Faculty of Computer Science, Dalhousie University *24-Sep-2018*

CSCI 2132 — Software Development

Lecture 9: Formatted Input and Output in C

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Extended regular expressions (ERE)
- Grep variations
- Introduction to C
    - writing a simple program
    - compilation process
    - C basics, similarities and differences with Java
    - printing output and reading input
    - a simple HST program

9 Formatted Input and Output in C

- Necessary to use `#include <stdio.h>`
- Main functions: `printf` (output), and `scanf` (input)

9.1 Formatted Output: printf

We have already seen that we can print to the standard output in C using a library function called `printf`. The following is the syntax of printf:

```
printf(format_string, exp1, exp2, ...);
```

Here a `format_string` is a C string that may contain conversion specifications. We will learn in more general terms what is a C string later, but for now we can think of it as a string literal, i.e., a string given between two double-quotes (`"` and `"`). A conversion specification is a place holder indicating where the value of an expression is to be filled in during printing, and in what format this value will be printed. For example, the following piece of code:

```
int i = 7;
double x = 2.71;
char c = 'A';
printf("Printing vars: i = %d, x = %.2f, c = %c\n", i, x, c);
```

would produce:

```
Printing vars: i = 7, x = 2.71, c = A
```

We can see that the conversion specifications ‘`%d`’, ‘`%.2f`’, and ‘`%c`’ are replaced with the values of the expressions `i`, `x`, and `c`.

Generally, a conversion specification starts with character `%` and ends with a conversion specifier. The printf function has an elaborate format for conversion specifications, which can be briefly explained as follows:

- `%` character starts the specification, and it is mandatory.
- *flags* are optional characters, and there could be zero or more flags, such as + in `%+d`, specifying a mandatory sign when printing a number, or `-` specifying left justification.
- *minimal width* is an optional positive integer specifying minimal width of the printout, as in `%10d`, which may be useful in aligning numbers in a table.
- `.`*precision* is an optional period (`.`) followed by an integer (negative integer has the same effect as zero), which is the number of decimal places to be printed for floating point numbers, but also minimal number of digits for integers, and maximal number of characters for strings.
- *length modifier* is an optional character or two, such as `l` used for integers to denote a long integer, or it can be used with ‘`f`’ as ‘`lf`’ to denote double instead of float. However, this particular modification is not necessary.
- *conversion specifier* is the final mandatory character, denoting how the value of the variable is to be converted to the printed string representation. The most frequently used ones are:

`d`	for an integer,
`f`	for a double or a float,
`c`	for a character,
`s`	for a string, and
`%`	for a literal percent sign.

As in any string literal, as we will see, we can use escape sequences, such as ‘`\n`’ for a new line, ‘`\t`’ for a tab character, ‘`\\`’ for a backslash itself, and some others. You can use the command `man 3 printf` on bluenose to read about the `printf` function in more details.

9.2 Formatted Input: scanf

The function scanf is used to read the standard input according to a specific format.

The `scanf` function is a function for reading input, which, similarly to `printf` uses a format string to specify how is input going to be converted to the values of given variables. The function `scanf` also uses a format string. Let us look at the following example:

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

When this code is executed, the `scanf` function will wait for the user's input, and when the user types four values for the variables `i`, `j`, `x`, and `y`; e.g.,
`1 -20 .3 -4.0e3`
the corresponding values will be stored and the execution of the program will continue. The symbol & is required here, and forgetting it is a common beginner's error. It is usually required, but not in all situations, such as when the variable is a string. The symbol `&` is an operator and its meaning will become more clear once we introduce pointers. For now, we just say that the `scanf` function expect memory addresses where it can store read values, and those addresses of variables are provided by the `&` operator in front of a variable name. We will just assume for now that we need always to use this operator.

The way scanf works is very much like a pattern-matching process. The scanf function processes the format string from left to right. For each character of the string or conversion specification, the scanf will do an appropriate matching with the input, and for every conversion specification, it will match an appropriate input, convert it, and store it in the memory at the provided address. If there is a mismatch during this matching process, the scanf function will stop and return without assigning all values. The scanf function generally returns the number of successfully converted values, and that is how we can know how successful the process was.

`scanf` Conversion Specifications

These are the general rules of how `scanf` works:

– *A sequence of white-space characters* (space, tab, newline, and some others) is matched with any sequence of white-space characters from input, including none.
– Any character other than `%` must be matched with itself in the input.
– The character `%` denotes a beginning of a conversion specification. The input is read according to this specification and the converted value is stored in the next address given in the argument list. All conversions will first skip any white-space characters from input, unless explicitly stated below. The percent character may be optionally followed by some optional modifiers, such as:
  - `*` which causes converted value to be discarded and not assigned to the next parameter;
  - *integer* which limits maximal number of characters to be converted from the input; and
  - `l` which may be used for integers for the `long int` values, or for floating-point numbers for the `double` values.

  Finally, the conversion specifier comes at the end, and the following are mostly used ones:
  - `%` matches a literal '`%`' without conversion.
  - `d` matches a decimal integer and converts it to an `int`.
  - `f` matches a floating-point number, and stores it into a `float` or if we use modifier `l` stores it into a `double`.
  - `s` matches a sequence of non-white-space characters and stores it into a string (we will learn later about strings).
  - `c` matches a sequence of characters whose length is optionally given before `c` (default is 1) and stores it into a string. This specifier does not ignore white space.
  - `[` matches a non-empty sequence of acceptable characters and stores them into a string. Similarly to regular expressions and wildcards, acceptable characters can be listed, like in `[apm]`, we can use ranges, as in `[a-zA-Z]`, and we can list unacceptable characters using `^`, as in `[^a-z]`.
  - `n` does not match anything. Instead, it simply stores number of characters consumed so far by scanf in the next `int` argument.

You can read more about `scanf` using the command '`man 3 scanf`'.

For example, we have the following piece of code:

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

If the user enters the following three lines of input:

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

by the description of the above pattern matching process, these four values will be assigned to variables correctly, even though there are multiple space characters and newline characters between them. The final newline will be put back to the input sequence and be the 1st character read by the next scanf.

If the user enters the following line of input for the same code snippet:

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

Then the values of i, j, x and y after this will be 1, -20, .3, and $-4.0e3 = -4000$, respectively. If the user enters

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

Then the values of i, j, x and y after this will be 1, -20, .3, and -4.5, respectively. The 5.5 (including the space character before it) will be left for the next scanf. If the user inputs

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

Then after assigning 1 to i, `scanf` sees a dot, and it returns without processing the rest of the format string and reading more input.