Advanced Programming (C & C++)

Mahboob Ali

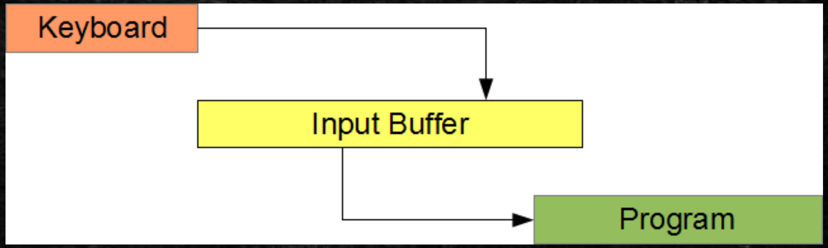
Input & Output Functions

Input & Output

- 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>`.

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.
- Two functions accept buffered input from the keyboard (the standard input device):
  - `getchar()` - unformatted input
  - `scanf()` - formatted input

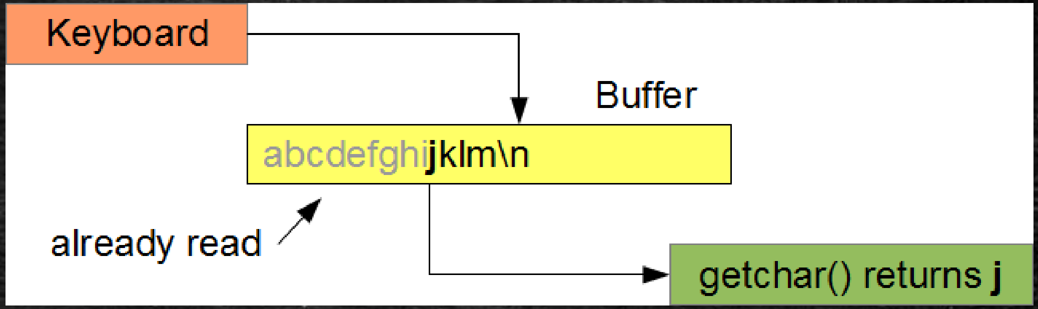


Unformatted Input

- 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 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.

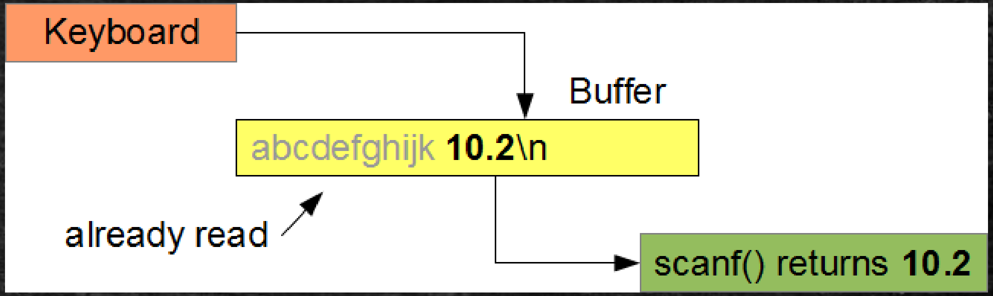
Pausing Execution

- 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



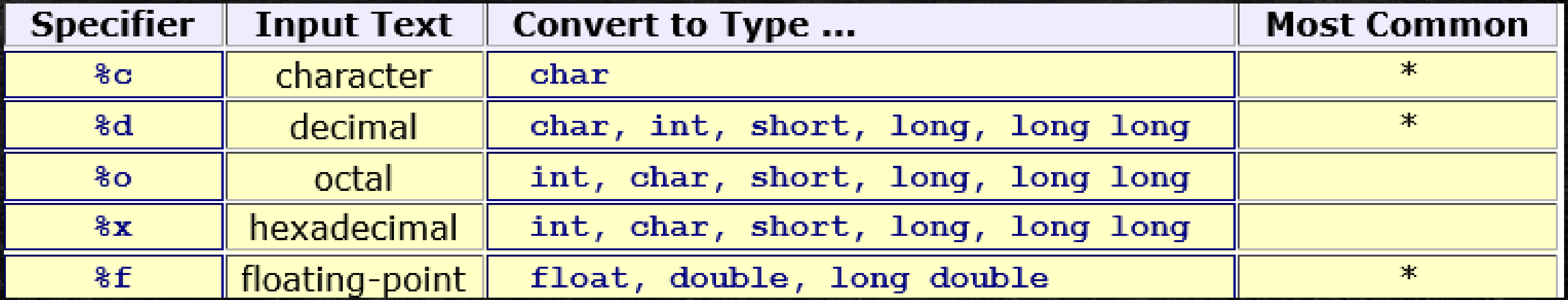
- The `scanf()` function retrieves the next set of unread characters from the input buffer and translates them according to the conversion(s) specified in the format string.
- `scanf()` extracts only as many characters as required to satisfy the specified conversion(s).
- The prototype for `scanf()`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

Whitespace

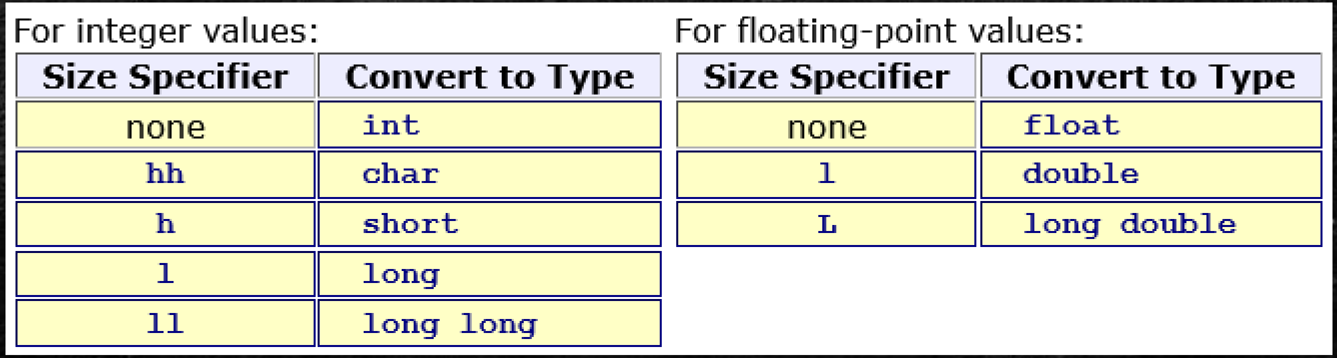
- `scanf()` treats the whitespace between text characters of the user's input as a separator between input values.
- There is no need to place a blank character between the conversion specifiers.

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`

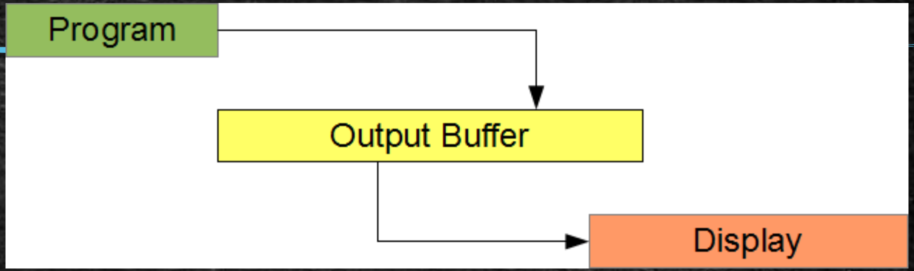
Return Value

- `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.

Output Functions

- The adequate provision of a user interface is an important aspect of software development: an interface that consists of user friendly input and user friendly output.
- The output facilities of a programming language convert the data in memory into a stream of characters that is read by the user.
- The `stdio` module of the C language provides such facilities.

Buffering



- Standard output is line buffered.
- A program outputs its data to a buffer.
- That buffer empties to the standard output device separately.
- When it empties, we say that the buffer flushes.
- Output buffering lets a program continue executing without having to wait for the output device to finish displaying the characters it has received.
- The output buffer flushes if
  - it is full
  - it receives a newline (`\n`) character
  - the program terminates
- Two functions in the `stdio` module that send characters to the output buffer are
  - `putchar()` - unformatted
  - `printf()` - formatted

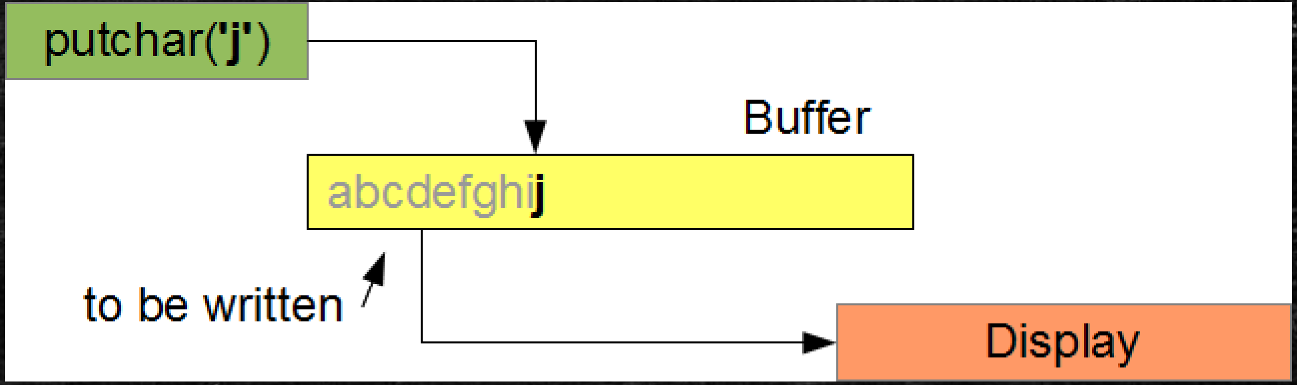
Unformatted Output

- The `putchar()` function sends a single character to the output buffer.
- We pass the character as an argument to this function.
- The function returns the character sent or EOF if an error occurred.
- The prototype for `putchar()` is

  `int putchar (int);`

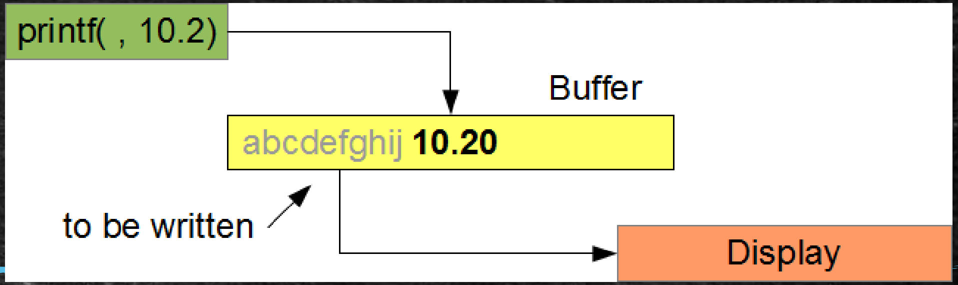
- To send the character 'a' to the display device, we write

```
// Single character output
// putchar.c
#include <stdio.h>
int main(void)
{
        putchar('a');
        return 0;
}
```



- Note that `putchar()` can take EOF as an argument.

Formatted Output



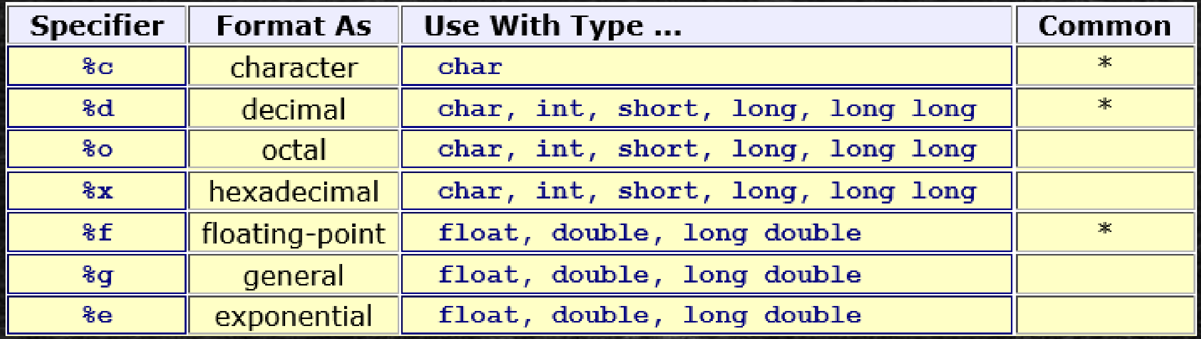
- The `printf()` function sends data to the output buffer under format control and returns the number of characters sent.
- The prototype for the printf() function is

```
    int printf(format, argument, ... );
```

- *`format`* is a set of characters enclosed in double-quotes that may consist of any combination of plain characters and conversion specifiers.
- The function sends the plain characters as is to the buffer and uses the conversion specifiers to translate each value passed as an argument in the function call.
- The ellipsis indicates that the number of arguments can vary. Each conversion specifier corresponds to one argument.

Conversion Specifiers

A conversion specifier begins with a % symbol and ends with a *conversion character*. The conversion character defines the formatting as listed in the table below



Specifier	Format As	Use With Type ...	Common
`%c`	character	`char`	*
`%d`	decimal	`char, int, short, long, long long`	*
`%o`	octal	`char, int, short, long, long long`	
`%x`	hexadecimal	`char, int, short, long, long long`	
`%f`	floating-point	`float, double, long double`	*
`%g`	general	`float, double, long double`	
`%e`	exponential	`float, double, long double`	

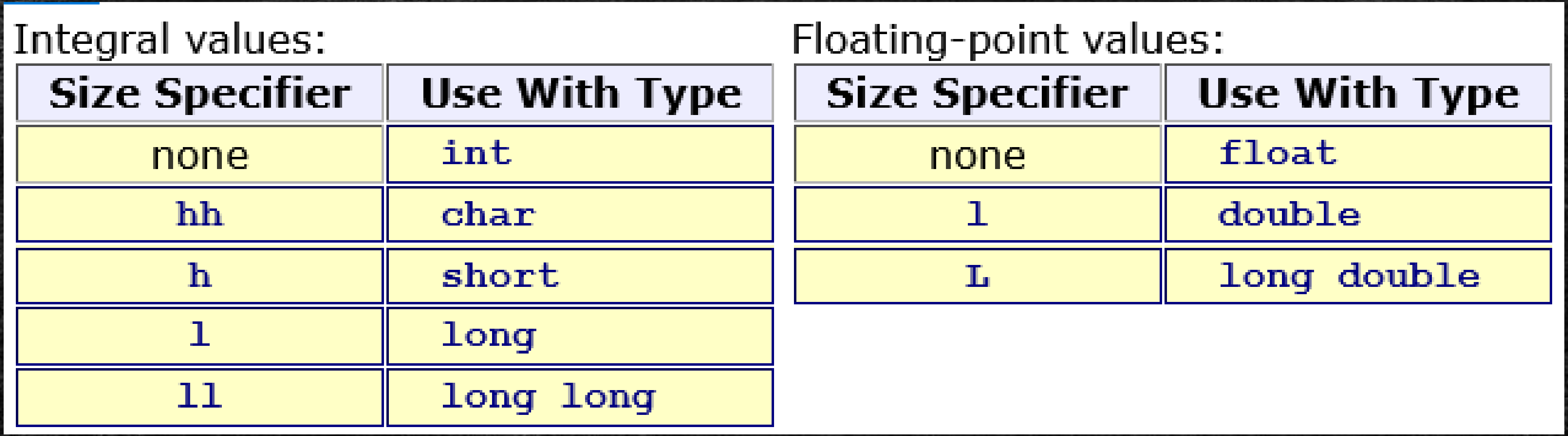
Conversion Controls

- We refine the output by inserting control characters between the % symbol and the conversion character.
- The general form of a conversion specification is

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

- The five control characters are
  - flags
    - - prescribes left justification of the converted value in its field
    - 0 pads the field width with leading zeros
  - 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 is space or 0 if the padding flag is on
  - . separates the field's width from the field's precision
  - 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 an f conversion
  - size identifies the size of the type being output

Conversion Controls



Integral values:

Size Specifier	Use With Type
none	int
hh	char
h	short
l	long
ll	long long

Floating-point values:

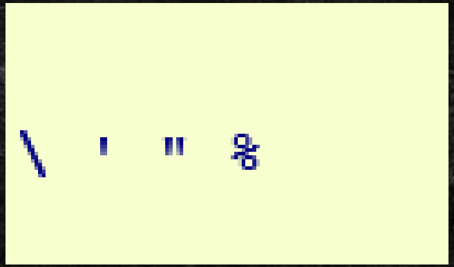
Size Specifier	Use With Type
none	float
l	double
L	long double

Special Characters

- To insert the special characters `\`, `'`, and `"`, we use their escape sequences.
- To insert the special character % into the format, we use the % symbol:

```
// Outputting special characters
// special.c

int main(void)
{
        printf("\\ \' \" %%\n");
        return 0;
}
```



```
\ ' " %
```