Reading a File | Coursera

Reading a File

Reading a File

Once we have our file open, we might want to read input from it. We typically will use one of three functions to do so: **fgetc**, **fgets**, or **fread**. Of these, **fgetc** is useful when you want to read one character (*e.g.*, letter) at a time. This function has the following prototype:

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

While you might expect this function to return a char, it returns an intso that it can return all possible chars, plus a distinct value to indicate that there are no more characters available in the stream—that the end of the file (*EOF*) has been reached. The value for end-of-file is *defined* as the constant **EOF** in **stdio.h**. Note that reading the character advances the current position in the stream.

The fact that function which read from streams advance the position of the stream poses a minor annoyance when writing a loop. Consider the following (broken) code which attempts to print every character from an input file:

```
//broken
FILE * f = fopen(inputfilename, "r");
if (f == NULL) { /* error handling code omitted */ }
while (fgetc(f) != EOF) {
  char c = fgetc(f);
  printf("%c",c);
}
//...other code...
```

This code will read one character, check if it is the end of the file, then read *a different* character, and print it. This code will actually print every other character from the input file, plus possibly something spurious at the end (if there are an odd number of characters in the file).

The cleanest way to re-structure the loop is to exploit the fact that an assignment is also an expression which evaluates to the value that is assigned. While that may sound like a technically complex mouthful, what it means is that **x = 3** is not only an assignment of 3 to x, but also an expression which evaluates to 3. We could therefore write **y = x = 3** to assign 3 to both x and y—however, we typically do not do so, as it makes the code less clear than writing two assignment statements. In this particular case, however, it is OK to exploit this property of assignments, and is in fact a common idiom. The following code correctly prints every character from the input file:

```
//fixed
FILE * f = fopen(inputfilename, "r");
```

```
if (f == NULL) { /* error handling code omitted */ }

int c;

while ( (c=fgetc(f)) != EOF ) {

  printf("%c",c);

}

//...other code...
```

Observe how we assign the result of **fgetc** to the variable c in the while loop's conditional expression. We then wrap that assignment statement in parenthesis to ensure the correct order of operations, and compare the value which was assigned (whatever fgetc returned) to EOF.

You may have noticed that the type of c is **int**, not **char**. If we declared c as a char, our program would have a rather subtle bug. Can you spot it? Remember that we said **fgetc**returns an intso that it can return any possible character value read from the file, plus some distinct value forEOF. Assigning this return value to a char then inherently discards information—we are taking N + 1 possible values and assigning them to something that can hold N different bit patterns (in the case of char, N = 256). On most systems EOF is −1, so in this particular case, we would not be able to distinguish between reading character number 255 and the end of the file—if our input had character number 255 in it, our program would prematurely exit the loop, and ignore the rest of the input! You should aim to think of these sorts of corner cases when you write test cases.

✓

Completed