C Language text I/O summary – The rules of the game

(G.Cabodi)

C language text I/O instructions (needing the inclusion of ***`stdio.h`***) can be broadly classified in three groups:

- ***character***
- ***string***
- ***formatted***

Let's briefly overview the three groups, taking into account that formatted I/O with the `%c` format specifier can be considered as equivalent to ***character*** based I/O, and formatted I/O with the `%s` specifier as equivalent to ***string*** based I/O.
In all groups, function names starting with f will have a file pointer parameter, whereas their counterparts without the initial f, implicitly work on `stdin` or `stdout`.
Whenever working with files, we'll use **`fin/fout`** as file pointers (`FILE *fin, *fout;`), and we'll omit **`fopen/fclose`** instructions.

Character I/O

Whenever you need to use character I/O, you have redundant choices. Some programmers prefer the %c option (formatted I/O with %c), and they seldom use putc/getc/…, others do exactly the opposite. Whereas formatted I/O basically subsumes putc/getc/… (and offers more), putc/getc/… can be a bit more efficient/faster. For a minimal C subset, go with %c.

Output

Given a char variable `c` (`char c;`), the following instructions are equivalent

```
fputc(c,fout);
putc(c,fout);
fprintf(fout,"%c", c);
```

`fputc` and `putc` are *mostly* equivalent, the latter one being potentially faster if implemented as a macro (*WARNING: you thus need to know how to work with macros, for non trivial usage*).

stdout

`putchar(c)` is equivalent to `putc(c,stdout)`.
`printf(...` is equivalent to `fprintf(stdout,...`

Examples

```
// write capital alphabetic chars to fout using fputc
for (c='A'; c<='Z'; c++)
  fputc(c,fout);

// write chars of string s using fprintf
int len = strlen(s);
for (i=0; i<len; i++)
  fprintf(fout,"%c", s[i]);
```

Input

Given a char variable `c` (`char c;`), the following instructions are equivalent

```
c = fgetc(fin);
c = getc(fin);
fscanf(fin,"%c", &c);
```

The difference between `fgetc` and `getc` is similar to the one already mentioned for `fputc/putc`.

stdin

`getchar()` is equivalent to `getc(stdin)`.
`scanf(...` is equivalent to `fscanf(stdin,...`

Examples

```
// read a file – print chars on stdout – use EOF for end-of-file check
int endOdFileFound = 0;
while (!endOfFileFound) {
  c = fgetc(fin);
  if (c!=EOF) {
    putchar(c);
  }
  else {
    endOfFileFound = 1;
  }
}

// same without flag
c = ’a’; // any char is OK
while (c!=EOF) {
  c = fgetc(fin);
  if (c!=EOF) {
    putchar(c);
  }
}

// same using do-while
do {
  c = fgetc(fin);
  if (c!=EOF) {
    putchar(c);
  }
} while (c!=EOF);

// same – more compact
while ((c = fgetc(fin))!=EOF) {
  putchar(c);
}
// copy fin to fout, use feof() function for end-of-file check
while (!feof(fin)) {
  fscanf(fin,”%c”, &c);
  if (!feof(fin))
    fprintf(fout,”%c”, c);
```

```
}

// same, with fgetc/fputc and flag
int again = 1;
while (again) {
  c = fgetc(fin);
  if (feof(fin))
    again = 0;
  else
    fputc(c,fout);
}
```

String I/O

Whenever you need to do string I/O, you have redundant choices: formatted I/O with %s, or fgets/fputs/gets/puts. Whereas the two options are fully equivalent for output, they are not for input. So for minimal C go with %s for output (omit puts/fputs) and understand the difference between %s and fgets/gets.

Output

Given a string variable `s` (`char s[LEN];`), the following instructions are equivalent

```
fputs(s,fout);
fprintf(fout,"%s", s);
```

stdout

The `puts` function is specific to `stdout`:

```
puts(s);
```

is equivalent to

```
fputs(s,stdout); fputc('\n',stdout);
```

or to

```
fprintf(stdout,"%s\n",s);
printf("%s\n",s);
```

as `puts` automatically appends a newline character ('\n') at the end of the string.
`printf(...` is equivalent to `fprintf(stdout,...`

Examples

```
/* write an array of N strings to fout using fprintf
   strings are considered as lines (they can include blanks)
   newline added after each string */
for (i=0; i<N; i++) {
  fprintf("%s\n", strings[i]);
}

// same but strings are considered words (without blanks)
// blanks added between strings, newline every M strings
for (i=0; i<N; i++) {
```

```
  fprintf("%s", strings[i]);
  if ((i+1)%M == 0)
    printf(" ");
  else
    printf("\n");
}
```

Input

Use `fgets` (or `gets`) when reading lines, use `fscanf/scanf` with %s format specifier when reading words (blank/newline separated strings, NOT containing blanks).

Given a string variable `s` (`char s[LEN];`), the following instructions

```
fgets(s,LEN,fout);
fscanf(fout,"%s", s); // EXCEPTION: NO & BEFORE s !!!
```

Are equivalent ONLY IF no whitespaces (e.g. blanks) are included in the string, and the string has length < LEN). The `fgets` function reads all characters until the newline, and stops after LEN-1 characters (the remaining characters are available for next input instructions).

The `gets` function is specific to `stdin`. It is often discouraged, as no limit on the number of characters is available: one could input a line of text longer than the size of the destination array.

`stdin`

```
gets(s);
```

is (almost) equivalent to

```
fgets(s,stdin);
```

BUT: `fgets` KEEPS newline characters, `gets` filters (removes) them out, and (repetition) no limit on the number of character is available with `gets`.

Examples

```
// read a file – print lines on stdout
char s[LEN];
while (fgets(s,LEN,fin)!=NULL) {
  fputs(s,stdout);  // or printf("%s",s);
}

// read a file – print words on stdout – blanks and newlines are missed
char s[LEN];
```

```
while (fscanf(fin,“%s”,s)!=EOF) {
  printf(fout,”%s\n”,s); // newline after each word
}

// copy fin to fout, use feof() function for end-of-file check
while (!feof(fin)) {
  fgets(s,LEN,fin);
  if (!feof(fin))
    fprintf(fout,”%s”, s);
}
```

Formatted I/O

Output is definitely simpler. With input, one should be careful about possibly missing synchronization between input statements and input stream.
The %c and %s format specifiers have already been (partially) explained, as well as the relations between scanf/fscanf and printf/fprintf.

Output

```
fprintf(fout,”this is a string”);
```

means ***“print the string constant to file fout”***, which is fully equivalent to

```
fputs(“this is a string”,fout);
```

Single or multiple newlines are possible.

Examples

```
fprintf(fout,”first line\nsecond line\nthird line”); // three lines printed

// same result with three statements
fprintf(fout,”first line\n”);
fprintf(fout,”second line\n”);
fprintf(fout,”third line\n”);
```

Format specifiers are possible: `%c`, `%s`, `%d`, `%f`, and `%g` for double. Let’s focus on `%d` and `%f`. For each format specifier within the constant string, an additional parameter is needed, representing a variable (or an expression) corresponding to an integer (`%d`) or a float (`%f`) value, that will be converted to the proper output *“decimal”* encoding (a sequence of output characters.

Examples

```
// Print integer numbers from 0 to 99 by rows of 10
for (i=0; i<100; i++) {
  printf(“%d”, i);
  if ((i+1)%10 == 0) // every 10 numbers
    printf(“\n”); // newline
  else
    printf(“ “); // blank
}
```

```
// Same with two nested loops
for (d=0; d<10; d++) {
  for (u=0; u<10; u++) {
    printf("%d ", 10*d+u); // we accept an extra blank before the newline
  }
  printf("\n"); // newline
}

// Same with one printf for each row
for (i=0; i<100; i+=10) { // increment by 10
  printf("%d %d %d %d %d %d %d %d %d %d\n",
         i,i+1,i+2,i+3,i+4,i+5,i+6,i+7,i+8,i+9);
}
```

Format specifiers allow extra fields (se here: https://en.wikipedia.org/wiki/Printf_format_string, or other equivalent documents, for a complete list). Let us only mention here **width** (possibly combined with the minus (-) flag) and **precision** fields

- **width**: formats %5d, %10f mean use (at least, more just if needed) a 5 characters field for an integer, and a 10 character s field for a float number. Numbers are right-aligned. For left alignment use %-5d, %-10f instead.
- **precision**: though working also with strings (%s), let's limit to float numbers; precision specifies how many digits to use to the right of the decimal point. So %10.2f means use a 10 character fiend, with 2 digits reserved to the decimal/fractional part (the digits after the decimal point)

Example

```
// numbers from 0 to 99 are now well aligned
for (i=0; i<100; i++) {
  printf("%3d", i);
  if ((i+1)%10 == 0) // every 10 numbers
    printf("\n"); // newline
}
```

Input

```
fscanf(fin,"this is a string");
```

means ***"I expect to EXACTLY read the string constant from fin"***, which is not so meaningful, as a program statement, but tells you what the parameter string means: so if you expect the user to type "YES", you could write

```
scanf("YES");
```

A good full description of the function behaviour can be found here:
http://www.cplusplus.com/reference/cstdio/fscanf/

WARNING: input will stop at the first input character that does not match the input string. The remaining input characters will still be there for the next input instruction (so possibly missing the synchronization between program and input stream!).
For instance, providing as input "Yeah", to the previous statement (`scanf("YES");`), will fail on the 'e' character, so the next input statement will resume from "eah".

So the following instructions

```
char s[LEN];
printf("Please type YES: ");
scanf("YES");
printf("Now type a string: ");
scanf("%s", s);
printf("The string is: %s\n");
```

will for instance end up printing

```
The final string is: Hello
```

If you type on keyboard (`stdin`):
```
Please type YES: YES
Now type a string: Hello
```

But in case you wrongly type:
```
Please type YES: Yeah
```

The output will end up with:
```
Now type a string: The final string is: eah
```

You will not be able to type Hello, as you already typed "eah", captured by the `scanf("%s", s);` statement. So you now should understand what synchronization means.

Whitespace characters

the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters: they are all characters for which function isspace returns true).

So the instructions

```
scanf(" ");
scanf("\n");
```

are fully equivalent and they will accept no whitespace, one or more whitespaces.
WARNING: In order to move to the next instruction, you need to provide at least a NON-WHITESPACE character, so that the instruction knows that whitespaces have ended. This is the reason why whitespaces are typically "in the middle" rather than "at the end of the format string.

Examples

```
fscanf(fin,” |%d|”, &n);
```

The previous statement reads an integer between two ’|’, preceded by an arbitrary sequence of whitespaces (e.g. blanks/newlines).

Format specifiers

Input format strings accept the same format specifiers used for output, so %c, %s, %d, %f (and %g for double). Extra fields are usually less meaningful for input than for output.
WARNING: all formats EXCEPT %c (so %s, %d, %f, %g) skip ALL WHITESPACE CHARACTERS BEFORE reading the matching input characters.

Examples:

```
/* The following statements are equivalent: they read 4 integer numbers,
separated by any sequence of either blanks or newlines */
fscanf(fin,”%d%d%d%d”, &a,&b,&c,&d);
fscanf(fin,”%d %d %d %d”, &a,&b,&c,&d);
fscanf(fin,” %d  %d\n%d%d”, &a,&b,&c,&d);

/* read name, surname (separated by spaces) and birth date
   in day/month/year format */
fscanf(fin,”%s%s%d/%d/%d”,name,surname,&d,&m,&y);
// same with spaces more explicit
fscanf(fin,”%s %s %d/%d/%d”,name,surname,&d,&m,&y);
```