dard header file *stdio.h* on your system. Read this file and find the line that pertains to printf(). The line will look something like

```
int   printf(const char *format, ...);
```

This line is an example of a function prototype. Function prototypes tell the compiler the number and type of arguments that are expected to be passed to the function and the type of the value that is returned by the function. As we will see in later chapters, strings are of type "pointer to char," which is specified by char *. The identifier format is provided only for its mnemonic value to the programmer. The compiler disregards it. The function prototype for printf() could just as well have been written

```
int   printf(const char *, ...);
```

The keyword const tells the compiler that the string that gets passed as an argument should not be changed. The ellipses ... indicate to the compiler that the number and type of the remaining arguments vary. The printf() function returns as an int the number of characters transmitted, or a negative value if an error occurs. Recall that the first program in this chapter in Section 1.2, "Program Output," on page 6, prints the phrase "from sea to shining C" on the screen. Rewrite the program by replacing the #include line with the function prototype for printf() given above. *Caution:* You can try to use verbatim the line that you found in *stdio.h*, but your program may fail. (See Chapter 8, "The Preprocessor.")

15 (Suggested to us by Donald Knuth at Stanford University.) In the *running_sum* program in Section 1.6, "Flow of Control," on page 25, we first computed a sum and then divided by the number of summands to compute an average. The following program illustrates a better way to compute the average:

```
/* Compute a better average. */

#include <stdio.h>

int main(void)
{
  int     i;
  double  x;
  double  avg = 0.0;    /* a better average */
  double  navg;         /* a naive average */
  double  sum = 0.0;

  printf("%5s%17s%17s%17s\n%5s%17s%17s%17s\n\n",
    "Count", "Item", "Average", "Naive avg",
    "_____", "____", "_______", "______
```

```
  for (i = 1; scanf("%lf", &x) == 1; ++i) {
    avg += (x – avg) / i;
    sum += x;
    navg = sum / i;
    printf("%5d%17e%17e%17e\n", i, x, avg, navg);
  }
  return 0;
}
```

Run this program so that you understand its effects. Note that the better algorithm for computing the average is embodied in the line

```
avg += (x – avg) / i;
```

Explain why this algorithm does, in fact, compute the running average. *Hint:* Do some simple hand calculations first.

16 In the previous exercise we used the algorithm suggested to us by Donald Knuth to write a program that computes running averages. In this exercise we want to use that program to see what happens when sum gets too large to be represented in the machine. (See Section 3.6, “The Floating Types,” on page 119, for details about the values a double can hold.) Create a file, say *data*, and put the following numbers into it:

```
1e308  1  1e308  1  1e308
```

Run the program, redirecting the input so that the numbers in your file *data* get read in. Do you see the advantage of the better algorithm?

17 (Advanced) In this exercise you are to continue the work you did in the previous exercise. If you run the *better_average* program taking the input from a file that contains some *ordinary* numbers, then the average and the naive average seem to be identical. Find a situation where this is not the case. That is, demonstrate experimentally that the better average really is better, even when sum does not overflow.

18 Experiment with the type qualifier const. How does your compiler treat the following code?

```
const int   a = 0;

a = 333;
printf("%d\n", a);
```