Basic I/O in C

Allowing Input from the Keyboard, Output to the Monitor

To control input and output (I/O), we use two functions from the standard C library.

Put this line at the top of your C program:

```
#include <stdio.h>
```

This directive tells the C compiler that your program uses the standard C I/O functions.

Write Output Using `printf`

To write text onto the display, use `printf`. The “f” means “formatted.”
- When using the function, you must specify the desired format between quotation marks.

Example:

```
printf ("Here is an example.");
```

The function call above writes the text between the quotes to the monitor.

Use Backslash to Include Special ASCII Characters

Certain ASCII characters
- control text appearance, and
- are hard to put between quotes.

For example
- ASCII’s linefeed character (or if, sometimes called newline)
- starts a new line of text.

To include linefeed, write \n between quotes.

The backslash indicates a special ASCII character. Use \\ for one backslash.
One Can Include Many Linefeeds

Example:
```c
printf("This\n\nhas\n\nlines!\n");
```
The call above prints the three lines below (at the left of the screen).
```c
This
text\nhas
lines!
```
The next `printf` also starts on a new line (because of the linefeed at the end of the format).

Use Format Specifiers to Print Expressions

`printf` also prints expression values
For example,
```c
printf ("Integers: %d %d %d\n", 6 * 7, 17 + 200, 32 & 100);
```
Output: [followed by ASCII linefeed]
```c
Integers: 42 217 32
```
The expressions to print
◦ appear after the format specification, and
◦ are separated by commas.

Many Format Specifiers are Supported

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%c</code></td>
<td>int or char as ASCII character</td>
</tr>
<tr>
<td><code>%d</code></td>
<td>int as decimal</td>
</tr>
<tr>
<td><code>%e</code></td>
<td>double as decimal scientific notation</td>
</tr>
<tr>
<td><code>%f</code></td>
<td>double as decimal</td>
</tr>
<tr>
<td><code>%%</code></td>
<td>one percent sign</td>
</tr>
</tbody>
</table>

See man pages on a lab machine for more.

These Tables Suffice for Our Class

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%u</code></td>
<td>unsigned int as decimal</td>
</tr>
<tr>
<td><code>%x</code></td>
<td>integer as lower-case hexadecimal</td>
</tr>
<tr>
<td><code>%X</code></td>
<td>integer as upper-case hexadecimal</td>
</tr>
</tbody>
</table>

See man pages on a lab machine for more.
Format Specifiers Print Only the Expression Values

If you want spacing, include it in the format.
Example:
```c
printf("%d%d%d", 12, -34, 56);
```
prints
```
12-3456
```
Except for format specifiers and special ASCII characters like linefeed, characters print exactly as they appear.

Pitfall: Passing the Wrong Type of Expression

Be sure that your expressions (and ordering) match the format.
Example:
```c
printf("%d %f", 10.0, 17);
```
may print (output is system dependent)
```
0 0.000000
```
A C compiler may be able to warn you about this kind of error.

Pitfall: Too Few/Many Expressions

If you pass more expressions than format specifiers, the last expressions are ignored.
If you pass fewer expressions than format specifiers, `printf` prints ... bits!
(In other words, behavior is unspecified.)
Again, a C compiler may be able to warn you about this kind of error.

Read Input Using `scanf`

To read values from the keyboard, use `scanf`. The “f” again means “formatted.”

```
scanf also takes
◦ a format in quotation marks, and
◦ a comma-separated list of variable addresses
```
Example:
```c
int A;  // memory address of variable A
scanf("%d", &A);
```
reads a decimal integer, converts it to 2’s complement, and stores the bits in A.
**scanf** Ignores White Space Typed by User

Example: int A;
int B;
scanf("%d%d", &A, &B);
The user can separate the two numbers with spaces, tabs, and/or linefeeds, such as …

5 42 /* A is 5, B is 42 */
5 /* two lines -> same result */
42
The user must push <Enter> when done.

**Other Characters in Format Must be Typed Exactly**

If format includes characters
- other than format specifiers and white space
- user must type them exactly with no extra spaces. Rarely useful.

Example: int A; int B;
scanf("%d<>%d", &A, &B);
Type “5<>42” and A==5, B==42.
But type “5 <>42” and A==5, while B is unchanged (no initializer, so B contains bits).

**Conversion Specifiers Similar to printf**

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>store one ASCII character (as char)</td>
</tr>
<tr>
<td>%d</td>
<td>convert decimal integer to int</td>
</tr>
<tr>
<td>%f</td>
<td>convert decimal real number to float</td>
</tr>
<tr>
<td>%lf</td>
<td>convert decimal real number to double</td>
</tr>
<tr>
<td>%u</td>
<td>convert decimal integer to unsigned int</td>
</tr>
<tr>
<td>%x or %X</td>
<td>convert hexadecimal integer to unsigned int</td>
</tr>
</tbody>
</table>
More Pitfalls for `scanf` than for `printf`

`scanf` has the same pitfalls as `printf`
- Be sure to match format specifiers (and ordering) to variable types.
- Be sure to match number of specifiers to number of addresses given.

And more!
- Don’t forget to write “&” before each variable. (Behavior is again undefined, but can be quite difficult to find the bug.)

---

`printf` Returns the Number of Characters Printed

Function calls are expressions.
Both `printf` and `scanf` return `int` (the calls evaluate to values of type `int`).

`printf` returns the number of characters printed to the display.
Writing a `printf` followed by a semicolon
- evaluates the expression (calls `printf`),
- then discards the return value.

The return value of `printf` is rarely used.

---

`scanf` Returns the Number of Conversions

`scanf` returns the number of conversions performed successfully, or 0 or -1 for no conversions.
The return value is important for checking user input.

For example,
```c
if (2 != scanf ("%d%d", &A, &B)) {
    printf ("Bad input!\n");
    A = 42; B = 10; /* defaults */
}
```