FORMATTED OUTPUT

To accomplish formatted output to stdout (your monitor), we will use the printf() function. The declaration (prototype) of printf() is provided below.

```
int printf(const char *format, arg1, arg2, ...);
```

These printf() function takes the format string specified by the *format* argument and applies each following argument to the format specifiers in the string in a left to right fashion. Each character in the format string is copied to stdout except the for conversion characters which specify the format specifier.

The format specifiers are used to translate the internal values (arg1, arg2, ...) into formatted characters to be copied to stdout. Each format specifier begins with a % and ends with a conversion character type. Between the % and the conversion character type, the following my exist (in order):

```
[flags]          (OPTIONAL) - Control the conversion
[width]          (OPTIONAL) - Defines the number of characters
                 to print
[.precision]     (OPTIONAL) - Defines the amount of precision to
                 print for a number type
[modifier]       (OPTIONAL) - Overrides the size (type) of the
                 argument
[type]           (REQUIRED) - The type of conversion to be
                 applied
```

FLAGS – The table below defines the *flags* that may be used.

FLAG	MEANING
-	The item is left-justified. It is printed beginning at the left of the field used for the item.
+	Signed values are to display a plus sign if positive and a minus sign if negative.
space	Signed values are displayed with a leading space (but no sign) if positive and with a minus sign if negative. Note: a + flag will override the *space* flag.
#	Use alternative form for the conversion specification: %#o produce initial 0 (zero) for printing octal values %#x produce initial 0x for printing hex values %#f guarantee decimal point is printed for floating values %#g prevent trailing zeros from being automatically removed
0 *(zero)*	For numeric forms, pad the field width for the item with leading zeros instead of white space. This flag is ignored if a – flag is present or if, for an integer form, a precision is specified.

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

WIDTH – The *width* of the field is specified here with a decimal value. If the value is not large enough to fill the width, then the rest of the field is padded with spaces (unless the 0 flag is specified). If the value overflows the width of the field, then the field is expanded to fit the value. If a ***** is used in place of the width specifer, then the next argument (which must be an **int** type) specifies the width of the field. Note: when using the ***** with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted.

For example,

```
#include <stdio.h>
int main(void)
{
      int num=7, width=10;
      printf("%0*d\n", width, num);
      printf(“%015d”, num);

      return 0;
}
```

Produces the following output:

```
Y:\apc-cc-conduit\Images>a
0000000007
000000000000007
Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

.PRECISION – The *.precision* is used to specify one of the following:

1. The maximum number of characters to be printed from a string
2. The number of digits after the decimal point of a floating-point value
3. the minimum number of digits of an integer

For example,

```
#include <stdio.h>
#define COLUMN_COUNT "\n12345678901234567890123456789\n"

int main(void)
{
      int   iNum  = 12345;
      float fNum  = 1.2345;
      char  *pStr = “abcdefghijklmnopqrstuvwxyz”;

      printf(“\nWith .precision = .3 %s", COLUMN_COUNT);
      printf(“%.3s\n%.3d\n%.3f\n\n”, pStr, iNum, fNum);

      printf(“\nWith .precision = .10 %s", COLUMN_COUNT);
      printf(“%.10s\n%.10d\n%.10f\n”, pStr, iNum, fNum);

      printf(“\nWithout .precision %s", COLUMN_COUNT);
      printf(“%s\n%d\n%f\n”, pStr, iNum, fNum);

      return 0;
}
```

Produces the following output:

```
Y:\apc-cc-conduit\images>a

With .precision = .3
12345678901234567890123456789
abc
12345
1.235


With .precision = .10
12345678901234567890123456789
abcdefghij
0000012345
1.2345000505

Without .precision
12345678901234567890123456789
abcdefghijklmnopqrstuvwxyz
12345
1.234500

Y:\apc-cc-conduit\images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

MODIFIER – A *modifier* changes the way a conversion specifier type is interpreted as specified in the table below.

MODIFIER	TYPE	EFFECT
h	d, i, o, u, x, X	Value is first converted to a short int or to a unsigned short int
h	p	Specifies that the pointer is pointing to a short int
l	d, i, o, u, x, X	Value is first converted to a long int or to a unsigned long int
l	p	Specifies that the pointer is pointing to a long int
L	e, E, f, g, G	Value is first converted to a long double

TYPE – The *conversion specifier types* for the printf() function are shown in the table below. They mark the end of the format specifier.

TYPE (Conversion Specifier)	OUTPUT
%c	Single character
%d	Signed decimal integer
%e	Floating-point number, e-notation
%E	Floating-point number, E-notation
%f	Floating-point number, decimal notation
%g	Floating-point number, Use of %f or %e – whichever is shorter
%G	Floating-point number, Use of %f or %E – whichever is shorter
%i	Signed decimal integer
%o	Signed octal integer
%p	A pointer
%s	Character string
%u	Unsigned decimal integer
%x	Unsigned hexadecimal integer, using hex digits 0 - f
%X	Unsigned hexadecimal integer, using hex digits 0 - F
%%	Print a percent sign

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

FORMATTED INPUT

To accomplish formatted input from stdin (from your keyboard), we will use the scanf() function. The declaration (prototype) of scanf() is provided below.

```
int scanf(const char *format, &arg1, &arg2, ...);
```

These functions take input in a manner that is specified by the format argument and store each input field into the following arguments in a left to right fashion. Each input field is specified in the format string with a format specifier that specifies how the input is to be stored in the appropriate variable.

Any other characters in the format string specify characters that must be matched (typed in) from the input. They are not stored in any of the arguments to the scanf() call. If the input does not match then the function stops scanning and returns. A whitespace character may match with any whitespace character (space, tab, carriage return, new line, vertical tab, or formfeed) or the next incompatible character

An input field is defined with a format specifier. Each format specifier begins with a % and ends with a conversion character type. Between the % and the conversion character type, the following my exist (in order):

```
[*]             (OPTIONAL) – Suppress assignment
[width]         (OPTIONAL) - Defines the maximum number of
                characters to be read
[modifier]      (OPTIONAL) - Overrides the size (type) of the
                argument
[type]          (REQUIRED) - The type of conversion to be
                applied
```

*** (Assignment suppressor)** – The assignment suppressor causes the input field to be scanned but not stored in a variable. That is, the item is skipped.

```
For Example,
      /* Scan in one character, but do not store */
      scanf("%*c");
```

Width:

The maximum field width. The input stops when the maximum field width is reached or when the first whitespace character is encountered, whichever comes first.

```
For Example,
      /* Scan in a string of up to 20 characters */
      scanf("%20s", str);
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

Modifier:
A modifier changes the way a conversion specifier type is interpreted as shown in the table below.

MODIFIER	TYPE	EFFECT
h	d, i, o, u, x, X	Value is to be stored in a short int or in unsigned short int
l	d, i, o, u, x, X	Value is to be stored in a long int or in a unsigned long int
l	e, E, f, g, G	Value is to be stored in a double
L	e, E, f, g, G	Value is to be stored in a long double

TYPE – The *conversion specifier types* for the scanf() function are shown in the table below. They mark the end of the format specifier.

TYPE (Conversion Specifier)	OUTPUT
%c	Interpret as a single character
%d	Interpret as a signed decimal integer
%e	Interpret as a floating-point number
%E	Interpret as a floating-point number
%f	Interpret as a floating-point number
%g	Interpret as a floating-point number
%G	Interpret as a floating-point number
%i	Interpret as a signed decimal integer
%o	Interpret as a signed octal integer
%p	Interpret as a pointer (an address)
%s	Interpret as a character string. The input begins with the first nonwhitespace character and includes everything up to the next whitespace character.
%u	Interpret as an unsigned decimal integer
%x	Interpret as an unsigned hexadecimal integer
%X	Interpret as an unsigned hexadecimal integer

All examples, homeworks, exams, and projects shall use formatted Input/Output from this point onward. You are encouraged to experiment with different formatting per the tables defined in this lecture.

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com