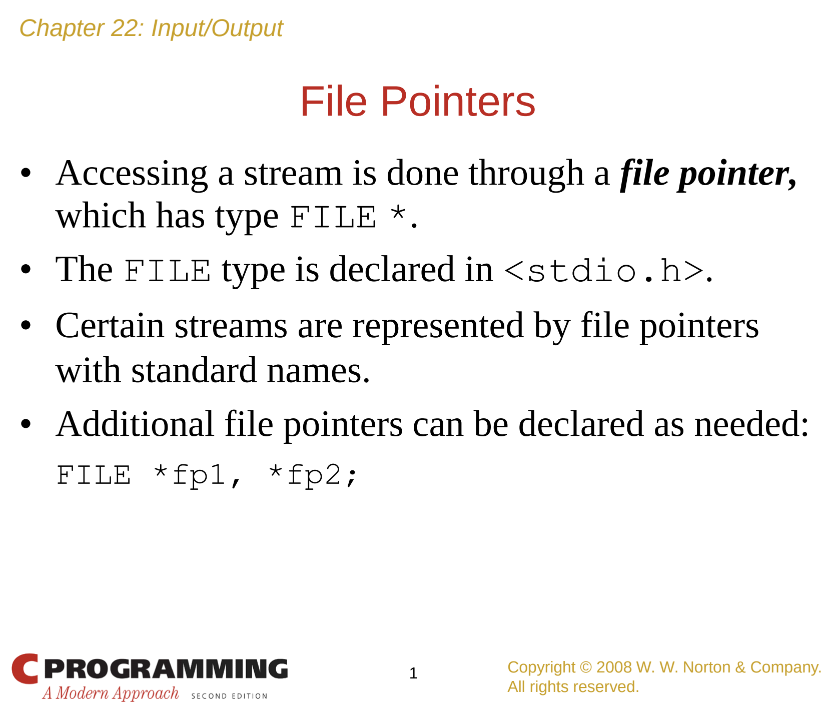
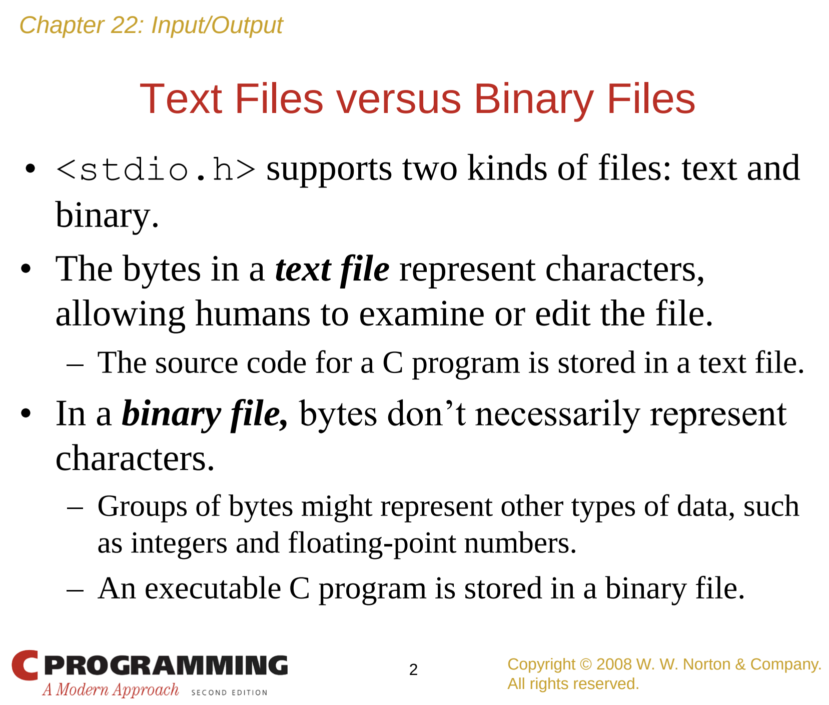


## File Pointers

- 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;
```



## Text Files versus Binary Files

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

## Text Files versus Binary Files

- Text files have two characteristics that binary files don't possess.
- ***Text files are divided into lines.*** Each line in a text file normally ends with one or two special characters.
  - Windows: carriage-return character (`'\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.

# Text Files versus Binary Files

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

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

# Text Files versus Binary Files

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

| 01111111 | 11111111 |
|---|---|

- Storing numbers in binary can often save space.

## Opening a File

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

```
FILE *fopen(const char * filename,
            const char * 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

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

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

- When it can't open a file, `fopen` returns a null pointer.

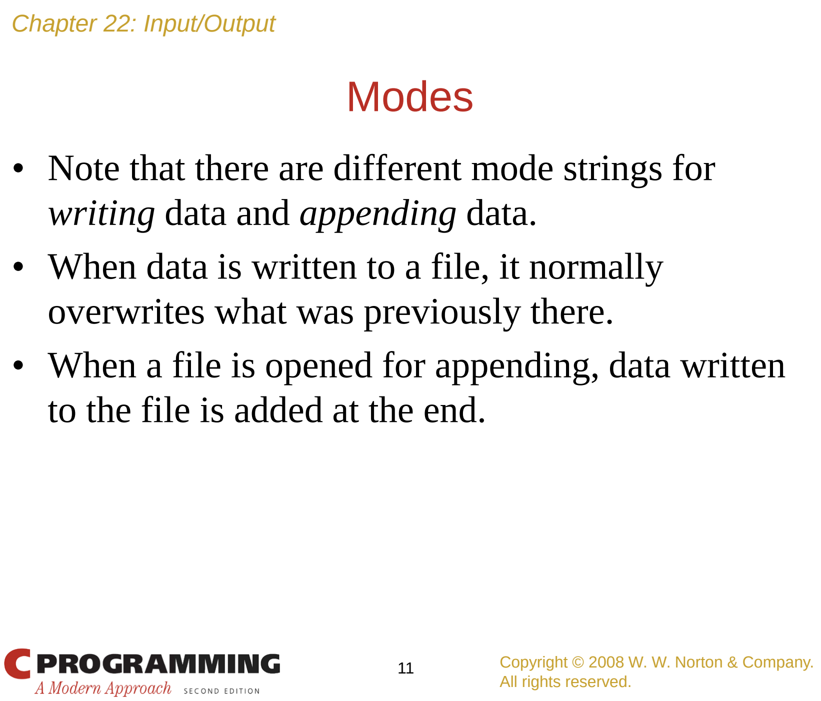
## Modes

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

## Modes

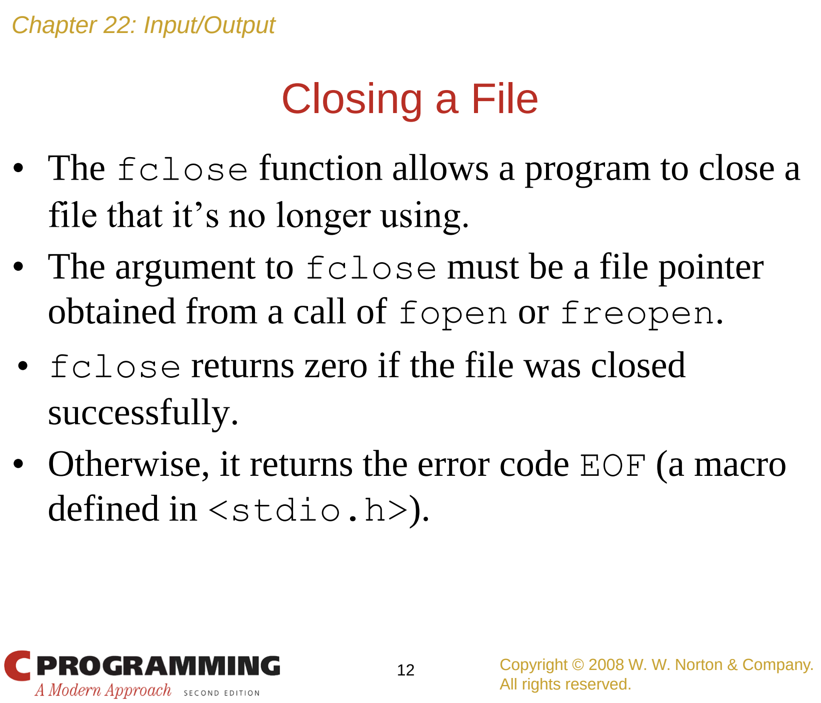
- Mode strings for text files:

| ***String*** | ***Meaning*** |
|---|---|
| `"r"` | Open for reading |
| `"w"` | Open for writing (file need not exist) |
| `"a"` | Open for appending (file need not exist) |



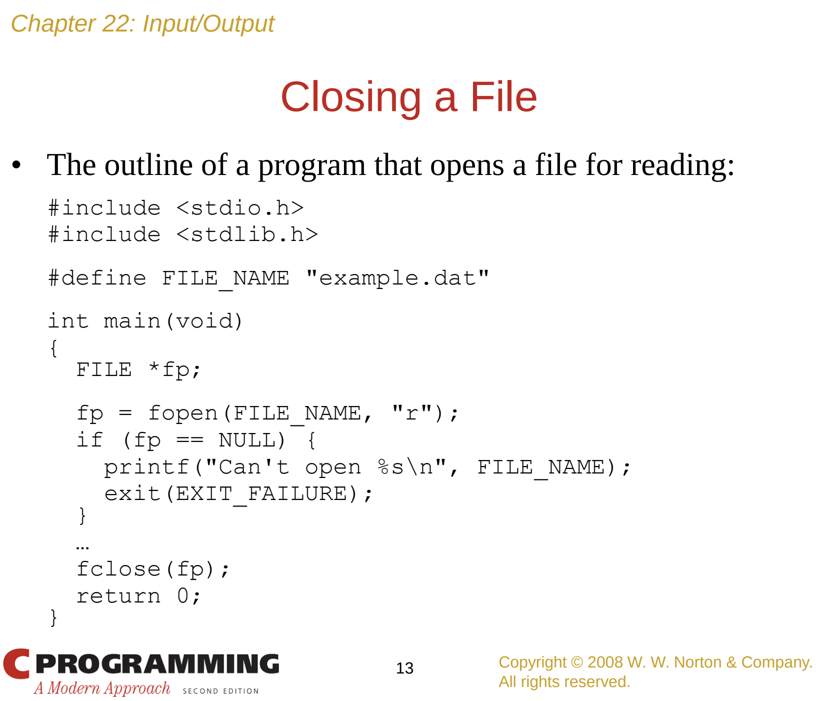
## Modes

- 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.



## Closing a File

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



## Closing a File

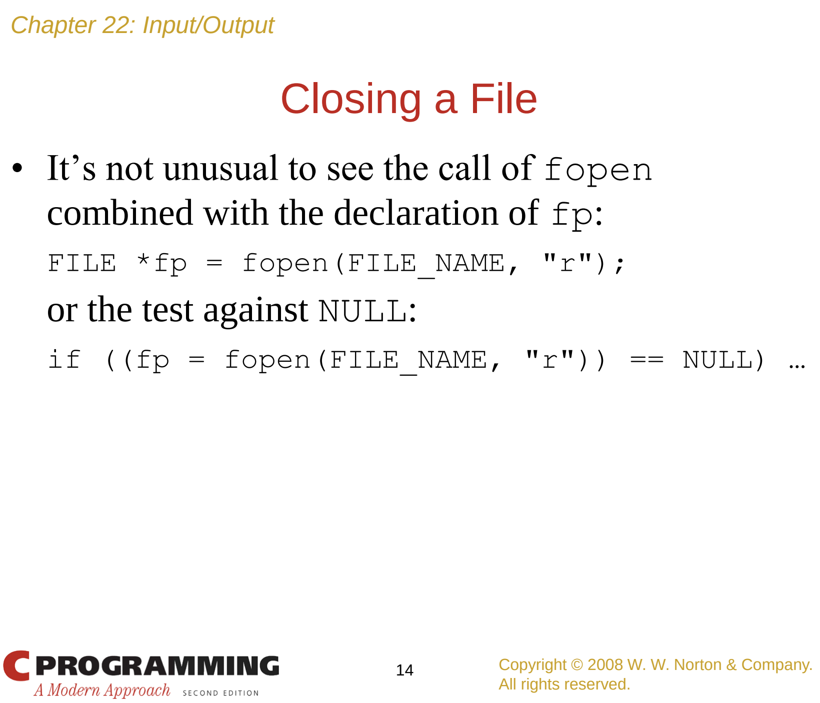
- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

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



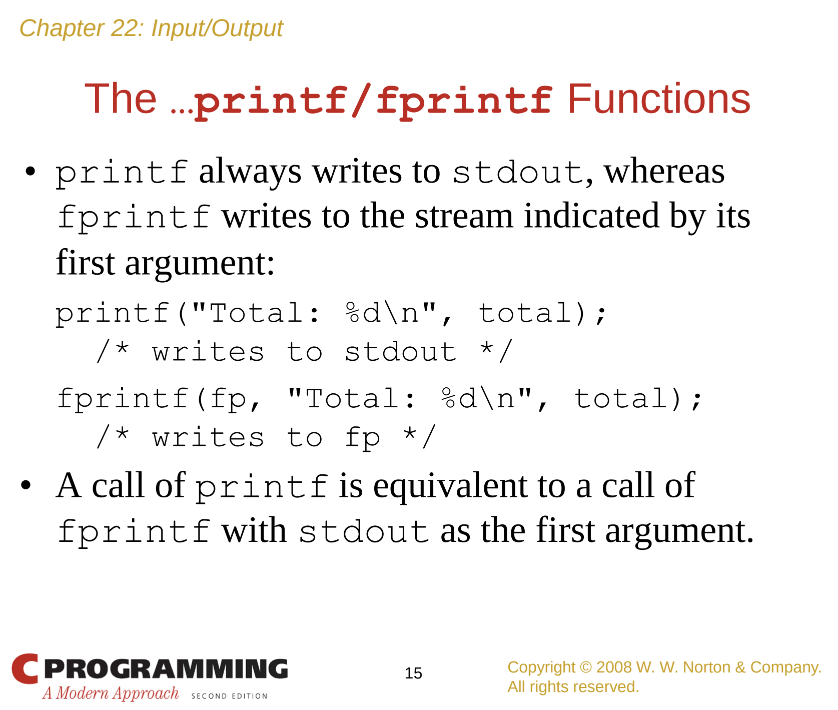
## Closing a File

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

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

or the test against `NULL`:

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

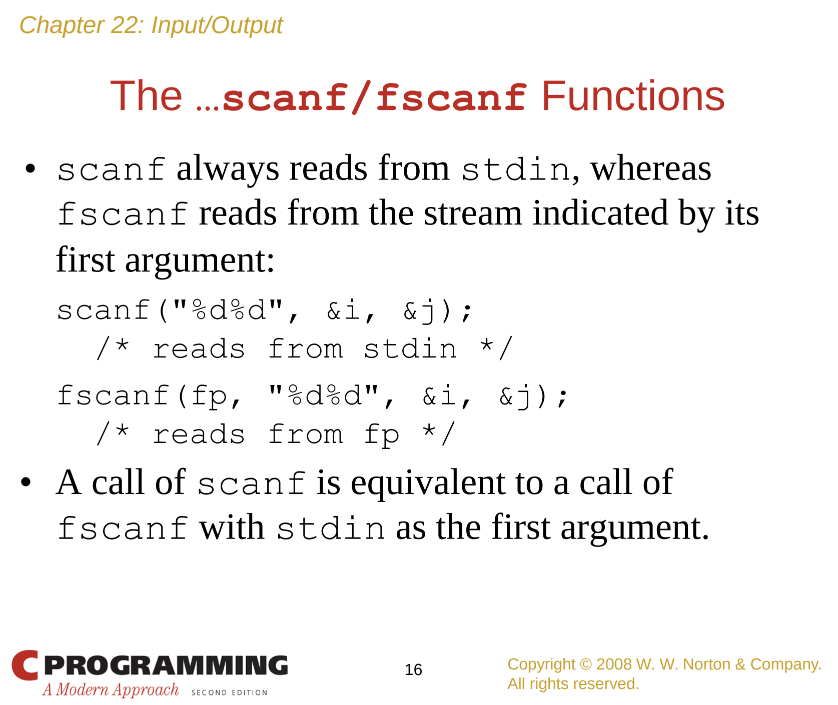


## The …printf/fprintf Functions

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

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- A call of printf is equivalent to a call of fprintf with stdout as the first argument.

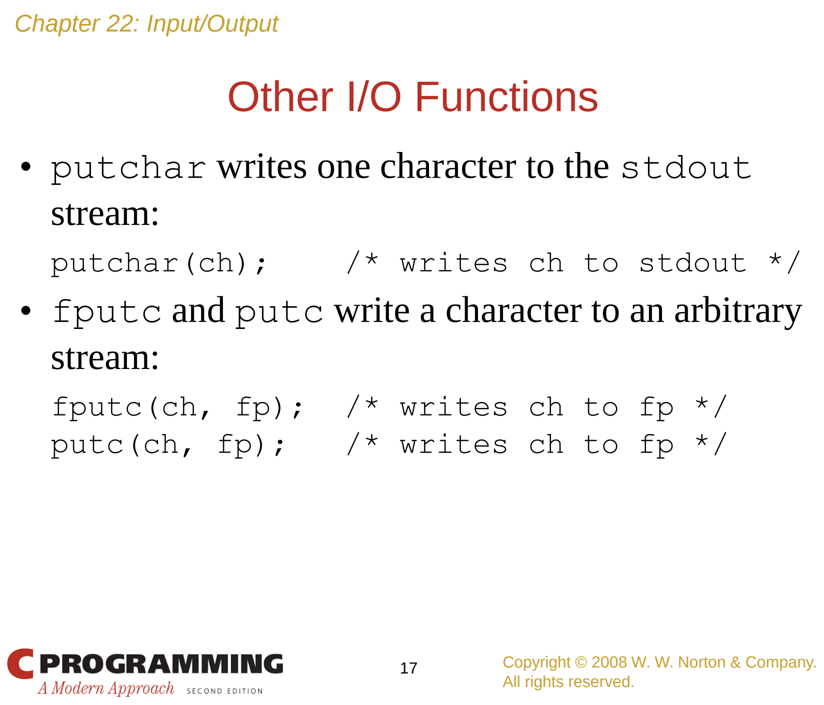


## The …scanf/fscanf Functions

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

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

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



## Other I/O Functions

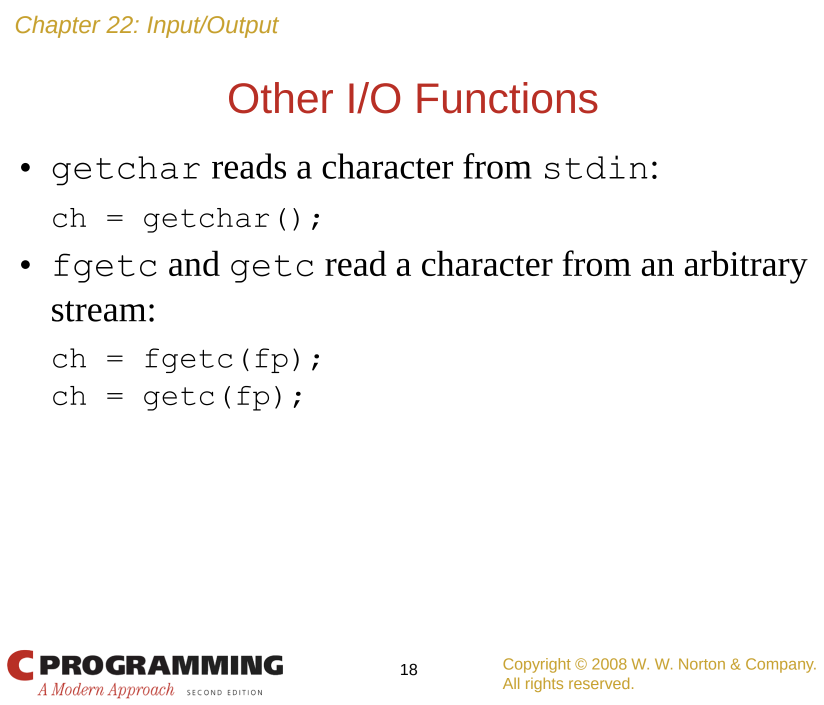
- `putchar` writes one character to the `stdout` stream:

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

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

```
fputc(ch, fp);   /* writes ch to fp */
putc(ch, fp);    /* writes ch to fp */
```

17



## Other I/O Functions

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

```
ch = getchar();
```

- `fgetc` and `getc` read a character from an arbitrary stream:

```
ch = fgetc(fp);
ch = getc(fp);
```

18

## Other I/O Functions

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

```
puts("Hi, there!");  /* writes to
stdout */
```

- After it writes the characters in the string, `puts` always adds a new-line character.
- `fputs` is a more general version of `puts`.
- Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);  /* writes to
fp */
```

- Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.

## Other I/O Functions

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

```
gets(str); /* reads a line from stdin */
```

- `gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).
- `fgets` is a more general version of `gets` that can read from any stream.

## Other I/O Functions

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

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

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

## Other I/O Functions

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

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