

## Chapter 3

## Formatted Input/Output

1

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.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

2

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.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
```

3

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.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 ***/
```

4

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.

## 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:

```
int    i = 123;
float  x = 321.0f;
printf("%f %d\n", i, x);  /* ** WRONG ** */
```

- Output:

```
0.000000 0
```

5

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.

## **%m.pX** 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.

6

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.

## **%m.pX** 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•**.

7

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.

## **%m.pX** 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.

8

Copyright © 2008 W. W. Norton & Company.  
All rights reserved.

## `%m.pX` Conversion Specifications

- Conversion specifiers for floating-point numbers:
- **e** — **Exponential format**.
  - *p* indicates **how many digits** should appear **after the decimal point** (the default is 6). If *p* is 0, no decimal point is displayed.
- **f** — **“Fixed decimal” format**.
  - *p* has the same meaning as for the *e* specifier.
- **g** — Either **exponential format** or **fixed decimal format**, depending on the number’s size.
  - *p* indicates the **maximum number of significant digits** to be displayed.
  - The *g* conversion won’t show **trailing (後面的) zeros**.
  - If the number has no digits after the decimal point, *g* doesn’t display the decimal point.

## Program: Using `printf` to Format Numbers

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

## `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 (轉義序列)**.
- 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)	<code>\a</code>
Backspace	<code>\b</code>
New line	<code>\n</code>
Horizontal tab	<code>\t</code>

## 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.

## 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 locate an item of the appropriate type in the input data,
  - **skipping blank space** if necessary.
- `scanf` then **reads the item**, stopping 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:  
`scanf("%d%d%f%f", &i, &j, &x, &y);`
- The numbers can be on **one line** or spread over **several lines**:
 

```

1
-20  .3
    -4.0e3
```
- `scanf` sees a **stream of characters** (□ represents **new-line**):  
`●●1□-20●●●.3□●●-4.0e3□`  
`ssrsrrrrsssrsssrssrrrrrr` (s = skipped; r = read)
- `scanf` “**peeks**” (取數指令/一瞥) at the final **new-line** without **reading it**.
- peek 在讀出資料後，不會將該資料自緩衝區刪除


## How scanf Works

- When asked to read an **integer**, `scanf`
  - first searches for a **digit**, a **plus sign**, or a **minus sign**;
  - it then reads digits until it reaches a **nondigit**.
- 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), followed by
  - an **exponent** (optional). An exponent consists of the letter **e** (or **E**), an **optional sign**, and **one or more digits**.
- `%e`, `%f`, and `%g`
  - are interchangeable when used with `scanf`.

## 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:  
1-20.3-4.0e3↵  new-line
- 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 **-4.0 × 10<sup>3</sup>** into **y** and puts the new-line character back.

## Ordinary Characters in Format Strings

- When it encounters
  - one or more **white-space characters** in a format string,
  - `scanf` reads **white-space characters** from the input until it reaches a **non-white-space character** (which is “put back”).
- When it encounters
  - a **non-white-space character** in a format string,
  - `scanf` 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.
- To allow spaces after the first number, use the format string **"%d /%d"** instead.

## 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 resemble `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`.

## Confusing `printf` with `scanf`

- Putting a **new-line character** at the end of a `scanf` format string is usually a bad idea.
- To `scanf`, a **new-line character** in a format string
  - is equivalent to a **space**; both cause `scanf` to advance to the next **non-white-space character**.
- If the format string is `"%d\n"`,
  - `scanf` will skip **white space**,
  - read an integer,
  - then skip to the next **non-white-space character**.
- A **format string** like this can cause an interactive program to “hang.”

## 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;  
}
```