



Lecture 08: Formatted Input/Output

Marshall McMullen
Computer Science Department
University of Arizona



Outline

- C and Input/Output
- The `printf` Function
- Conversion specifications
- The `scanf` Function



C and Input/Output

- C does not have any native functions for input/output. Instead, I/O functions are provided by extensions in standard library `stdio.h`
- Consequently, to perform any I/O, must have the following line at the top of your code: `#include <stdio.h>`
- C (and UNIX) both buffer input and output for improved efficiency



The `printf` Function

- To print output to `stdout`, use the `printf` function. For usage, see `man printf`.

```
#include <stdio.h>
int printf(const char *format, ...);
```
- `printf` takes a *format string* which contains *place holders of a specified type*, followed by a comma separated list of the actuals that match up with each place holder. Examples:

```
printf("Height: %d width: %d\n", height, width);
printf("Profit: %.2f\n", profit);
```
- Another way to think about `printf`: displays the contents of a string with an unlimited number of values possibly inserted at specified points in the string
 - Values to be inserted can be constants, variables, or more complication expressions
 - Ordinary characters in the format string are printed as they appear



The printf Function (cont)

- **Conversion specifications**
 - always begin with the % character
 - Placeholder for a value to be filled in when string is printed
 - The character(s) that come after the % character specify how the value is to be converted from binary to printed form
- **Note:** Compilers generally do not check the number of conversion specifications in a format string against the number of actuals – common source of errors. Newer compilers can sometimes issue warnings.
 - If there are not enough actuals, garbage will be printed
 - If there are too many actuals, they are discarded



The printf Function (cont)

- In general, a conversion specification has the following form:
 - `%m.pX` or `%-m.pX`
 - Where `m` and `p` are integer constants and `X` is a letter
- `m` and `p` are optional
- `m` specifies the **minimum field width**, or the number of characters to print. The value is right justified by default. A '-' sign makes it left justified.
- `p` specifies the **precision**. It's semantics are a little more complicated as they depend completely on the type being converted to. Specifics are available in the man page, and summarized for the most common types below



The printf Function (cont)

- `X` is the type to convert to. The most common are as follows:
 - `d` – displays an integer in decimal (base 10) form. `p` indicates the minimum number of digits to display, padding with zeros if needed
 - `e` – displays a floating-point number in exponential format (scientific notation). `p` indicates how many digits should appear after the decimal point. If `p` is zero, the decimal point is not displayed.
 - `f` – displays a floating-point number in “fixed decimal” format, without an exponent. `p` has the same meaning as for `e`.
 - `g` – displays a floating-point number in either exponential format or fixed-decimal format, depending on the number's size. `p` indicates the maximum number of significant digits (*not* the digits after the decimal point!)
 - `u` – prints an unsigned number
 - `p` – prints a pointer argument numerically
 - `x` – prints an integer as an unsigned hexadecimal number in lower case
 - `X` – prints an integer as an unsigned hexadecimal number in upper case



The printf Function (cont)

- Modifiers (used between % and conversion specifier `X`):
 - Left justify
 - + Always print a + or – to indicate if number is positive or negative
 - <space> If number doesn't start with a + or – sign, prefix it with space
 - # For `%e`, `%E`, `%f`, forces number to include a decimal point
 - ' Separate digits of an integer using local-specific separator (e.g. ',' in the US: 1,000,000)
- Now, how do we print the '%'?
- `printf` returns the number of characters printed. This return value is **normally** ignored.



Escape Sequences

- Escape sequences enable strings to contain characters that have a special meaning to the compiler – including non-printing control characters

- alert (bell)	\a
- backspace	\b
- new line	\n
- horizontal tab	\t
- "	\"
- \	\\



The scanf Function

- Just like `printf` prints output in a specified format, `scanf` reads input in a specific format
- A `scanf` format string, like `printf`, can contain both ordinary characters and conversion specifications
- The conversion specifications for `scanf` are essentially the same as those for `printf`
- The major difference is that when specifying what variable you want to read the value into, you must prefix the variable's name with a '&' (ampersand) – this is the *addressof* operator that we'll discuss more fully when we get to *pointers*.
- Example:

```
int i,j;
float x,y;
scanf("%d%d%f%f", &i, &j, &x, &y);
```



The scanf Function (cont)

- Failing to use an & on the variable to store the input into will undoubtedly cause your program to crash – or more specifically to produce a segmentation fault (segfault)
- Experienced C programmers avoid using `scanf` because:
 - Programmer must check number of conversion specifications to ensure they match the number of variables and each one is correct
 - Compiler does not check correct number of parameters/actuals
 - '&' symbol can be a major source of error – not always required
 - No graceful error recovery if user enters invalid data
 - One size doesn't fit all...
- `scanf` returns the number of successful conversions. Unlike `printf`, this return value can be vital in ensuring it successfully performed all the requested conversions.



The scanf Function (cont)

- `scanf` is a pattern matching function that skips blank spaces and tries to match input to the format string
- `scanf` returns immediately without looking at the rest of the format string if any item doesn't match
- `scanf` ignores white-space characters (space, horizontal and vertical tabs, new line)
- `scanf` doesn't always require white space between matched items
- When it encounters a character that cannot be part of the currently matching item, it puts the item back onto the input stream and starts scanning the next item. Example:

```
int i,j;
float x,y;
scanf("%d%d%f%f", &i, &j, &x, &y);
Inputs: 1 -20 .3 -4.0e3 versus 1-20.3-4.0e3
```



The scanf Function (cont)

- White spaces: When `scanf` encounters white spaces, it repeatedly reads them until it reaches a non-white space then it puts it back on the stream and then continues scanning the next item
- Ordinary characters that are not white space and are not format specifiers are checked against the input. If they match, `scanf` discards them and proceeds with the scan. If they do not match, it aborts. Example:
 `"%d\\%d"` with input `5555\\7777`
- Confusing `printf` with `scanf`
 - These two functions have similarities, but also have significant differences. Be sure you pay attention to the differences!
 - Observe, `scanf` is not a “wonder-function” that both prints output and reads input at the same time...!



Reading Assignments

- *C Programming: A Modern Approach*. By K.N. King.
 - Chapter 3



Acknowledgments

- *C Programming: A Modern Approach*. By K.N. King.