More Input and Output

Learning Outcomes

After reading this section, you will be able to:

- Implement algorithms using standard library procedures to incorporate existing technology
- Stream data using standard library functions to interact with users and access persistent text

Introduction

The standard input and output library (stdio) that ships with C compilers provides comprehensive support for communicating with the user and with secondary storage. This support includes numerical as well as character string processing under format control and optionally line by line processing without format control. For platforms that don't support line-by-line input processing, we write our own custom procedures.

This chapter reviews the conversion specifiers for formatted input and output along with the library functions for line by line input and output. Specifiers not covered in previous chapters are included here. This chapter concludes with two custom functions for input that safeguard line mismatching and memory overflow.

Input

The stdio library functions for processing input are:

- `scanf()` - input from standard input under format control
- `fscanf()` - input from file under format control
- `getchar()` - character by character input from standard input (see Input Functions)
- `fgetc()` - character by character input from file (see Input Functions)
- `gets_s()` - line by line input from standard input (not universally implemented)
- `fgets()` - line by line input from file

NOTE

Typically, ***standard input*** refers to the **keyboard**.

Formatted Input

The `scanf(...)` and `fscanf(...)` functions accept data from the standard input device or secondary storage respectively and store that data in memory at the address specified in their argument list. Their prototypes are:

```
int scanf(const char *format, address);
int fscanf(FILE *, const char *format, address);
```

format

`format` receives a string literal that describes how to convert input text into data stored in memory. Calls to these functions can take multiple arguments. `format` contains the conversion specifier(s) for translating the input characters. Conversion specifiers begin with a `%` symbol and identify the type of the destination variable. The possible specifiers are listed below. Other specifiers may be found on the web.

Specifier	Input Text is a	Destination Type
`%c`	character	`char`, `char []`
`%d`	decimal	`int`, `short`, `long`, `long long`
`%i`	integer	`int`, `short`, `long`, `long long`
`%o`	unsigned octal	`unsigned int`, `short`, `long`, `long long`
`%x`	unsigned hexadecimal	`unsigned int`, `short`, `long`, `long long`
`%u`	unsigned decimal	`unsigned int`, `short`, `long`, `long long`

Specifier	Input Text is a	Destination Type
`%n`	--	`int`, `short`, `long`, `long long`
`%f` `%e` `%g` `%a`	floating-point	`float`, `double`, `long double`
`%s`	character string	`char []`
`%[ ]` `%[^ ]`	character string	`char []`
`%p`	address	any type

REMARKS

- `%n` does not read any characters but instead returns the number of characters processed.
- `%f`, `%e`, `%g` and `%a` treat floating-point input identically.
- *Size specifiers* also apply to `%i` , `%o`, `%x`, `%u` and `%n`, but are not listed here.

address

`address` receives the address of the destination variable. We specify a separate address argument for each conversion specifier in the format string.

Conversion Control

We may insert control characters between the `%` and the conversion character. The general form of a conversion specification is:

```
% * width size conversion_character
```

The three control characters are:

1. ***** - suppresses storage of the converted data (**discards** it without storing it)
2. ***width*** - specifies the maximum number of characters to be interpreted
3. ***size*** - specifies the size of the storage type

EXCEPTION

A conversion specifier that includes a `*` does not have a corresponding address in the argument list. This is an exception to the matching conversion-specifier/argument rule.

The size specifiers covered in this course are listed below. Others may be found on the web.

Specifier with Size	Input Text is a	Destination Type
`%hhd` `%hhi`	very short decimal	`char`
`%hd` `%hi`	short decimal	`short`
`%ld` `%li`	long decimal	`long`
`%lld` `%lli`	very long decimal	`long long`
`%lf` `%le` `%lg` `%la`	floating-point	`double`
`%Lf` `%Le` `%Lg` `%La`	floating-point	`long double`
`%hhu` `%hho` `%hhx`	unsigned very short decimal	`unsigned char`
`%hu` `%ho` `%hx`	unsigned short decimal	`unsigned short`
`%lu` `%lo` `%lx`	unsigned long decimal	`unsigned long`
`%llu` `%llo` `%llx`	unsigned very long decimal	`unsigned long long`
`%hhn`	character string	`char`
`%hn`	character string	`short`
`%ln`	character string	`long`
`%lln`	character string	`long long`

Problems with `%c`

`scanf()` and `fscanf()` only extract the characters they need from the buffer, but problems arise with `%c` conversions. Consider the following program. On reading an integer value, `scanf()` leaves the newline character (`'\n'`)in the input buffer. Since the next call to `scanf()` starts with a `%c` specifier, `scanf()` treats the unprocessed `'\n'` as the input character. As a result, this program never collects the tax status input from the input buffer.

```
/* scanf with %c Specification
* scanf_c.c
*/

#include <stdio.h>

int main(void)
{
    int items;
    char status; // tax status g or p

    printf("Number of items : ");
    scanf("%d", &items);

    printf("Status : ");
    scanf("%c", &status);   // ERROR: assigns \n to variable 'status'
                            //        and will not pause for user input

    printf("%d items (%c)\n", items, status);

    return 0;
}
```

The above program produces the following output:

```
Number of items : 25
Status : 25 items (
)
```

NOTE

Notice how the newline character (`'\n'`) (which was assigned to the tax **status** variable) places the closing parenthesis on a newline.

There are different ways to handle unprocessed `'\n'` characters. Some are listed in the ***code snippets*** below.

HELPFUL HINT

A **space character** *before* a conversion specifier forces the skipping of ***all leading whitespace*** before the next conversion.
For example, `" %c"` directs `scanf()` to skip any whitespace characters before attempting to read the next non-whitespace character.

Method-1:

```
scanf("%d", &items);
scanf("%c%c", &junk, &status); // store one character in junk first
```

Method-2:

```
scanf("%d", &items);
scanf("%*c%c", &status);       // discard(ignore) one character first
```

Method-3:

```
scanf("%d", &items);
scanf(" %c", &status);         // discard(ignore) all whitespace first
```

Method-4:

```
scanf("%d%*c", &items);        // discard(ignore) newline ('\n')
scanf("%c", &status);
```

Method-5:

```
scanf("%d", &items);
clear();                       // call a custom function to clear the buffer
scanf("%c", &status);
```

`"%*c%c"` discards/ignores **one** character and accepts the next.
`"%c"` discards/ignores **all** whitespace characters before the next non-whitespace character.

One corrected version of the above program is:

```
// scanf with %c Specification
// scanf_cc.c

#include <stdio.h>

int main(void)
{
    int items;
    char status; // tax status g or p

    printf("Number of items : ");
    scanf("%d", &items);

    printf("Status : ");
    scanf(" %c", &status);   // note the space

    printf("%d items (%c)\n", items, status);

    return 0;
}
```

Unformatted Input

The library functions for processing unformatted input are:

- `getchar()` - character by character input from standard input (see Input Functions)
- `fgetc()` - character by character input from file (see Input Functions)
- `gets_s()` - line by line input from standard input (not universally implemented)
- `fgets()` - line by line input from file

gets_s (Optional)

The `gets_s()` function...

- accepts an empty string
- assumes no more than the specified number of characters
- reads the `'\n'` as the delimiter
- replaces the delimiter with the null terminator

`gets_s()` takes two arguments. Its prototype is:

```
char *gets_s(char *address, int n);
```

The first parameter receives the address of the string to be filled. The second parameter receives the maximum number of characters that can be stored including the null terminator byte. On success, this function returns the address of the filled string:

```
// Read and Display Lines
// gets_s.c

#include <stdio.h>

int main(void)
{
    char first_name[21];
    char last_name[21];

    printf("First Name : ");
    gets_s(first_name, 21);

    printf("Last Name  : ");
    gets_s(last_name, 21);

    puts(first_name);
    puts(last_name);

    return 0;
}
```

The above program produces the following output:

```
First Name : Arnold
Last Name  : Schwartzenegger
Arnold
Schwartzenegger
```

INFO

The behaviour of `gets_s()` is **undefined** if the user inputs a line **longer** than the allocated string. On a Windows platform, this function **crashes**.
The standard recommends use of `fgets()` instead of `gets_s()`.

fgets

The `fgets()` function...

- reads a stream of bytes from the specified file
- accepts an empty string
- accepts no more than the specified number of characters
- reads until the `'\n'` delimiter
- includes the `'\n'` delimiter in the character string
- does not discard the `'\n'` delimiter
- adds the null terminator byte to the character string

The prototype for this function is:

```
char* fgets(char str[], int max, FILE *fp);
```

`str` receives the address of the string to be filled.
`max` receives the maximum number of bytes in `str` including space for the null terminator byte.
`fp` receives the address of the `FILE` object.
`fgets()` appends the null terminator byte to the stored string. `fgets()` *returns* the **address** of `str` if successful, otherwise, `NULL` in the event of an end of file or read error.

Output

The stdio library functions for processing output are:

- `printf()` - output to standard output under format control
- `fprintf()` - output to a file under format control
- `putchar()` - character by character output to standard output (see Output Functions)
- `fputc()` - character by character output to a file (see Output Functions)
- `puts()` - character string output to standard output (see Output Functions)
- `fputs()` - character string output to a file (see Output Functions)

Formatted Output

The `printf(...)` and `fprintf(...)` functions report the value of the variable(s) or expression(s) in the argument list to the standard output device or the specified file respectively. Their prototypes take the form:

```
int printf(const char *format, ...);
int fprintf(FILE *, const char *format, ...);
```

format

`format` is a string literal containing conversion specifiers and any characters to be output directly. Each conversion specifier begins with a `%` symbol and identifies the type of the source variable. The order of the specifiers matches the order of the values received.

Conversion Specifiers

The conversion specifiers include:

Specifier	Output Text is a	Use with Type
%c	character	`char`
%d	signed decimal	`int`, `short`, `long`, `long long`

Specifier	Output Text is a	Use with Type
`%i`	signed integer	`int`, `short`, `long`, `long long`
`%u`	unsigned decimal	`unsigned int`, `short`, `long`, `long long`
`%o`	unsigned octal	`unsigned int`, `short`, `long`, `long long`
`%x`	unsigned hexadecimal	`unsigned int`, `short`, `long`, `long long`
`%X`	unsigned hexadecimal (uppercase)	`unsigned int`, `short`, `long`, `long long`
`%n`	--	`int *`
`%f`	floating-point	`float`, `double`, `long double`
`%F`	floating-point (uppercase)	`float`, `double`, `long double`
`%e`	scientific floating-point	`float`, `double`, `long double`
`%E`	scientific floating-point (uppercase)	`float`, `double`, `long double`
`%g`	shortest floating-point	`float`, `double`, `long double`
`%G`	shortest floating-point (uppercase)	`float`, `double`, `long double`
`%a`	hexadecimal floating-point	`float`, `double`, `long double`
`%A`	hexadecimal floating-point (uppercase)	`float`, `double`, `long double`
`%s`	string of characters	`char *`

Specifier	Output Text is a	Use with Type
`%%`	the character %	`char *`
`%p`	address	--

REMARKS

- `%n` does not output any characters but instead returns the number of characters processed so far.
- ***Scientific*** (`%e` `%E`) refers to output in mantissa/exponent form `d.dddEdd` (for example, `0.123e3`, which stands for `0.123 x 103` or `123.0`).
- ***General*** (`%g` `%G`) refers to output in the shortest form possible; decimal or mantissa/exponent (for example, `0.123e-5` rather than `0.00000123` and `3.1` rather than `0.31e1`).

Conversion Control

We may insert control characters between the `%` and the conversion character. The general form of a conversion specification is:

```
% flags width . precision size conversion_character
```

The five control characters are:

1. ***flags***
   - prescribes left justification of the converted value in its field
   - `0` pads the field width with **leading zeros**
2. ***width*** sets the minimum field width within which to format the value (overriding with a wider field only if necessary). Pads the converted value on the left (or right, for left alignment). The padding character used is either a **space** or `0` if the padding flag is on
3. `.` separates the field's width from the field's precision
4. ***precision*** sets the number of digits to be printed after the decimal point for `%f` conversions and the minimum number of digits to be printed for an **integer** (adding leading zeros if

necessary). A value of `0` suppresses the printing of the decimal point in a `%f` conversion. An `*` instead of a number applies the value from the next argument in the argument list

5. ***size*** identifies the *minimum* size of the type being output

The size specifiers covered in this course are listed below. Others may be found on the web

Specifier with Size	Output Text is	Use with Type
`%hhd` `%hhi`	very short decimal	`char`
`%hd` `%hi`	short decimal	`short`
`%ld` `%li`	long decimal	`long`
`%lld` `%lli`	very long decimal	`long long`
`%lf` `%lF` `%le` `%lE` `%lg` `%lG` `%la` `%lA`	floating-point	`double`
`%Lf` `%LF` `%Le` `%LE` `%Lg` `%LG` `%La` `%LA`	floating-point	`long double`
`%hhu` `%hho` `%hhx` `%hhX`	unsigned very short decimal	`unsigned char`
`%hu` `%ho` `%hx` `%hhX`	unsigned short decimal	`unsigned short`
`%lu` `%lo` `%lx` `%hhX`	unsigned long decimal	`unsigned long`
`%llu` `%llo` `%llx` `%hhX`	unsigned very long decimal	`unsigned long long`
`%hhn`	character string	`char`
`%hn`	character string	`short`
`%ln`	character string	`long`
`%lln`	character string	`long long`

Custom Input (Optional)

Mismatching Line Input

Managing line-oriented input helps in debugging. Consider a set of input lines some of which contain incorrect input. Ideally, a one-to-one correspondence should exist between the lines of input data and the lines read by the program. Even if the user inputs a line incorrectly, subsequent correct input may still be acceptable. In other words, incorrect input on one line should not cause incorrect reading of subsequent lines.

Ideally, line by line input should:

- store characters only to a specified maximum
- accept an empty string
- read the `'\n'` as the line delimiter
- discard the delimiting character along with any characters that overflow memory
- append the null terminator to the set of characters stored

The following code meets all of these conditions:

```
// Custom Line-Oriented Input
// getline.c

#include <stdio.h>

// getline accepts a newline terminated
// string s of up to max - 1 characters,
// adds the null terminator and discards
// the remaining characters in the input
// buffer including terminating character
char *getline(char *s, int n)
{
    int i, c;

    for (i = 0; i < n - 1 && (c =getchar()) != EOF && c != (int)'\n'; i++)
    {
        s[i] = c;
    }

    s[i] = '\0';
    while (n > 1 && c != EOF && c != (int)'\n')
    {
        c = getchar();
    }

    return c != EOF ? s : NULL;
}

int main(void)
{
    char first_name[11];
    char last_name[11];

    printf("First Name : ");
    getline(first_name, 11);

    printf("Last Name  : ");
    getline(last_name, 11);

    puts(first_name);
    puts(last_name);

    return 0;
}
```

The above program produces the following output:

```
First Name : Arnold
Last Name  : Schwartzenegger
Arnold
Schwartzen
```

This function, unlike `gets_s()` has well-defined behavior if the number of characters entered exceeds the amount of memory available to store the string.

Insufficient Memory (Optional)

Consider the file named `spring.dat`, the contents of which are listed below. Each record in this file contains three fields: the first field holds the `quantity`, the second field holds a C string describing the item (`label`) and the third field holds the unit `price` of the item. The field delimiter is the semicolon (`;`) character:

```
2;Light Jacket;95.89
3;Long Pants;67.89
2;Large Duster;45.98
```

The following program reads each record from the file and displays the fields in a tabular format:

```
// Tabular Data
// table.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    char label [14];
    int n;
    double price;

    fp = fopen("spring.txt","r");
    if (fp != NULL)
    {
        printf("    Spring Items\n"
            "    ============\n\n"
            "No Description  Price\n"
            "---------------------\n");

        while (fscanf(fp, "%d;%13[^;];%lf%*c", &n, label, &price) == 3)
        {
            printf("%2d %-13s%5.2lf\n", n, label, price);
        }

        fclose(fp);
    }

    return 0;
}
```

The above program produces the following output:

```
    Spring Items
    ============

No Description  Price
---------------------
 2 Light Jacket 95.89
 3 Long Pants   67.89
 2 Large Duster 45.98
```

NOTE

Notice how the field delimiters have been embedded within `fscanf()`'s format string.

Safe Coding

The above program executes successfully only if the descriptive strings in the file do not contain **more than 13 characters**. The data in a different file that contains longer labels will not fit into the space allocated by the program.

To process any file and safeguard against ***memory overflow***, we can modify the program to **skip the extra characters** in the description field that exceeds the memory allocated for the `label` C string character array variable. We do so by reading each record in two separate statements:

```c
// Insufficient Memory
// table_plus.c

#include <stdio.h>

int main(void)
{
    FILE *fp = NULL;
    char label [14];
    int n;
    double price;
    char c;

    fp = fopen("spring.txt","r");

    if (fp != NULL)
    {
        printf("    Spring Items\n"
            "    ============\n\n"
            "No Description  Price\n"
            "---------------------\n");

        while (fscanf(fp,"%d;%13[^;]%c", &n, label, &c) == 3)
        {
            if (c == ';')
            {
                fscanf(fp,"%lf\n", &price);
            }
            else
            {
                fscanf(fp, "%*[^;];%lf%*c", &price);
            }

            printf("%2d %-13s%5.2lf\n", n, label, price);
        }

        fclose(fp);
    }

    return 0;
}
```

The above program produces the following output:

```
       Spring Items
       ============

No Description  Price
---------------------
 2 Light Jacket 95.89
 3 Long Pants   67.89
 2 Large Duster 45.98
```

The first statement reads the first two fields stopping at the second delimiter or once memory is full, whichever comes first. If the statement has encountered the second delimiter, the second statement reads the `price`; if not, the alternate version of the second statement skips the remaining characters in the field and the second delimiter and only then reads the `price`.

The program stops reading altogether as soon as it encounters a record with other than 3 input values - the quantity, the descriptive string and the second delimiter.