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*`, `*expr1*`, `*expr2*`, …);`
- 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

- In a format string
  - *Ordinary characters* 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);
printf("i,j,x,y = %d %d %f %f, respectively\n",i,j,x,y);
printf("i,j,x,y = %d%d%f%f, respectively\n", i, j, x, y);
```

- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
i,j,x,y = 10 20 43.289200 5527.000000, respectively
i,j,x,y = 102043.2892005527.000000, respectively
```

The `printf` Function

- Compilers are *not* required to check that
  - the number of conversion specifications in a format string matches the number of output items
    - Too many conversion specifications will produce *meaningless* output
      `printf("%d %d\n", i);    /*** WRONG ***/`
    - Too few conversion specifications might produce *meaningless* output
      `printf("%d\n", i, j);    /*** WRONG ***/`
  - a conversion specification is appropriate
    - An incorrect specification will produce *meaningless* output
      `printf("%f %d\n", i, x);   /*** WRONG ***/`
- Using "`gcc -Wall`" to compile your program will issue you warnings about such situations, and more

Conversion Specifications

m → minimum
p → precision

- 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* can also be dropped
- In the conversion specification `%10.2f`
  – *m* is 10, *p* is 2, and *X* is `f`
- In the conversion specification `%10f`
  – *m* is 10, *p* (along with the period) is missing, and *X* is `f`
- In the conversion specification `%10.f`
  – *m* is 10, *p* is missing, and *X* is `f`
- In the conversion specification `%.2f`
  – *m* is missing, *p* is 2 and, *X* is `f`

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 will be right-justified within the field
  ✓ `%6d` 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 `%-6d` would display 123 as `123•••`
  (• represents the space character)

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

`%.6d` displays the number 123 as `000123`

`%.2d` displays the number 123 as `123`

`%8.6d` displays the number 123 as `••000123`

`%-8.6d` displays the number 123 as `000123••`

Conversion Specifications

- *Conversion specifiers* for floating-point numbers:
  - `f` *Fixed decimal format*:
    - *p* indicates the number of ***digits after the decimal point*** (*the default is 6*)
    - If *p* is 0, no decimal point is displayed
  - `e` *Exponential format*:
    - *p* has the ***same meaning*** as for the `f` specified
  - `g` *Either exponential format* or *fixed decimal format*, depending on the value of the number to be printed--- `e` ***format is selected when:***

    →too small, i.e., exponent is less than -4, or

    →too large numbers, i.e., exponent is greater than or equal the precision
    - *p* indicates the *maximum* ***number of significant digits to be displayed***
    - The `g` conversion will *not* show trailing zeros
    - If the number has no digits after the decimal point, `g` doesn't display the decimal point

Conversion Specifications

- Examples showing how the `%f`, `%e`, and `%g` conversion displays some:

	Result of Applying		
Number	**`%10.4f`**	**`%10.4e`**	**`%10.4g`**
`123456.`	`123456.0000`	`1.2346e+05`	`1.235e+05`
`12345.6`	`12345.6000`	`1.2346e+04`	`1.235e+04`
`1234.56`	`1234.5600`	`1.2346e+03`	`1235`
`123.456`	`123.4560`	`1.2346e+02`	`123.5`
`12.3456`	`12.3456`	`1.2346e+01`	`12.35`
`1.23456`	`1.2346`	`1.2346e+00`	`1.235`
`.123456`	`0.1235`	`1.2346e-01`	`0.1235`
`.0123456`	`0.0123`	`1.2346e-02`	`0.01235`
`.00123456`	`0.0012`	`1.2346e-03`	`0.001235`
`.000123456`	`0.0001`	`1.2346e-04`	`0.0001235`
`.0000123456`	`0.0000`	`1.2346e-05`	`1.235e-05`
`.00000123456`	`0.0000`	`1.2346e-06`	`1.235e-06`

tprintf.c

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

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`
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 -4000
```

`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
  - 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
- Using `"gcc -Wall"` to compile your program will issue you warnings about such situations, and more

How `scanf` Works

- `scanf` tries to match
  - *groups of input characters* with
  - *conversion specifications* in the *format string*
- For each conversion specification, `scanf` tries to
  - skip blank space, if necessary
  - locate the start of an item of the appropriate type in the input data
  - Start reading the item
  - stop when it reaches a character that *can't belong* to the item
  - If the item was read successfully,
    - `scanf` continues processing the rest of the format string
  - If not,
    - `scanf` returns immediately

How `scanf` Works

- As it searches for a number, `scanf` ignores ***white-space characters*** (space, horizontal and vertical tab, form-feed, and new-line)
- A call of `scanf` that reads four numbers can be

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

- The numbers can be on one line or spread over several lines

```
  1
-20   .3
   -4000
```

- `scanf` sees the above stream of characters as

```
••1¤-20•••.3¤•••-4000¤
```

(¤ represents new-line and • represents space)

```
ssrsrrrsssrrssssrrrrr
```
(s = skipped; r = read)

- `scanf` "***peeks***" at the final new-line ***without reading it***

How `scanf` Works

- When asked to read an integer, `scanf`
  - first looks for a plus or minus sign (optional), followed by
  - digits
  - it then reads digits until it reaches a non-digit
- When asked to read a floating-point number, `scanf`
  - looks for a plus or minus sign (optional), followed by
  - digits (possibly containing a decimal point)
  - it then reads digits until it reaches a non-digit

How `scanf` Works

- When `scanf` encounters a character that can't be part of the current item, the character is "put back" to be read again during the scanning of the next input item or during the next call of `scanf`

How `scanf` Works

- Sample input:

```
1-20.3-4000¤
```

- The call of `scanf` is the same as before

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

- Here's how `scanf` would process the new input
  - `%d` stores 1 into `i` and puts the `-` character back
  - `%d` stores –20 into `j` and puts the `.` character back
  - `%f` stores 0.3 into `x` and puts the `-` character back
  - `%f` stores –4000.0 into `y` and puts the `new-line` character back

Ordinary Characters in Format Strings

- Ordinary characters can appear in a format string
- When `scanf` encounters a ***non-white-space*** character in a format string, it compares it with the next input character
  - If they match, `scanf` discards the input character and continues processing the format string
  - If they don't match, `scanf` puts the offending character back into the input, then aborts

Ordinary Characters in Format Strings

- Examples:
  - If the format string is `"%d/%d"` and the input is `•5/•96`
    - ✓ `scanf` succeeds
  - If the input is `•5•/•96`
    - `scanf` fails, because the `/` in the format string doesn't match the space in the input
  - If the format string is `"%d / %d"` and the input is
    - ✓ `•5/•96` ➔ `scanf` succeeds
    - ✓ `5/96` ➔ `scanf` succeeds
    - ✓ `•5•••••/•••••••96` ➔ `scanf` succeeds
    - ✓ `5`
      `/`
      `96` ➔ `scanf` succeeds

Confusing `printf` with `scanf`

- Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two
- One common mistake is to put `&` in front of variables in a call of `printf`

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

Confusing `printf` with `scanf`

- Incorrectly assuming that `scanf` format strings should be like `printf` format strings is another common error
- Consider the following call of `scanf`

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

  - `scanf` will first look for an integer in the input, which it stores in the variable `i`
  - `scanf` will then try to match a comma with the next input character
  - If the next input character is a space, not a comma, `scanf` will terminate without reading a value for `j`

Program: Adding Fractions

- The **`addfrac.c`** program prompts the user to enter two fractions and then displays their sum
- Sample program output

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

addfrac.c

```
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 *denom1;
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n",result_num, result_denom)

  return 0;
}
```