Chapter 3

Formatted Input/Output

The `printf` Function

- The `printf` function must be supplied with a ***format string***, followed by any values that are to be inserted into the string during printing:

  `printf(`*string*`, `*expr*`1, `*expr*`2, ...);`
- The format string may contain both ordinary characters and ***conversion specifications***, which begin with the `%` character.
- A conversion specification is a placeholder representing a value to be filled in during printing.
  - `%d` is used for `int` values
  - `%f` is used for `float` values

The `printf` Function

- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

The `printf` Function

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.
- Too many conversion specifications:

```
printf("%d %d\n", i);    /*** WRONG ***/
```

- Too few conversion specifications:

```
printf("%d\n", i, j);    /*** WRONG ***/
```

The `printf` Function

- Compilers aren't required to check that a conversion specification is appropriate.
- If the programmer uses an incorrect specification, the program will produce meaningless output:

```
printf("%f %d\n", i, x);  /*** WRONG ***/
```

Conversion Specifications

- A conversion specification can have the form `%`*m*`.`*pX* or `%-`*m*`.`*pX*, where *m* and *p* are integer constants and *X* is a letter.
- Both *m* and *p* are optional; if *p* is omitted, the period that separates *m* and *p* is also dropped.
- In the conversion specification `%10.2f`, *m* is 10, *p* is 2, and *X* is `f`.
- In the specification `%10f`, *m* is 10 and *p* (along with the period) is missing, but in the specification `%.2f`, *p* is 2 and *m* is missing.

Conversion Specifications

- The ***minimum field width,*** *m*, specifies the minimum number of characters to print.
- If the value to be printed requires fewer than *m* characters, it is right-justified within the field.
  - `%4d` displays the number 123 as `•123`. (• represents the space character.)
- If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size.
- Putting a minus sign in front of *m* causes left justification.
  - The specification `%-4d` would display 123 as `123•`.

Conversion Specifications

- The meaning of the ***precision,*** *p*, depends on the choice of *X*, the ***conversion specifier.***
- The `d` specifier is used to display an integer in decimal form.
  - *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary).
  - If *p* is omitted, it is assumed to be 1.

Program: Using `printf` to Format Numbers

- The `tprintf.c` program uses `printf` to display integers and floating-point numbers in various formats.

Escape Sequences

- The `\n` code that used in format strings is called an ***escape sequence.***
- Escape sequences enable strings to contain nonprinting (control) characters and characters that have a special meaning (such as ").
- A partial list of escape sequences:

Alert (bell)	`\a`
Backspace	`\b`
New line	`\n`
Horizontal tab	`\t`

Escape Sequences

- A string may contain any number of escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

Escape Sequences

- Another common escape sequence is \", which represents the " character:

```
printf("\"Hello!\"");
  /* prints "Hello!" */
```

- To print a single \ character, put two \ characters in the string:

```
printf("\\");
  /* prints one \ character */
```

The `scanf` Function

- `scanf` reads input according to a particular format.
- A `scanf` format string may contain both ordinary characters and conversion specifications.
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.

The `scanf` Function

- In many cases, a `scanf` format string will contain only conversion specifications:

```
int i, j;
float x, y;

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

- Sample input:

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

`scanf` will assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively.

The `scanf` Function

- When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable.
- Another trap involves the `&` symbol, which normally precedes each variable in a `scanf` call.
- The `&` is usually (but not always) required, and it's the programmer's responsibility to remember to use it.