Formatted Output and Input

K&R: Chapter 7, especially §7.2;

Appendix B1.2

(For Contrast – Basic Output)

- Basic functions send ASCII text to output
- **putchar()** – prints a single character to stdout
  - accepts an integer/character as an argument
- **putc()**, **fputc()** – sends a single character to a "file handle"
- **puts()** – prints null-terminated text string to stdout
  - adds newline at end of string
- **fputs()** – sends one null-terminated text string to a "file handle"
  - no newline added!

Formatted Output

- Display of numeric output requires conversion to ASCII text
- **printf(** *fmt-str*, *…* **)** – general-purpose function to print all C datatypes with formatting
  - *fmt-str* is a required *formatting string*
  - Contains *conversion specifier* items
  - Includes optional constant text
  - *…* are additional arguments as needed, depending on the conversion specifiers
- example: **printf("answer is %d\n", num)**
  - The "**%d**" pairs with the argument "**num**"

Try these in a program –

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("%d \n", -8);
    printf("%u \n", -8); //?
    printf("%f \n", -8); //?
    printf("--------\n", -8); //?

    printf("%d \n", -8.5); //?
    printf("%u \n", -8.5); //?
    printf("%f \n", -8.5);
    printf("--------\n", -8); //?

    printf("%s \n", "-8.5");
    printf("%s \n", -8); //?
    printf("%s \n", -8.5); //?
    return 0;
}
```

- Is the compiler satisfied?
- What happens with the integer "-8" ?
- What happens with the floating-point value "-8.5" ?
- What happens when you treat them like strings?
- *The moral: Make sure the conversion specifier matches the type of the value it pairs with!*

C's Conversion Specifier – "%fw.plT"

- *Conversion specifier* : a piece of text containing up to six symbols, in order:
  - `%` – marks the start of the specifier
  - `f` – *optional* flag
  - `w` – *optional* minimum field width
  - `.p` – *optional* precision
  - `l` – *optional* length modifier
  - `T` – ***required*** type – must match its argument
- Examples of "`%fw.plT`" ...
  - `%4li` `%+6d`
    `%08.3f` `%-6.3Lg`
    `%c` `%#08x`

Conversion Types

%fw.plT

Conversion Specifier	Description
%d, %i	Signed integer value, base-10.
%u	Unsigned integer value, base-10.
%x, %X	Signed value, base-16 (hexadecimal) (lowercase, UPPERCASE).
%o	Signed value, base-8 (octal).
%c	Single character (accepts integer).
%f	Floating-point fixed-decimal-point format.
%e, %E	Floating-point exponential format (lowercase, UPPERCASE).
%g, %G	Floating-point format. Uses lowercase/UPPERCASE exponential format if exponent is less than -4 or not less than precision, fixed-decimal-point otherwise.
%s	String (null-terminated array of characters).
%p	Pointer (memory address).
%n	*Int pointer; returns number of chars written. No output.*
%%	No argument is converted; results in a '%' character.

Length modifier

%fw.pIT

- Optional letter indicating the expected size of the corresponding type
- Modifiers only apply to some types

Length Modifier	Description
`h`	**%hi** – Short integer.
`l`	**%li** – Long integer, **%lf** – "long float" (double).
`L`	**%Lf** – Long double.

long and short types

%fw.pIT

- long int, long unsigned
  - Theoretically twice as many bits as int/unsigned ; only guaranteed to be "at least as many bits"
  - Conversion specifiers: %ld %li %lu
- short int, short unsigned
  - Theoretically half as many bits as int/unsigned ; only guaranteed to be "no more bits"
  - Conversion specifiers: %hd %hi %hu
- double, long double
  - More precision, bigger exponents than float
  - Conversion specifiers:
    %lf %le %lg , %Lf %Le %Lg

Conversion Field Width & Precision

%f**w.p**lT

- Field width **w**
  - Minimum number of characters to use for representation
  - Padded with blanks unless otherwise specified
  - Left- or right- justified based on type
- precision **.p**
  - For float types (f, e, g, F, E, G): number of decimal places
  - No meaning for integer, string types:
- Each is optional
- Each can be given as *****
  - meaning: read value from the argument list

Changing the Field Width

%f**w.p**lT'

- The field width can be specified as another argument, identified by a '*'
- Example:

```
printf( "%5.2f", 1.33333)
```

can also be written as

```
w = 5
p = 2
printf( "%*.*f", w, p, 1.33333)
```

- This provides for *computed* field widths

Conversion Flags

%fw.plT

Flag	Description
'#'	Use alternate form (varies with specifier).
'0'	Zero-pad numbers on left.
'-'	Left-justify the value.
' '	Pad positive numbers with a space.
'+'	Add + or – to numeric values.

- *Optional* flags, inserted between the '%' and the rest of the conversion specifier

Example: Combining Conversion Modifiers

%fw.plT

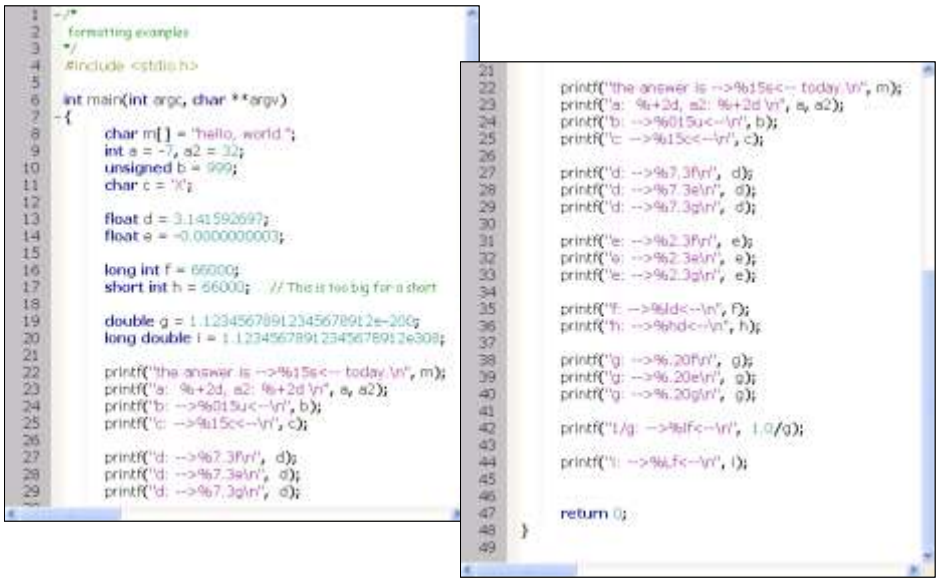
- Field-width modifiers:
  - ***w*** field is *w* characters
  - ***0w*** *n*-character field, left padded w/ zeros
  - ***.p*** number has *p* decimal places
  - ***w.p*** field is *w* characters, including *p* decimals
- Example:

```
printf("%4s and %05.1f", "ab", 1.65432)
```

gives

```
"  ab and 001.7"
```

Conversion Specifications in Action:



```
/*
 formatting examples
*/
#include <stdio.h>

int main(int argc, char **argv)
{
        char m[] = "hello, world";
        int a = -7, a2 = 32;
        unsigned b = 999;
        char c = 'X';

        float d = 3.141592697;
        float e = -0.0000000003;

        long int f = 66000;
        short int h = 66000;    // This is too big for a short

        double g = 1.12345678912345678912e-200;
        long double i = 1.12345678912345678912e308;

        printf("the answer is -->%15s<-- today.\n", m);
        printf("a:  %+2d, a2: %+2d \n", a, a2);
        printf("b: -->%015u<--\n", b);
        printf("c: -->%15c<--\n", c);

        printf("d: -->%7.3f\n",  d);
        printf("d: -->%7.3e\n",  d);
        printf("d: -->%7.3g\n",  d);

        printf("e: -->%2.3f\n",  e);
        printf("e: -->%2.3e\n",  e);
        printf("e: -->%2.3g\n",  e);

        printf("f: -->%ld<--\n", f);
        printf("h: -->%hd<--\n", h);

        printf("g: -->%.20f\n",  g);
        printf("g: -->%.20e\n",  g);
        printf("g: -->%.20g\n",  g);

        printf("1/g: -->%lf<--\n",  1.0/g);

        printf("i: -->%Lf<--\n", i);


        return 0;
}
```

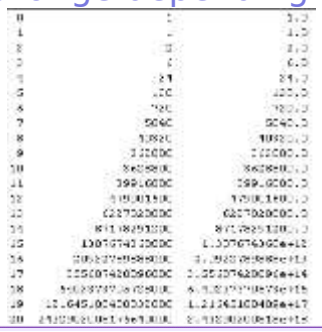
Relatives of printf()

- fprintf(file, str, …)
  - sends its output to an opened disk file instead of to stdout
- sprintf(strptr, str, …)
  - writes null-terminated output into a supplied string
  - The string *must* be long enough, or bad things can happen!

Activity

- Write a program that produces a table showing n, factorial(n), and real_factorial(n) to one decimal place, for all values of n up to (and including) a value that the user specifies.
- Each column should be just wide enough for the largest value – this will change depending on what the user puts in!

0	1	1.0
1	1	1.0
2	2	2.0
3	6	6.0
4	24	24.0
5	120	120.0
6	720	720.0



Hints

- How wide should the column be? – How many digits in the answer?
  - The log10() function and the floor() function can be used to answer this
- Why real_factorial()?
  - Which version handles larger values? (n>20)

The factorial( ) function

- Integer- and floating-point-based versions of the factorial function
- *Which has the bigger valid range?*

```
int fact_int(int n)
{
   int f = 1;
   for (int i = 1; i <= n; i++)
      f *= i;
   return f;
}

double fact_double(double n)
{
   double f = 1.0;
   for (double i = 1.0; i <= n; i++)
      f *= i;
   return f;
}
```

comparable to printf()

see §7.4

SCANF() – FORMATTED INPUTS

scanf( )

- Alternative to reading a string and converting
- Uses a *conversion specifier*, like printf() does
- Simple usage for numeric inputs:
  - `scanf(" %d", &int_variable);`
  - `scanf(" %c", &char_variable);`
  - `scanf(" %f", &float_variable);`
  - `          ^       ^`
- The space helps in the format string
- The ampersand is important in on the variable name
  - but NOT needed for inputting character strings

Try these in a program –

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int a;
    double b;
    char mystring[10];

    printf("Enter an integer: ");
    scanf(" %d", &a);

    printf("Enter a float: ");
    scanf(" %lf", &b);

    printf("Enter some text: ");
    scanf(" %10s", mystring);

    printf("%s\n %lf\n %d\n",
        mystring, b, a);

    return 0;
}
```

- scanf():
  - the numeric variable arguments need the "&" (*address-of* operator)
  - the text-string variable **doesn't** need the "&"
  - The single space before each of the conversion specifiers absorbs any leading whitespace in the input stream.

More scanf() details

- scanf() can read more than one item at a time
  - use this to "parse" an input line
- scanf()'s return value shows how many items are successfully read
  - Return value is "EOF" (End Of File) if error occurs, such as wrong data type (string input instead of expected number)
- Single whitespace in formatting string handles variable whitespace in inputs
  - ...including *no* whitespace when field lengths are specified
- Always specify a maximum length when reading string inputs!
  - Otherwise you risk a buffer overflow, as with gets()

Multiple inputs From one scanf()

No "&" used for char-string arguments

spaces in format-string

Maximum field width is given

```c
/* Multiple scanf() input example
* 2016-08-21
*/
#include<stdio.h>

int main(int argc, char **argv)
{
   char first[10], last[30];
   int age;
   int retval;

   printf("Enter your first and last names, and age: ");

   retval = scanf( " %10s %30s %d", first, last, &age);

   if (retval != 3) {
      printf("You only entered %d values!\n", retval);
   } else {
      printf("%s, %s - %d years old.\n", last, first, age);
   }

   return 0;
}
```

Try this version –

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int a;
    double b;
    char mystring[10];

    printf("Enter an integer, a float, and some text: ");
    scanf(" %d %lf %10s", &a, &b, mystring);

    printf("%s\n %lf\n %d\n", mystring, b, a);

    return 0;
}
```

Add a bit of error-checking –

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int a;
    double b;
    char mystring[10];
    int return_value;

    printf("Enter an integer, a float, and some text: ");
    return_value = scanf(" %d %lf %10s", &a, &b, mystring);
    if (return_value != 3)
        printf("Only %d values entered!\n",
            return_value);

    printf("%s\n %lf\n %d\n", mystring, b, a);

    return 0;
}
```