UNIVERSITY OF EXETER

Physics and Astronomy

Unformatted and binary input and output

Comments and questions to John Rowe.

Recap: formatted input

The `scanf()` family of functions we have used so far are **intelligent** functions for situations where we know what to expect (for example, an integer) and want interpret the input accordingly.

They are extremely convenient for reading in numbers as they skip over white space, including new lines. The user can leave any number of spaces between inputs, or even just put one per line. This is referred to as **formatted** input as the function has to know the format of the data it is expecting (integer, floating-point number, text string without spaces, etc.)

After it has read the expected characters, the system leaves itself positioned at the next character after the last one it has used, which is nearly always a space or a new-line, '\n'.

An example

Consider a file that starts:

```
1 2
8.7
32.8
Mary had a little lamb
...
```

As far as our program is concerned, it is as if it were a giant character string starting:

`"1 2 \n8.7\n32.8\nMary had a little lamb\n..."`

Notice how I had accidentally typed a space at the end of the first line. This is quite common.

If the code now executes the statement:

```
fscanf(infile, "%d %d %lf", &j, &k, &x);
```

The value 1, 2 and 8.7 get read into j, k and x respectively with `fscanf()` conveniently skipping over unwanted spaces and new-line characters. Having read in everything up to and including "8.7" the imaginary character string now contains:

`"\n32.8\nMary had a little lamb\n..."`

You will notice the remaining "string" starts with a newline character, `'\n'` . This will be very important later on.

As long as we continue to use `fscanf()` to read in integer and numbers everything will be fine as all the unwanted spaces just get skipped over.

Unformatted input: reading in text without interpreting it

Quick discussion:

In this section we shall be dealing with situations where it is fairly easy to describe what we want to happen but actually making it happen involves some irritating and potentially confusing details.

Turn to your neighbour and ask:

- In situations such as these, what should be our instinctive reaction?.

Sometimes we just want to read in a whole line of text into a character array, referred to as **unformatted** input. We don't expect it to have any special form such as a number. The most common reason is to be able to input text containing spaces, for example people's names or free-form text for a notebook application.

We shall first look at the mechanics of reading in the text and then deal with the subtleties of combining formatted and unformatted input.

Unformatted input: fgets()

The **`fgets()`** function reads a line from a file without interpreting it. That is, `fgets()` reads the unread input up to and including the next new-line character. The "file" can be the keyboard if `stdin` (standard input) is used. It has the form:

```
fgets(buffer, maxbytes, file);
```

Here `file` is a `FILE *`. It can be obtained obtained in the usual way using `fopen()` or we could use the predefined value **`stdin`** if we want to read from standard input.

`buffer` is a character array at least `maxbytes` long. Like `snprintf()`, `fgets()` is "well behaved" and always puts a zero, '\0', at the end of the text even if the input line is too long, thus always leaving a valid zero-terminated string.

Thus `fgets()` reads in at most `maxbytes - 1` bytes of actual input, or up to the next new-line character, whichever comes first. It returns `NULL` if the read failed completely, for example if we have reached the end of the input file.

Notice that for unformatted input the `FILE` is the last argument, unlike in `fscanf()` where it is the first.

Removing the new-line character

If space permits, `fgets()` includes the new-line character, sent when the user presses the "Return" or "Enter" key. This is always the final character of the string. If we don't want this character, we just need to replace it with '\0', thus shortening the string by one.

The following snippet calls `fgets()` to read a line from standard input and then checks to see if the last character of a string stored inside a character array is '\n'. If so it replaces it with '\0', thus shortening the string by one.

```
if (fgets(line, N, stdin) != NULL) {
  int end = strlen(line) - 1;
  if (line[end] == '\n')
    line[end] = '\0';
}
```

If this were a program we were writing for other people to use we would need to consider what to do if the final character were **not** a new line as it probably indicates that the line was too long for out buffer.

With this we can look at a very short program that uses `fgets()` to read a line of text from the keyboard and print it out again:

Mixing formatted and unformatted input

Whether we have a file of data or are reading from the keyboard, we are always free to mix formatted and unformatted input.

A common situation is to use formatted input to get options from a menu, or to read in data values, and to then need to read in a complete line of text.

The first attempt often looks like this:

```
//
// Flawed attempt to read in an integer followed by some text.
//
int main() {
  char line[N];
  int value;

  printf("Please enter the integer value\n");
  scanf("%d", &value);

  printf("Now please type in the text string, spaces are allowed!\n");

  if (fgets(line, N, stdin) != NULL) {
    // Chop off final '\n';
    int end;
    end = strlen(line) - 1;
    if (line[end] == '\n')
      line[end] = '\0';
    printf("The value is %d, the text is >%s<\n", value, line);
  }

  return 0;
}
```

The "conversation" goes like this:

```
Please enter the integer value
12
Now please type in the text string, spaces are allowed!
The value is 12, the text is ><
```

The user is given no chance to type in a line of text, instead `fgets()` just seems to read a completely blank line. What's happening?

The answer is that, when we typed in "12" we actually typed **three** characters, '1', '2' and the carriage return, '\n'. As in the previous example, the system reads in the two characters '1' and '2' that form the integer 12 and leaves itself positioned at the very next character, which is the new line '\n'. Thus the "next line" is completely empty!

We can illustrate this by typing at the keyboard not just "12<return>" (three keystrokes) but "12<space>abc<return>" (seven keystrokes). The final line of output now looks like this:

```
The value is 12, the text is > abc<
```

Recap: `scanf()` is reading the two characters '1' and '2', inspecting the next character, seeing it is white-space, which ends the number and is therefore ignored and **left for the next input function to deal with**. If that next function is another call to `scanf()` that's fine as `scanf()` skips over white-space. But `fgets()` doesn't so it just sees the new-line character which it treats as being an empty line.

There are various bad solutions at this point (some people's first reaction is just to read in one more character in the hope that nobody will ever type `12<space><return>`), but the most common situation is that we require the input line to be non-blank. In this case it's easy to write a loop that carries on reading a line from the file, or keyboard, until it finds a line that contains a non-space character.

If we are reading from `stdin` it looks like this:

```
int readoneline(char line[], int maxbytes) {
  while ( 1 ) {
    int i;

    if ( fgets(line, maxbytes, stdin) == NULL ) {
      return 0;   // Out of data
    }

    // We don't the new-line character so chop it
    i = strlen(line) - 1;
    if ( line[i] == '\n')
      line[i] = '\0';

    // Look for a non-blank character.
    for (i = 0; line[i]; ++i)
      if ( isspace(line[i]) == 0) {
        return 1;
      }
  }
}
```

Step through a complete example

This is quite a useful function.

Optional: handling bad input

This is an **advanced topic** and can be omitted if desired.

So far we have handled checking the values of numbers typed in at the keyboard by enclosing the call to `scanf()` inside an infinite loop, checking the values typed in and printing an error message if they are incorrect or `break`ing out of the loop if they are OK.

But you may already have encountered the situation where you have typed a non-numeric character by mistake, say 'q' instead of '1'. The problem is that in this situation `scanf()` leaves the input at the first character that doesn't match what it expects, the 'q'. The loop, if it's properly written, does not break so `scanf()` is called again and does exactly the same thing again: it stops at the 'q'. And so on for ever.

This might be thought to be an unhelpful response, but the question arises "what should the system do in this situation?".

Quick discussion

Turn to your neighbour and ask what would be the best thing to do in this situation.

One possible approach

The general idea here is to print a helpful message to the screen and skip the rest of the line. This only makes sense if we are reading from the keyboard there is no point in doing this if we are reading from a file.

Even here there are a few subtleties, for example: what happens if we reach the end of the input? This could be because `stdin` is coming from a file not the keyboard, or because the terminal window has closed. Or the input may be coming over the network and the connection might break. Thankfully we can tell when this happens as `scanf()` returns the special value **`EOF`**, usually equal to -1 and certainly negative. Note the distinction: `scanf()` returns zero when there was data but the wrong sort, such as a letter for a format of `%d`, and `EOF` when there is no data at all.

With that in mind we can write an error handling function which we shall call `skipline()`:

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

void skipline(int num);

int main() {
  int x, y;
  int got;

  while (1 == 1) {
    printf("Please enter two integers > 0\n");

    if ((got = scanf("%d %d", &x, &y)) != 2)
      skipline(got);
    else if ( x <= 0 || y <= 0 )
      printf("Only integers greater than zero are allowed\n");
    else
      break;

    printf("\n\tPlease try again.\n\n");
  }

  printf("Read: %d %d\n", x, y);
  return 0;
}

//
// Read and discard the rest of the line from stdin, printing it so
// the user knows what's going on. If num == EOF exit
//
void skipline(int num) {
  int i;

  if ( num == EOF ) {
    printf("End of standard input\n");
    exit(1);
  }

  // Not EOF - skip the rest of the line
  printf("\nSkipping unexpected input: ");
  while ((i = getchar()) != '\n') {
    if ( i == EOF ) {
      printf("End of standard input\n");
      exit(1);
    }
    putchar(i);
  }

  putchar('\n');
}
```

getchar() and putchar()

`getchar()` reads a single character from standard input (there is a version `getc(file)` to read from a file).

If you look closely at the code above you will see it returns an `int`, not a `char` as we might expect. The reason is the one we mentioned above: we need a way of telling if we have run out of input. All the possible values of a `char` (including zero) are by definition possible successful values of `getchar()`.

So on failure `getchar()` returns `EOF` which is not a possible `char`.

Example

If we want to read in a `char` variable, let's call it `c`, we might write:

```
char c;
int i;

if ((i = getchar()) == EOF ) {
  fprintf(stderr, "Out of data!\n");
  exit(99);
}
/* Else */
c = i;  /* Success */
```

Similarly `putchar(int value)` and `putc(int value, FILE *)` print a single character to `stdout` or a file respectively.

With that in mind `skipline()` should be reasonably clear. It's important to notice that we have put all of the nastiness of checking for the end of file **inside** of `skipline()`, we have not left any of it for the calling function to handle.

Dealing with errors is always tedious and should be separated from the main logic of the code as far as possible.

Optional: binary input and otput of numerical data

Sometimes we require a program to be able to save data to a file but we know the file will only ever be read by another program. For example, long-running numerical simulations often periodically save their state so they can be stopped and restarted from their last saved state.

Considering the example of a two-dimensional matrix, the obvious way to do it is to use a double loop and `fprintf()`:

```
for (int m = 0; m < M; ++m)
   for (int n = 0; n < N; ++n)
     fprintf(datafile, "%g\n", x[m][n]);
```

and later read it in with:

```
for (int m = 0; m < M; ++m)
   for (int n = 0; n < N; ++n)
     fscanf(datafile, "%lg", &y[m][n]);
```

However this is inefficient:

- The program is having to take eight bytes of binary data, translate it to human-readable form and then translate it back again.

- We are losing precision.
- The file is bigger.

We can improve the precision, at the cost of a larger file size, by writing using a format such as `"%.10g"` which means "use ten decimal places". But the data we read in still won't be exactly the same as we wrote out and that's a problem.

Binary input and output

C gives us the ability to write the raw bytes to a file, in this case (assuming a true 2-D array) with a single function call:

```
fwrite(x, sizeof x[0][0], N*M, datafile);
```

This means: write `N*M` chunks of data each of the size of `x[0][0]` from `x` into `datafile`.

The corresponding "read" call is:

```
fread(y, sizeof y[0][0], N*M, datafile);
```

Now `y` is identical to `x` as we have just copied the bytes to and from the disk.

The complete program

```
/*
 * Unformatted store and read of a matrix.
 */

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

#define N 100
#define M 200

int main() {
  double x[M][N], y[M][N], error = 0.0;
  FILE *datafile;

  for (int m = 0; m < M; ++m)
    for (int n = 0; n < N; ++n)
      x[m][n] = sin(0.01 * n + 0.0153 * m);

  // Write the data
  if ((datafile = fopen("Bindat.dat", "w")) == NULL ) {
    printf("Cannot create file\n");
    exit(1);
  }

  fwrite(x, sizeof x[0][0], N*M, datafile);
  fclose(datafile);

  // Now read it back in
  if ((datafile = fopen("Bindat.dat", "r")) == NULL ) {
    printf("Cannot read file\n");
    exit(1);
  }

  fread(y, sizeof y[0][0], N*M, datafile);
  fclose(datafile);

  // Now compare
  for (int m = 0; m < M; ++m)
    for (int n = 0; n < N; ++n)
      error += fabs(y[m][n] - x[m][n]);

   printf("Total error is: %g\n", error);
   return 0;
}
```

The error is zero (precisely) as we have read back **exactly the same bytes** as we wrote.

Two gotchas

Partial and/or pseudo-arrays

The "all in one go" approach above writes the whole array, but in some circumstances we may only be using a part of the whole array. For example we may be going from x[0][0] to x[m][n] where m<=M and n<=N. The solution is to write each row separately using a loop:

```
for (int j = 0; j < m; ++j)
  fwrite(x[j], sizeof x[j][0], n, datafile);
```

Alternatively, suppose the `x[m][n]` were not part or a true array but a dynamically-allocated psudo-array using `malloc()`:

```
double **x;

x = xmalloc(m * sizeof *x);
for (int j = 0; j < m; ++j)
  x[j] = xmalloc(n * sizeof *x[j]);
```

Then `x` is a (dynamically allocated) array of pointers, not a 2-D array of doubles. The above code, writing each row separately, works in this case as well.

MS Windows

For horrible historical reasons MS Windows distinguishes between text and binary files so it is recommended you open binary files with `"wb"` and `"rb"`:

```
fopen("Bindat.dat", "wb") // MS Windows only
```
