Input Functions

Invoke standard library procedures to stream data from users

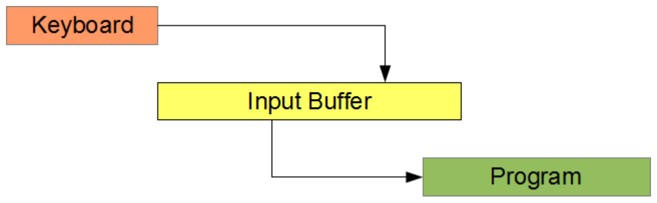
"In nearly all secure programs, your first line of defense is to check every piece of data you receive." (IBM DeveloperWorks, 2016)

Some programming languages leave input and output support to the libraries developed for the languages. For instance, the core C language does not include input and output specifications. These facilities are available in a set of functions, which are defined in the **`stdio`** module. This module ships with the C compiler. Its name stands for standard input and output. Typically, standard input refers to the system keyboard and standard output refers to the system display. The system header file that contains the prototypes for the functions in this module is **`<stdio.h>`**.

This chapter describes some of the input facilities supported by the **`stdio`** module, introduces buffered input, describes two library functions that accept formatted and unformatted buffered input and demonstrates how to validate user input.

BUFFERED INPUT

A *buffer* is a small region of memory that holds data temporarily and provides intermediate storage between a device and a program. The system stores each keystroke in the input buffer, without passing it to the program. The user can edit their data before submitting it to the program. only by pressing the **`\n`** key, the user signals the program to start extracting data from the buffer. The program then only retrieves the data that it needs and leaves the rest in the buffer for future retrievals. The figure below illustrates the buffered input process.

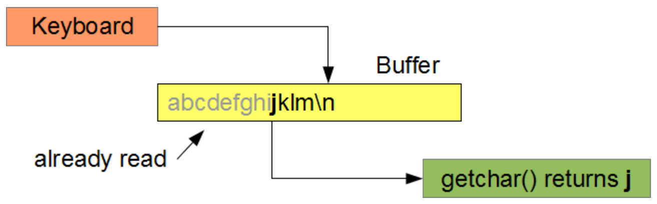


Two functions accept buffered input from the keyboard (the standard input device):

- **`getchar()`** - unformatted input
- **`scanf()`** - formatted input

UNFORMATTED INPUT

The function **`getchar()`** retrieves the next unread character from the input buffer.



The prototype for **`getchar()`** is

```
int getchar(void);
```

`getchar()` returns either

- the character code for the retrieved character
- `EOF`

The character code is the code from the collating sequence of the host computer. You can find the ASCII collating sequence here. If the next character in the buffer waiting to be read is `'j'` and the collating sequence is ASCII, then the value returned by `getchar()` is 106.

`EOF` is the symbolic name for end of data. It is assigned the value `-1` in the `<stdio.h>` system header file. On Windows systems, the user enters the end of data character by pressing `Ctrl-Z`; on UNIX systems, by pressing `Ctrl-D`.

Clearing the buffer

To synchronize user input with program execution the buffer should be empty. The following function clears the input buffer of all unread characters.

```
// clear empties the input buffer
//
void clear(void)
{
        while (getchar() != '\n')
                ;  // empty statement intentional
}
```

The iteration continues until `getchar()` returns the newline (`'\n'`) character, at which point the buffer is empty and `clear()` returns control to its caller.

Pausing Execution

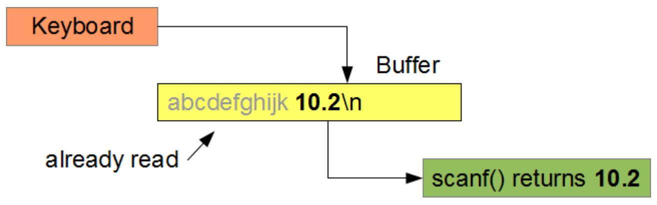
To pause execution at a selected point in a program, consider the following function

```
// pause execution
//
void pause_(void)
{
        printf("Press enter to continue ...");
        while (getchar() != '\n')
                ;  // empty statement intentional
}
```

This function will not return control to the caller until the user has pressed `'\n'`.

FORMATTED INPUT

The `scanf()` function retrieves the next set of unread characters from the input buffer and translates them according to the conversion(s) specified in the format string. `scanf()` extracts only as many characters as required to satisfy the specified conversion(s).



The prototype for `scanf()` is

```
int scanf(format, ... );
```

format consists of one or more conversion specifiers enclosed in a pair of double quotes. The ellipsis refers to one or more addresses.

`scanf()` extracts characters from the input buffer until `scanf()` has either

- interpreted and processed data to match all conversion specifiers in the format string
- found a character that fails to match the next conversion specified in the format string
- emptied the buffer completely

In a mismatch between the conversion specifier and the next character in the buffer, `scanf()` leaves the offending character in the buffer and returns to the caller. In the case of an emptied buffer, `scanf()` waits until the user adds more data to the buffer.

Each conversion specifier describes how `scanf()` is to interpret the next set of characters in the buffer. Once `scanf()` has completed a conversion, it stores the result in the address passed to the corresponding parameter.

We provide as many conversion specifiers in the format string as there are address arguments in the call to `scanf()`.

Conversion Specifiers

Each conversion specifier begins with a `%` symbol and ends with a conversion character. The conversion character describes the type to which `scanf()` is to convert the next set of text characters.

Specifier	Input Text	Convert to Type ...	Most Common
`%c`	character	`char`	*
`%d`	decimal	`char, int, short, long, long long`	*
`%o`	octal	`int, char, short, long, long long`	
`%x`	hexadecimal	`int, char, short, long, long long`	
`%f`	floating-point	`float, double, long double`	*

The following program converts two input fields into data values of `int` type and `float` type respectively:

```
/* scanf conversion specifiers
 * scanf.c
 */

#include <stdio.h>

int main(void)
{
        int items;
        float price;
        printf("Enter items, price : ");
        scanf("%d%f", &items, &price);
        return 0;
}
```

```
Enter items, price : 4 39.99
```

Whitespace

`scanf()` treats the whitespace between text characters of the user's input as a separator between input values. There is no need to place a blank character between the conversion specifiers.

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

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

For integer values:

Size Specifier	Convert to Type
none	`int`
`hh`	`char`
`h`	`short`
`l`	`long`
`ll`	`long long`

For floating-point values:

Size Specifier	Convert to Type
none	`float`
`l`	`double`
`L`	`long double`

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

Problems with %c (Optional)

Because `scanf()` only extracts the characters that it needs from the input buffer, problems arise with `%c` conversions. If you encounter such difficulty see the section with this tilte in the chapter entitled More Input and Output.

Plain Characters (Optional)

Plain characters in the format string - those not preceded by the conversion symbol - serve a special purpose. Each such character requires exact duplication on input. If the user enters a plain character other than that specified in the format string, `scanf()` abandons further interpretation.

To input `%` as a plain character (and distinguish it from the symbol identifying a conversion specifier), we insert `%%` into the format string.

Return Value

`scanf()` returns either the number of addresses successfully filled or `EOF`. A return value of

- 0 indicates that `scanf()` did not fill any address
- 1 indicates that `scanf()` filled the first address successfully
- 2 indicates that `scanf()` filled the first and second addresses successfully
- ...
- `EOF` indicates that `scanf()` did not fill any address AND encountered an end of data character

The return code from `scanf()` does not reflect success of `%*` conversions or any successful reading of plain characters in the format string.

VALIDATION (OPTIONAL)

Since we can never predict that all users will never make mistakes in inputting data to our programs, input validation is an important part of our programming tasks.

To validate the input data that a program receives, we can perform many checks. We localize our validation in special functions that trap erroneous input and request corrections to that input. Erroneous input may include:

- invalid characters
- trailing characters
- out-of-range input
- incorrect number of input fields

The following program includes a special function (`getInt()`), which provides robust validation for integer input. This function tests for no input, trailing characters and out-of-range input.

```
/* Robust Input Validation
 *  getInt.c
 */

#include <stdio.h>
int getInt(int min, int max);
void clear(void);
#define MIN 3
#define MAX 15

int main(void)
{
        int input;

        input = getInt(MIN, MAX);
        printf("\nProgram accepted %d\n", input);

        return 0;
}

// getInt accepts an int between min and max
// inclusive, returns the value of the int accepted
//
int getInt(int min, int max)
{
        int value, keeptrying = 1, rc;
        char after;

        do {
                printf("Enter an integer in\n"
                 "range [%d,%d] : ", min, max);
                rc = scanf("%d%c", &value, &after);
                if (rc == 0) {
                        printf("**Bad char(s)!**\n");
                        clear();
                } else if (after != '\n') {
                        printf("**Trail char(s)!**\n");
                        clear();
                } else if (value < min ||
                 value > max) {
                        printf("**Out of range!**\n");
                } else
                        keeptrying = 0;
        } while (keeptrying == 1);

        return value;
}

// clear empties the input buffer
//
void clear(void)
{
        while (getchar() != '\n')
                ;  // empty statement intentional
}
```

```
Enter an integer in
range [3,15] : we34
**Bad char(s)!**

Enter an integer in
range [3,15] : 34.4
**Trail char(s)!**

Enter an integer in
range [3,15] : 345
**Out of range!**

Enter an integer in
range [3,15] : 14

Program accepted 14
```

EXERCISES

- List the different ways of handling an unprocessed `'\n'` in the input stream before reading a character from that stream
- Complete the exercise on Input Validation.