CSCI 2132: Software Development

Formatted I/O in C

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Overview

Formatted output: `printf`

Formatted input: `scanf`

To use: `#include <stdio.h>`

Overview

Formatted output: `printf`

Formatted input: `scanf`

To use: `#include <stdio.h>`

General format:

```
printf(format_string, expr1, expr2, ... )
```

- `format_string` contains one conversion specification (`% ...`) per expression

Overview

Formatted output: `printf`

Formatted input: `scanf`

To use: `#include <stdio.h>`

General format:

```
printf(format_string, expr1, expr2, ... )
```

- `format_string` contains one conversion specification (`%` ...) per expression

Example:

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

How `printf` Works

- `printf` reads the format string and prints it
- For each conversion specification it encounters, it interprets the next expression as the specified type and prints it.

Warning:

- Mismatches between conversion specifier and parameter type may go undetected.
- (Modern C compilers seem to catch them.)

printf Example

```
int i = 7;
double x = 2.71;
char c = ‘A’;
printf(“i = %d, x = %.2f, c = %c\n”, i, x, c);
```

Output:

```
i = 7, x = 2.71, c = A
```

Conversion Specifiers

Conversion specifiers start with %

In its basic form, each specifier states the type of expression it expects:

- %d: integer
- %u: unsigned integer
- %f: floating point number (float)
- %s: string
- %c: character
- %%: literal “%”

Conversion Modifiers

Full format of conversion specifier:

```
%[flags][minw].[prec][lenm]spec
```

- `spec`: specifier (`%d`, `%f`, ...)
- `lenm`: length modifier (e.g., `%lf` double instead of `%f` float)
- `prec`: precision (number of digits after decimal point, `%.2f`)
- `minw`: minimum width (e.g., `%10d`)
- `flags`: (`%+d` mandatory sign, `%0d` pad with zeroes, ...)

More info: `man 3 printf`

Formatted Input: `scanf`

Format similar to `printf`:

```
scanf(format_string, addr1, addr2, ... )
```

- `scanf` reads the format string and `stdin` and matches them
- On each conversion specification, input is interpreted as given type and stored in next memory address.
- If matching fails, `scanf` stops reading

Return value: number of converted values or EOF in some cases

More information: `man 3 scanf`

scanf Example

```
int i, j;
double x, y;
scanf(“%d%d%lf%lf”, &i, &j, &x, &y);
```

Possible input:

```
1 -20 .3 -4.0e3
```

Why do we need the &?

scanf Example

```
int i, j;
double x, y;
scanf(“%d%d%lf%lf”, &i, &j, &x, &y);
```

Possible input:

```
1 -20 .3 -4.0e3
```

Why do we need the &?

- C passes all arguments by value
- scanf needs addresses (memory locations) where to store read values
- & is the “take address” operator

`scanf` Matching Procedure

- White space is matched with white space or nothing.
- Characters other than conversion specifiers match themselves.

Conversion specifiers:

- `%d`, `%u`, `%s`, `%c`, `%f`, `%%` (numeric conversion skips leading whitespace)
- `%[0-9]`, `%[^A-Z]` similar to wildcards, matches arbitrary number of occurrences of the given characters, store as string
- `%n`: no matching, store number of characters consumed so far

Modifiers:

- `*`: Parse the value but don't store it
- `l`: `float` vs `double`, `int` vs `long int`

A scanf Example

```
int n, i, j;
double x, y;
n = scanf("%d%d%lf%lf", &i, &j, &x, &y);
printf("n = %d\
printf("i = %d                 , j);
printf("x = %.2l               f\n", x, y
```

scanf skips whitespace

scanf reads as far as it can

```
     1
-20    .3
   -4.0e3
```

```
1-20.3-4.0e3
```

```
n = 4
i = 1, j = -20
x = 0.30, y = -4000.00
```

```
n = 4
i = 1, j = -20
x = 0.30, y = -4000.00
```

A `scanf` Example

```
int n, i, j;
double x, y;
n = scanf("%d%d%lf%lf", &i, &j, &x, &y);
printf("n = %d\[illegible]
printf("i = %d[illegible] j);
printf("x = %.2l[illegible]f\n", x, y[illegible]
```

scanf reads as far as it can

errors result in reading fewer values

```
1 -20.3 -4.5 5.5
```

```
1.1 -20 -4.5 .5
```

```
n = 4
i = 1, j = -20
x = 0.30, y = -4.50
```

```
n = 1
i = 1, j = <junk>
x = <junk>, y = <junk>
```

Some Finer Points about `scanf`

Are the following two `scanf` statements equivalent?

```
int i; double x;
scanf("%d %lf", &i, &x);
scanf("%d%lf", &i, &x);
```

Yes and no:

- They both succeed on the same inputs and assign the same values to `i` and `x`.
- The spaces between the values assigned to `i` and `x` are skipped by
  - Matching the space between `%d` and `%lf`
  - Skipping whitespace when matching `%lf`

Some Finer Points about `scanf`

Are the following two `scanf` statements equivalent?

```
int i;
scanf(“%d”, &i);
scanf(“%d ”, &i);
```

No:

- The first reads an integer, possibly skipping leading whitespace.
- The second consumes as many spaces after the read number as possible.
    - You need to enter a non-whitespace to make it finish.

Some Finer Points about `scanf`

Are the following two `scanf` statements equivalent?

```
double x, y;
scanf(“%lf,%lf”, &x, &y);
scanf(“%lf ,%lf”, &x, &y);
```

No:

- The first fails if there are spaces between the first number and the comma.
- The second succeeds no matter how many whitespaces surround the numbers.

A Slightly Larger Example

Specification:

- Print "`Enter expression: `".
- Accept input in the form "`a/b + c/d`" with arbitrary spacing around the numbers.
- Output "`e/f`" where "`e/f = a/b + c/d`" (no need to simplify).

Solution

```
#include <stdio.h>

int main() {
  int a, b, c, d, e, f;

  printf("Enter expression: ");
  scanf("%d / %d + %d / %d", &a, &b, &c, &d);

  f = b * d;
  e = a * d + c * b;

  printf("%d/%d\n", e, f);

  return 0;
}
```