**Lecture 23 - Chapter 9: C Formatted Input/Output – Wed Nov 8 or Thurs Nov 9**

**Announcements**

Reading:

* Chapter 9

Assignments:

* Due: Assignment #9
* Assign: Assignment #10 - due on **Nov 20** (MW class) or **Nov 21** (TR class) **(no late assignments accepted)**

**Today’s Goals**

1. Streams
2. Formatting Output with printf
3. Printing Integers
4. Printing Floating-Point Numbers
5. Printing Strings and Character
6. Other Conversion Specifiers
7. Printing with Field Widths and Precision

**Today’s Terminology 5 min**

**Terminology**

* Stream
  + A sequence of bytes of data
* Input Stream
  + Sequence of bytes flowing into program
* Output Stream
  + Sequence of bytes flowing out of the program
* Format Control String
  + Used in printf and scanf statements to describe the format of the output or type of input
* Conversion Specification
  + Begins with a percent sign (%)
  + Ends with conversion specifier (d, f, s, …)
* Literal Characters
  + A literal is a value like the character ‘a’
* Assignment Suppression Character - \*
  + Allows scanf to read data from the input stream and discard it without assigning it to a variable

**Streams**

**Streams**

* Used to move data into and out of a program
* All input and output is performed using streams:
  + Input stream
    - Data flows from a device like a keyboard, file, etc.
  + Output stream
    - Data flows to a device like the screen, printer, file, etc.
  + Error stream
    - Data flows to a device like the screen, printer, file, etc.

**Standard Streams in Programs**

* 3 streams are automatically opened when program execution begins
  + standard input - keyboard
  + standard output - screen
  + standard error - screen
* Standard input and output streams can be redirected to other devices (chapter 11)

**Formatting**

* In this chapter, we look at formatting with printf and scanf
* printf – for precise output formatting
* scanf – for precise input formatting

**Printing Output with printf**

**Format Control String**

* Used to describe the format of the output
* Contains:
  + Conversion specifiers
  + Field widths,
  + Flags,
  + Precisions, and
  + Literal characters

**printf Formatting Capabilities**

* printf can be used to do the following:
  + Rounding floats
  + Aligning numbers with decimal points
  + Right or left justification of output
  + Inserting literal characters in output
  + Representing floats in exponential format
  + Representing unsigned integers in octal or hexadecimal format
  + Displaying all types of data with field widths and precisions

**General Format**

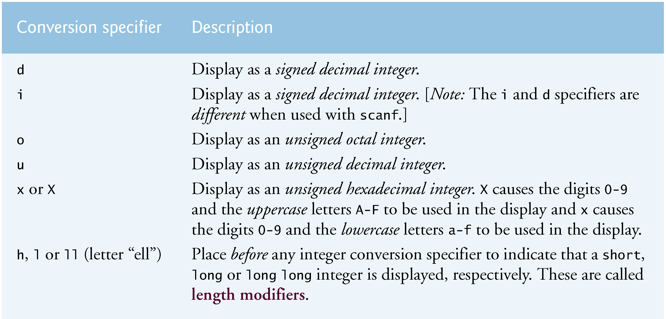
* printf (format-control-string, other-arguments);

describe how to format output values for each conversion specification in format-control string

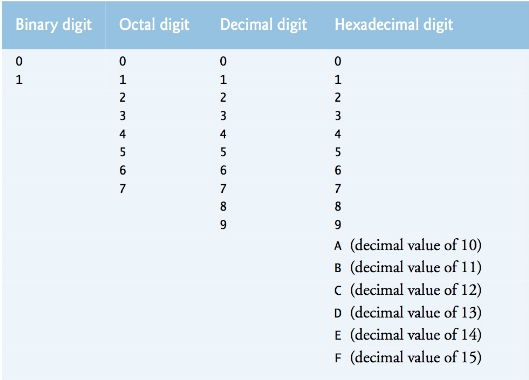
**Printing Integers**

**Integer Conversion Specifiers**

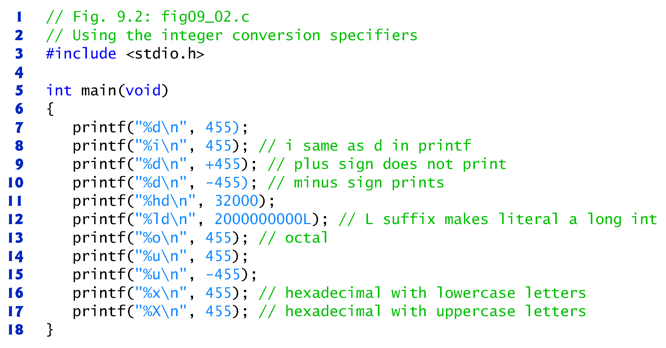
* Figure 9-1 contains the integer conversion specifiers



* Appendix C contains information on the different number systems



* **Example**
  + The following code shows the different specifiers in action



* + The output is:

455

455

455

-455

32000 <- **printf (“hd\n”, 32000)** -- **%hd** the ‘h’ in front of the ‘d’ indicates short

2000000000L <- **printf (“ld\n”, 2000000000L)**

**%ld** the ‘l’ in front of ‘d’ indicates a long integer

The **L** on **2000000000L** makes the literal a long integer

707 **<- printf (“%o\n”, 455);**

Octal of 455 is 707

|  |  |  |  |
| --- | --- | --- | --- |
| 83 | 82 | 81 | 80 |
| 512 | 64 | 8 | 1 |

512 is too much so start with 64

How many 64’s are there in 455?

455/64 = **7**

This means 64\*7 = 448

How much is left in 455 once we remove the 448?

455-448 = **7** left

|  |  |  |  |
| --- | --- | --- | --- |
| 83 | 82 | 81 | 80 |
| 512 | 64 | 8 | 1 |
| 0 | 7 | 0 | 7 |

455 **<- printf (“%u\n”, 455);**

4,294,966,841 **<- printf (“%u\n”, -455);**

unsigned decimal of -455

How did they get 4,294,966,841???

First, an unsigned int is 4 bytes = 32 bytes, so **455** in binary is:

231 230 229 228 ……215 214 213 212 211 210 29 28  27 26 25 24 23 22 21 20

0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1

To get **-455**, first flip each bit

231 230 229 228 ………… 215 214 213 212 211 210 29 28  27 26 25 24 23 22 21 20

1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0

Next add 1

231 230 229 228 ………… 215 214 213 212 211 210 29 28  27 26 25 24 23 22 21 20

1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 1

This is the binary representation (two’s complement) of -455

See Appendix C – C.6

If you calculate it out it is 4,294,966,841

Which is the result of **Printf(“%u\n”, -455);**

Two’s complement

Used to represent negative numbers

Makes addition/subtraction simple

https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

1c7 **<- printf (“%x\n”, 455);**

hexadecimal of 455 is 1c7 (with lowercase letters -- %x)

1C7 **<- printf (“%X\n”, 455);**

hexadecimal of 455 is 1C7 (with uppercase letters -- %X)

|  |  |  |  |
| --- | --- | --- | --- |
| 163 | 162 | 161 | 160 |
| 4096 | 256 | 16 | 1 |

How many 256’s are in 455?

**455/256 = 1.7**

This means 256\*1 = 256

How much is left in 455 once we remove the 256?

455 – 256 = 199

How many 16’s are in 199?

**199/16 = 12.4**

This means 12\*16 = 192 (**123456789ABCDEF**

How much is left in199 once we remove 192

199 – 192 = 7

How many 1’s are in 7?

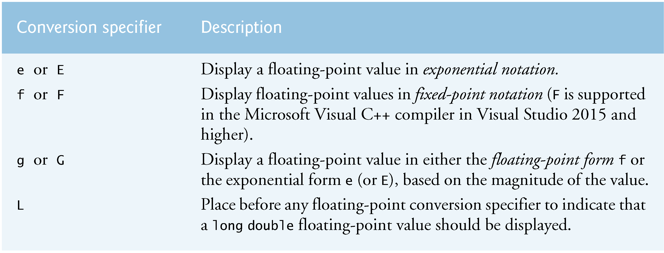
**7/1 = 7**

|  |  |  |  |
| --- | --- | --- | --- |
| 163 | 162 | 161 | 160 |
| 4096 | 256 | 16 | 1 |
| 0 | 1 | c | 7 |

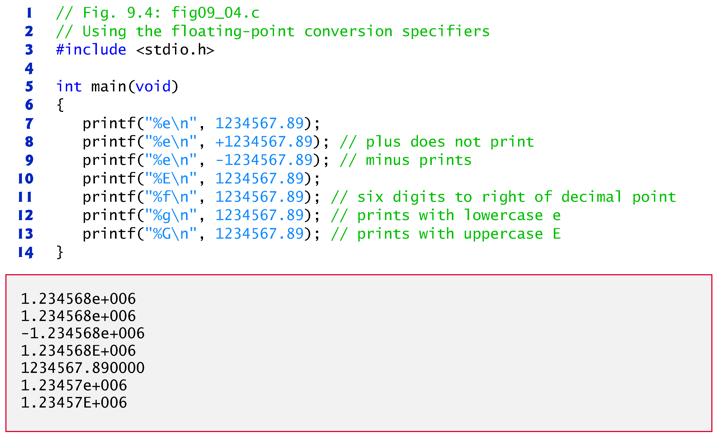
**Printing Floating-Point Numbers**

**Floating-point Conversion Specifiers**

* Figure 9-3 contains the floating-point conversion specifiers



* **%e** or **%E**
  + Way to display floating point values in exponential notation
  + e – displays a lowercase e
  + E – displays an uppercase E
  + Example
    - 150.4582 <- floating-point notation
    - 1.504582 X 102 <- scientific notation
    - 1.504582E+02 <- exponential notation
  + e, E (and f) provide 6 digits of precision to right of decimal
* **%g** or **%G**
  + Use to get rid of the trailing zeros
  + If exponent < -4 or >= 6 prints in exponential notation otherwise floating-point notation
* **%L**
  + Use when dealing with long double
* **Example**
  + The following code shows **e, E, f, g, G** specifiers in action



* + Looking at
    - **printf** ("%e\n", 1234567.89);
      * 1.234568e+006
      * Displayed in exponential notation
      * Shows 6 digits of precision
    - **printf** ("%f\n", 1234567.89);
      * 1234567.890000
      * Shows 6 digits of precision
    - **printf** ("%g\n", 1234567.89);
      * 1.23457e+006
      * Displayed in exponential notation since exponent is >= 6

**Printing Strings and Characters**

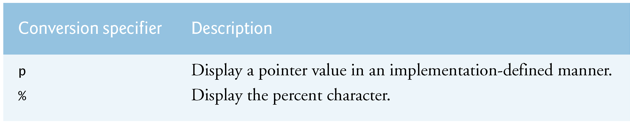
**String and Character Conversion Specifier**

* **%s** for strings
  + Requires a pointer to a char as argument
    - char \*buffer;
    - char buffer[10];
  + Causes characters to be printed until ‘\0’ is reached
* **%c** for characters
  + Requires a char argument
* Common errors with %s and %c
  + Using %c to print a string – syntax error
  + Using %s to print a char – run time error
  + Writing “a” instead of ‘a’ – syntax error
    - “a” is a string so that means there are 2 characters – the ‘a’ and the ‘/0’
    - ‘a’ is a character so that means there is 1 character!

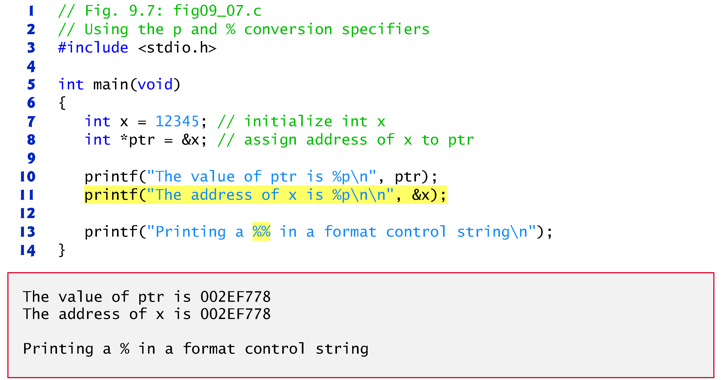
**Other Conversion Specifiers**

**Other Conversion Specifiers**

* Figure 9-6 contains the p and % specifiers



* **Example of p and %**



**Printing with Field Widths and Precision**

**Field Width**

* Use a field width to specify the exact size of the printed data
* Specify by placing the “field width” between the **%** and **conversion specifier**
  + %**5**d
  + %**80**s
  + %**2.2**f
* If field width is larger than data
  + Right justify
* If field width is smaller than data
  + Field width is increased
  + Creates different offsets than what may be expected

**Field Width Example**

* From book Figure 9-8

**printf** ("%4d\n", 1);

**printf** ("%4d\n", 1234);

**printf** ("%4d\n", 12345);

**Displays**

1 Right justified since only needed one space

1234 Fits into the 4 spaces

12345 Bigger than 4 spaces, added an extra space to handle

**Precision**

* This term means different things based on the type
  + Integers (d)
    - **Min** number of digits to be printed
    - If value is less than stated precision, appends zeros
    - If value is more than state precision, just ignores precision
    - Default is 1
  + Floats (f, e, E)
    - Number of digits after decimal point
  + Strings (s)
    - **Max** number of characters from beginning of string to be printed
* Specify by placing decimal point (.) followed by integer representing precision between **%** and **conversion specifier**
  + %**.5**d
  + %**.10**s
  + %**.2**f

**Precision Example**

* For integers

**printf** ("%.4d\n", 87);

**printf** ("%.4d\n", 12345);

**Displays**

0087

12345

* For floats

**printf** ("%.2f\n", 87.1);

**printf** ("%.2f\n", 875.125);

**Displays**

87.10

875.13 <- rounded up when more values than stated in precision

* For strings

**printf** ("%.5s\n", "Test");

**printf** ("%.3s\n", "Test");

**Displays**

Test <- not right justified

Tes

**Field Width and Precision Example**

* Can specify the width of the field and the precision together

**printf** ("%2.2f\n", 87.1);

**printf** ("%2.2f\n", 875.125);

**Displays**

87.10

875.13 <- field width was not big enough so it was increased, also rounded