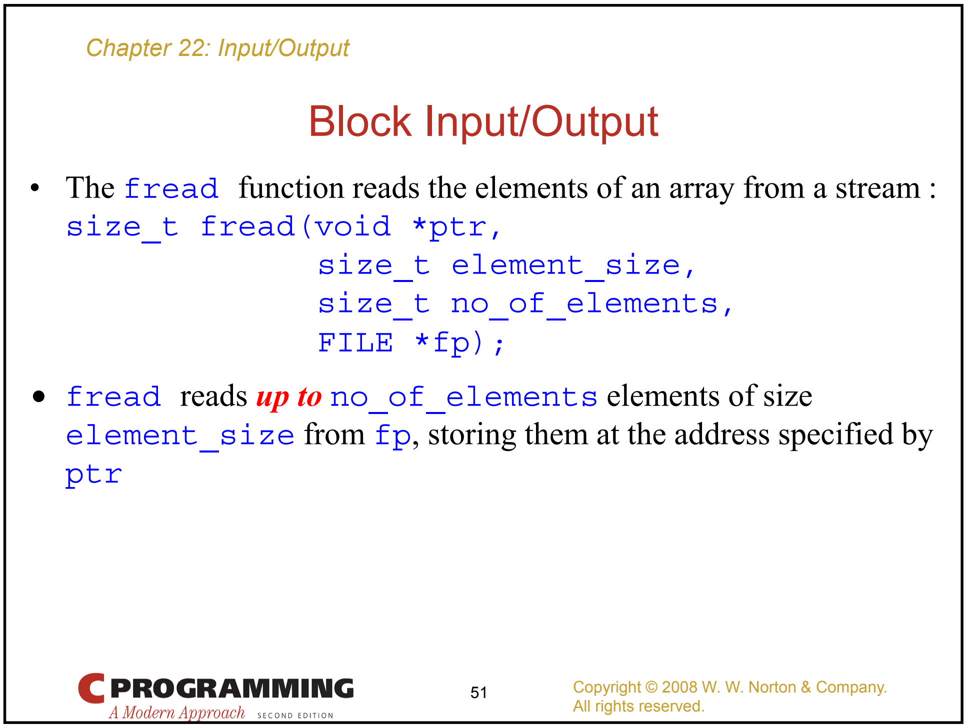
- On success,
  - `gets` and `fgets` return a pointer to the string read
  - `puts` and `fputs` return a nonnegative number
- On error ,
  - `gets` and `fgets` return `NULL`
  - `puts` and `fputs` return `EOF`
- The `feof` function can *confirm* that *end-of-file* was actually reached

  `int feof(FILE *stream); /*returns nonzero if eof*/`
- `fgets` includes a *new-line* character at the end of its input string iff it is within the first `n - 1` characters; `gets` doesn't
- `puts` always adds an *extra new-line* character to the end of its output; `fputs` doesn't



Block Input/Output

- 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
- It is possible to use them with text streams as well, although with care
  - *i.e., how we will deal with new-line, is it* `'\x0a'` *or* `'\x0d' '\x0a'`*?*

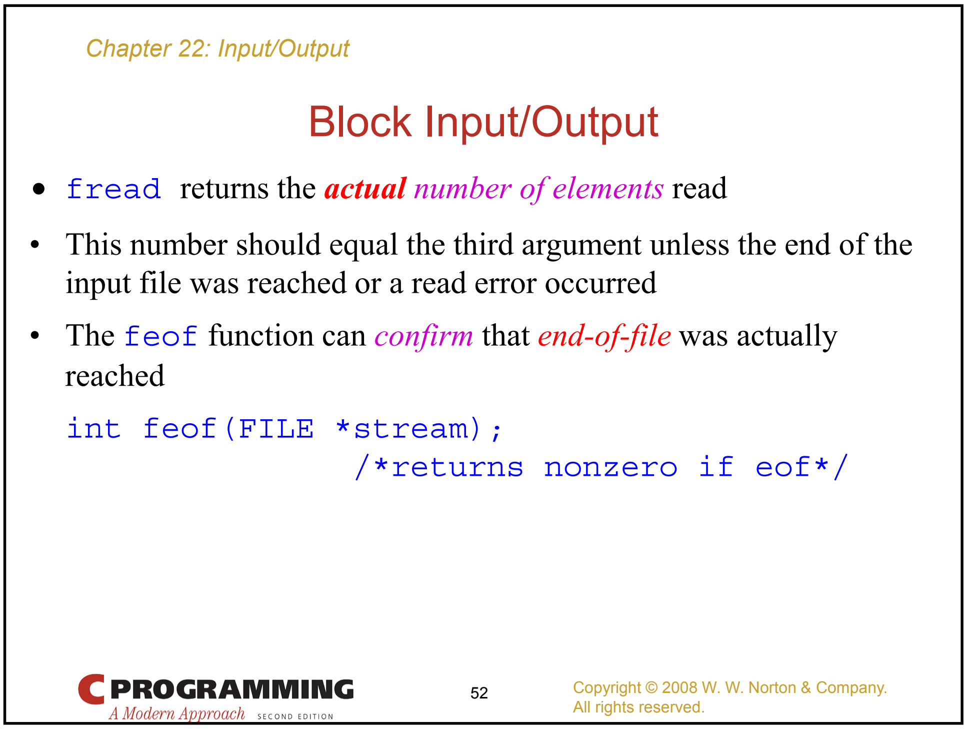


Block Input/Output

- The `fread` function reads the elements of an array from a stream :

```
size_t fread(void *ptr,
             size_t element_size,
             size_t no_of_elements,
             FILE *fp);
```

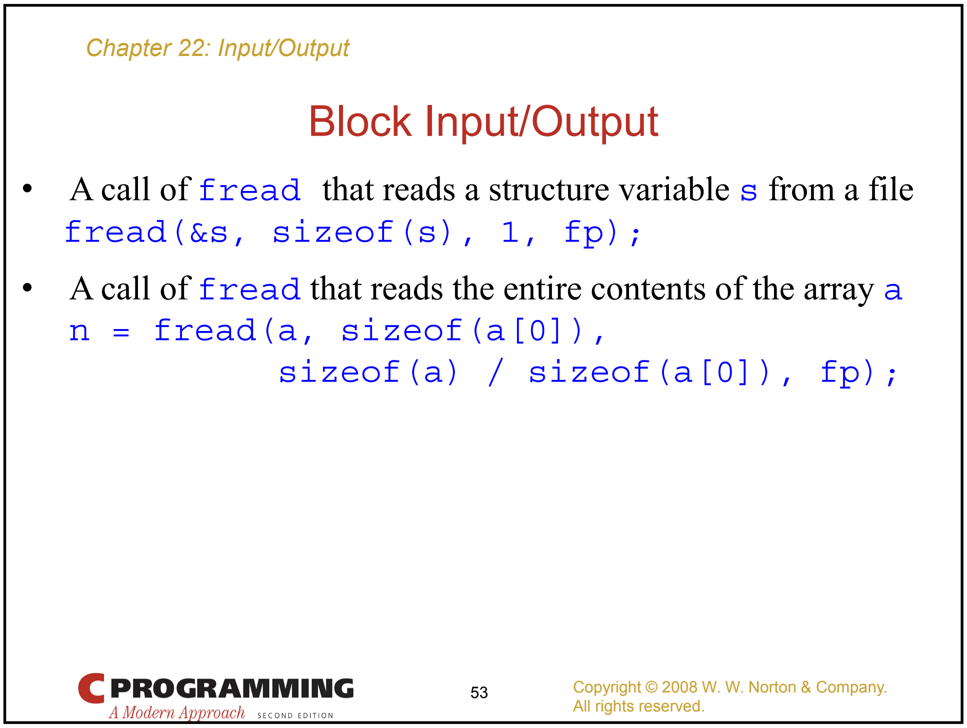
- `fread` reads ***up to*** `no_of_elements` elements of size `element_size` from `fp`, storing them at the address specified by `ptr`



Block Input/Output

- `fread` returns 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
- The `feof` function can *confirm* that *end-of-file* was actually reached

```
int feof(FILE *stream);
               /*returns nonzero if eof*/
```



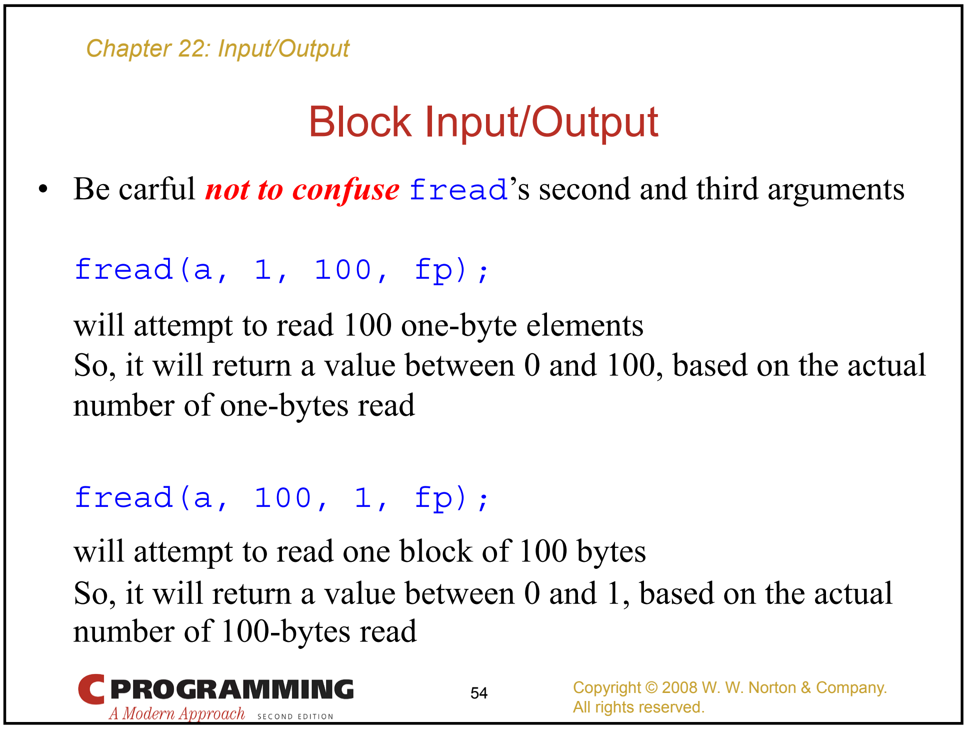
Block Input/Output

- A call of `fread` that reads a structure variable `s` from a file

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

- A call of `fread` that reads the entire contents of the array `a`

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```



Block Input/Output

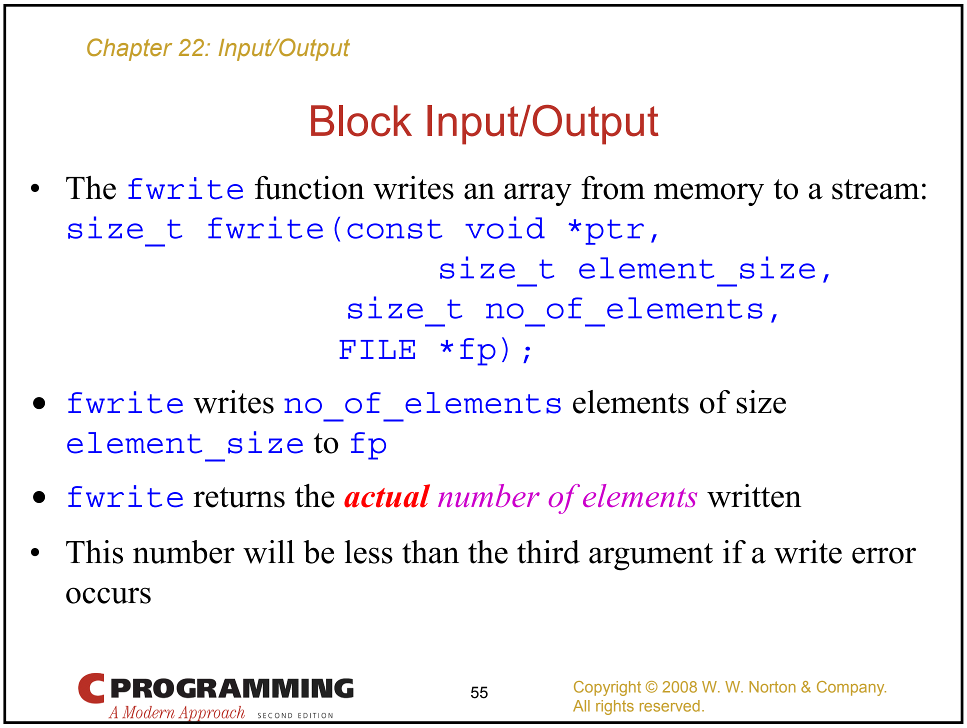
- Be carful ***not to confuse*** `fread`'s second and third arguments

```
fread(a, 1, 100, fp);
```

will attempt to read 100 one-byte elements
So, it will return a value between 0 and 100, based on the actual number of one-bytes read

```
fread(a, 100, 1, fp);
```

will attempt to read one block of 100 bytes
So, it will return a value between 0 and 1, based on the actual number of 100-bytes read

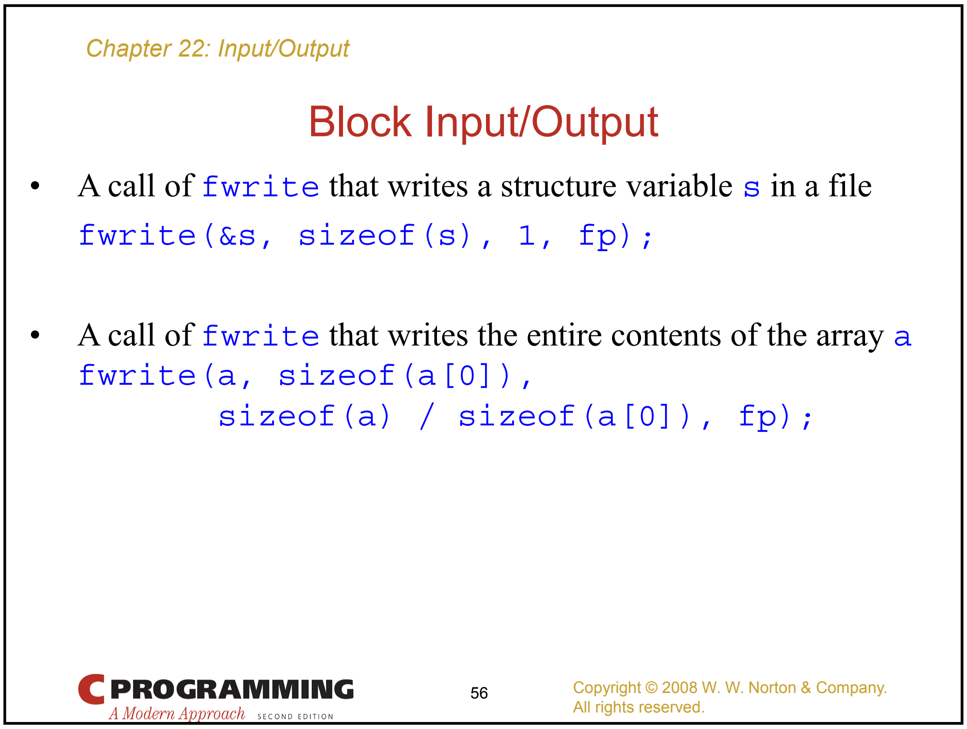


Block Input/Output

- The `fwrite` function writes an array from memory to a stream:

```
size_t fwrite(const void *ptr,
                   size_t element_size,
              size_t no_of_elements,
             FILE *fp);
```

- `fwrite` writes `no_of_elements` elements of size `element_size` to `fp`
- `fwrite` returns the ***actual*** *number of elements* written
- This number will be less than the third argument if a write error occurs



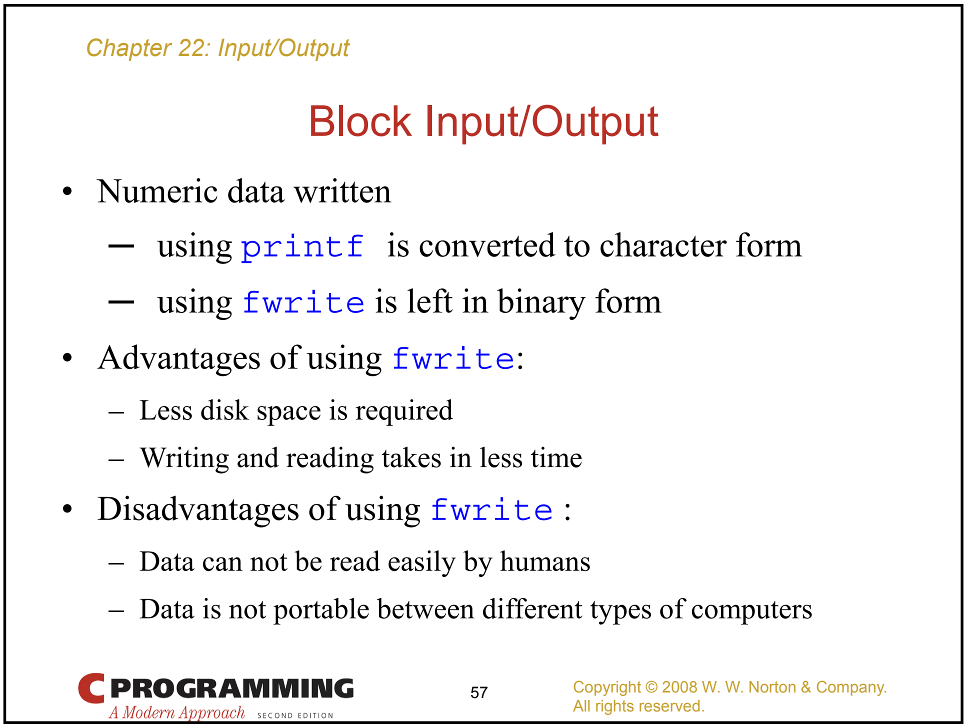
Block Input/Output

- A call of `fwrite` that writes a structure variable `s` in a file

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

- 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 Input/Output

- Numeric data written
  - using `printf` is converted to character form
  - using `fwrite` is left in binary form
- Advantages of using `fwrite`:
  - Less disk space is required
  - Writing and reading takes in less time
- Disadvantages of using `fwrite` :
  - Data can not be read easily by humans
  - Data is not portable between different types of computers

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 functions that allow a program to determine the current file position or to change it

File Positioning

- The `fseek`, `ftell`, and `rewind` functions support random access within files
- Random access is most often used with binary files
- `fseek` allows repositioning within a file

```
int fseek(FILE *stream, long int offset, int place);
```

- The new file position is determined by `offset` and `place`
- `offset` is a (*possibly negative*) byte count relative to the position specified by `place`

File Positioning

- *Place* can be SEEK_SET, SEEK_CUR, or SEEK_END

```
#define SEEK_SET  0 /* Seek from beginning of file.*/
#define SEEK_CUR  1 /* Seek from current position. */
#define SEEK_END  2 /* Seek from end of file.      */
```

- Examples of fseek:

```
fseek(fp,0,SEEK_SET);   /*move to beginning of file*/
fseek(fp,-10,SEEK_CUR); /*move back 10 bytes       */
fseek(fp,0,SEEK_END);   /*move to end of file      */
```

File Positioning

- ftell returns the current file position, relative to the beginning of the file

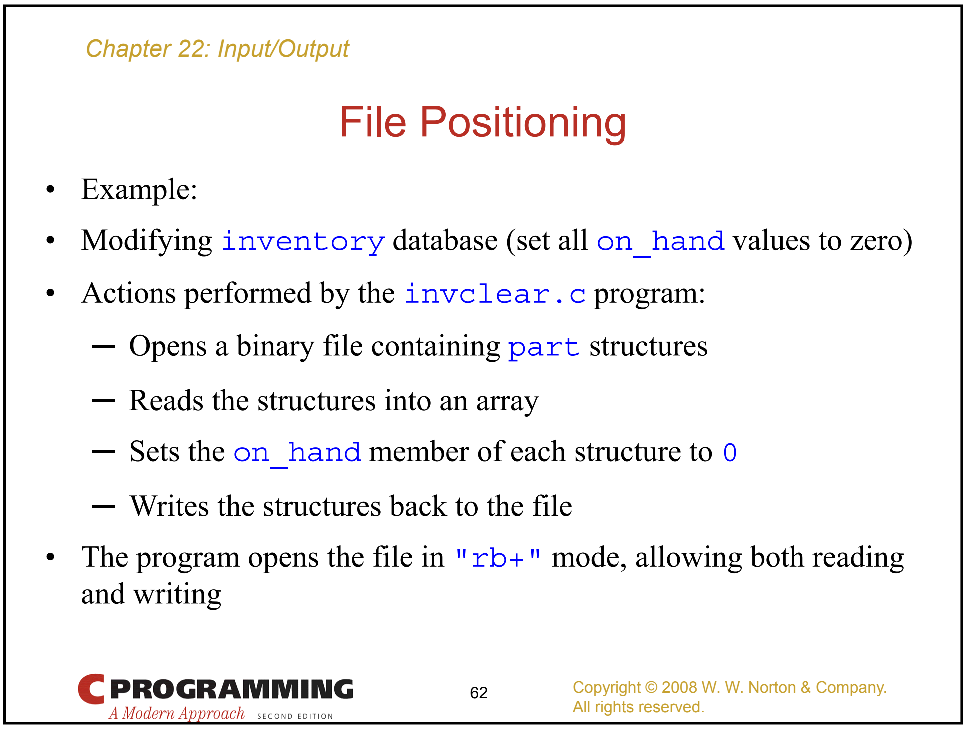
```
long int ftell(FILE *stream);
```

- This may be saved and later supplied to a call of fseek:

```
long int file_pos;
file_pos = ftell(fp);
…
fseek(fp, file_pos, SEEK_SET);
/* return to previous position */
```

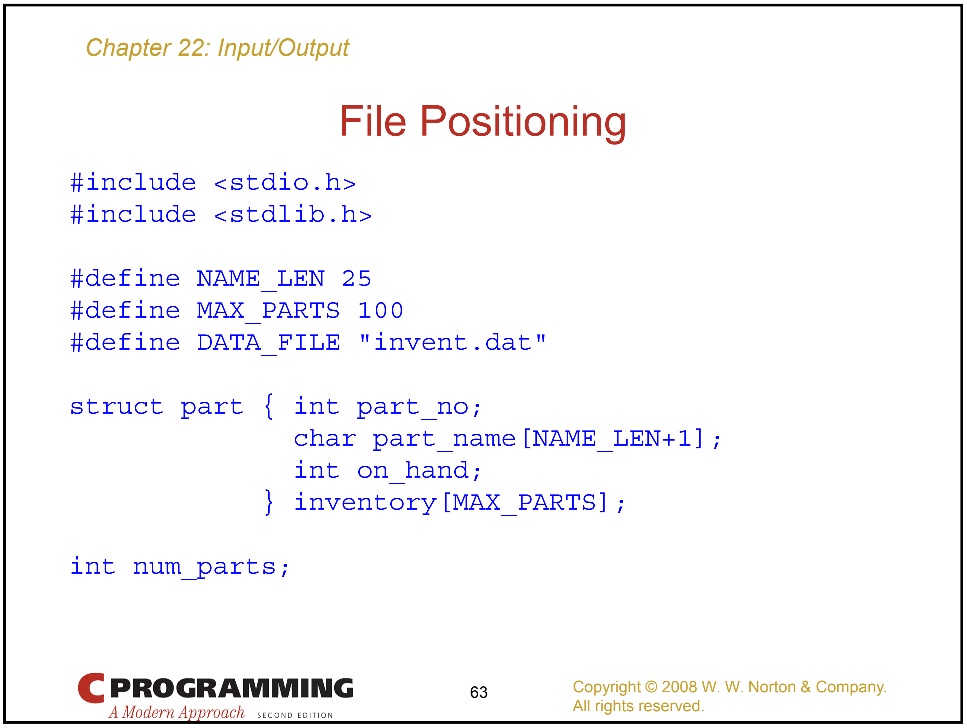
- The call rewind(fp) is equivalent to

```
fseek(fp,0,SEEK_SET);
```



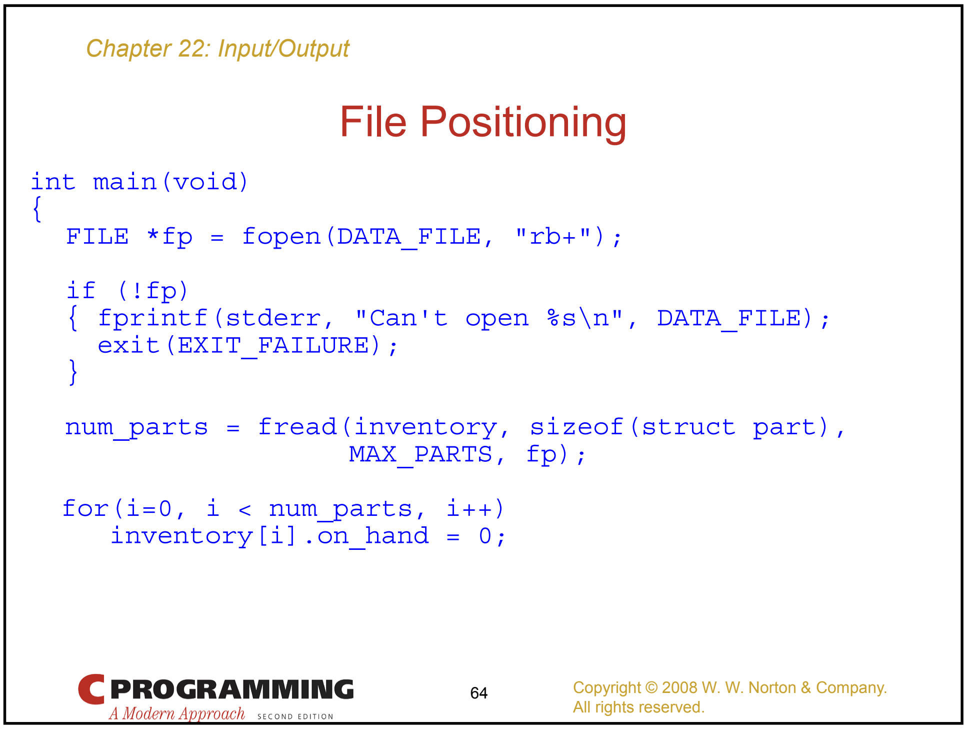
File Positioning

- Example:
- Modifying `inventory` database (set all `on_hand` values to zero)
- 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



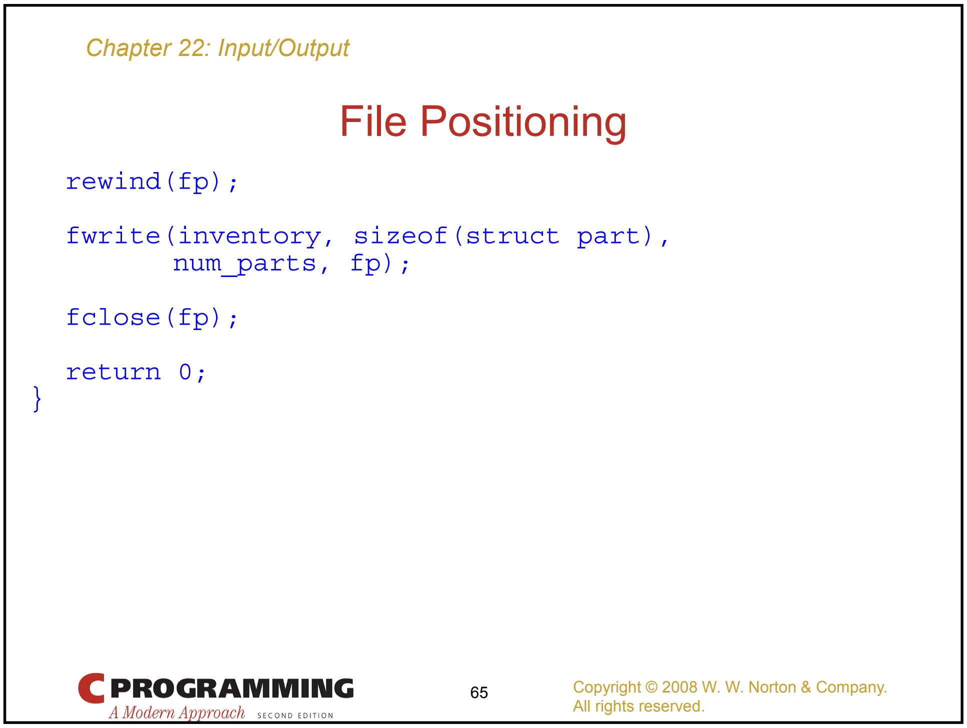
File Positioning

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

#define NAME_LEN 25
#define MAX_PARTS 100
#define DATA_FILE "invent.dat"

struct part { int part_no;
              char part_name[NAME_LEN+1];
              int on_hand;
            } inventory[MAX_PARTS];

int num_parts;
```



File Positioning

```
int main(void)
{
  FILE *fp = fopen(DATA_FILE, "rb+");

  if (!fp)
  { fprintf(stderr, "Can't open %s\n", DATA_FILE);
    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;
```



File Positioning

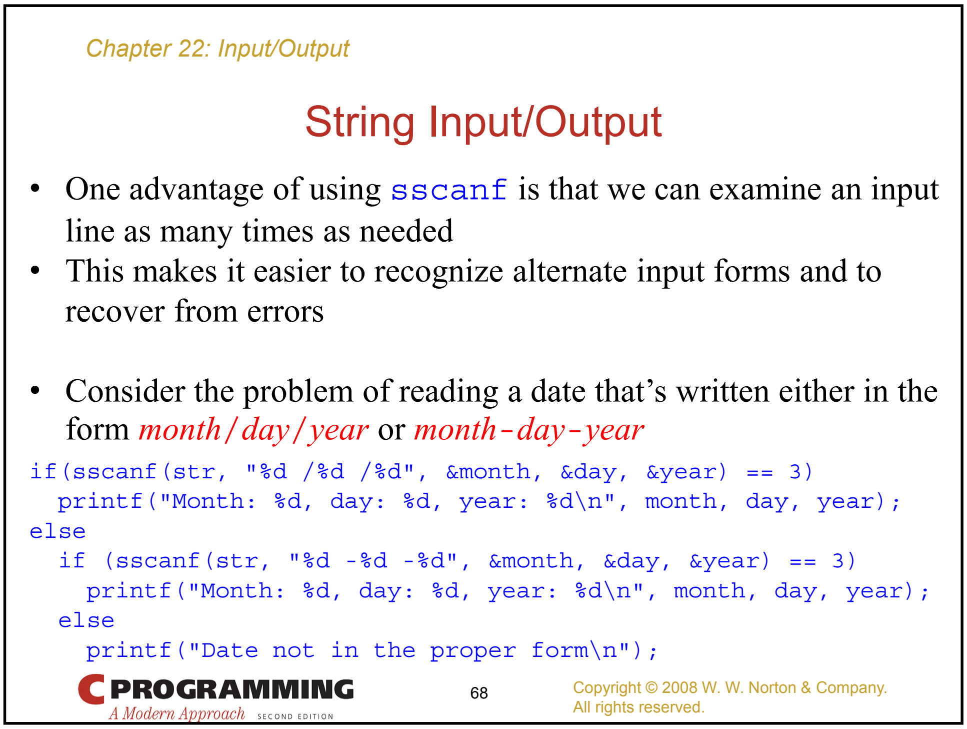
```
  rewind(fp);

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

  fclose(fp);

  return 0;
}
```

String Input/Output

- String input/output functions can read and write data using a string as though it were a stream
- `sscanf` reads characters from a string
- `sprintf` and `snprintf` write characters into a string
- These functions are closely resemble `scanf` and `printf`

String Input/Output

- Prototype for `sscanf`
-
```
int sscanf(const char *string,
           const char *format, ...)
```
- Reads formatted input from a string (pointed to by its 1st argument), instead of reading from a stream
- `sscanf`'s second argument is a format string identical to that used by `scanf` and `fscanf`
- 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

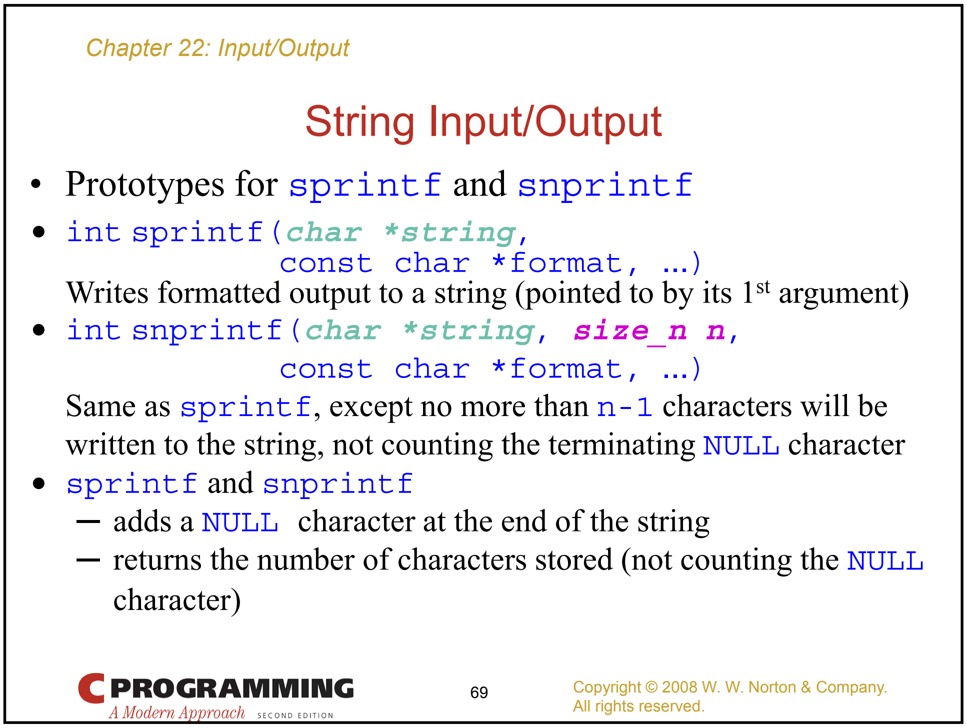


String Input/Output

- 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");
```



String Input/Output

- Prototypes for `sprintf` and `snprintf`
- `int sprintf(char *string,`
  `const char *format, …)`
  Writes formatted output to a string (pointed to by its 1st argument)
- `int snprintf(char *string, size_n n,`
  `const char *format, …)`
  Same as `sprintf`, except no more than `n-1` characters will be written to the string, not counting the terminating `NULL` character
- `sprintf` and `snprintf`
  - adds a `NULL` character at the end of the string
  - returns the number of characters stored (not counting the `NULL` character)

String Input/Output

- Examples:
- A call that writes `"9/20/2010"` into `date`:

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```

- `sprintf(name, "%s, %s", "Einstein", "Albert");`
  The string `"Einstein, Albert"` is written into `name`

- `snprintf(name, 15, "%s,    %s", "Einstein", "Albert");`
  The string `"Einstein, Albe"` is written into `name`