Data Input and Output

LIBRARY FUNCTIONS

- **are prewritten routines that carry out various commonly used operations or calculations.**
- **They are contained within one or more library files.**
- **In C ,Library-function declarations in a special source files, called *header files.***
- **During the process of converting a C source program into an executable object program, the compiled source program may be linked with one or more library file*s to produce the final executable program***
- ***#include – used to include header files***

LIBRARY FUNCTIONS

Function	*Type*	*Purpose*
`abs(i)`	int	Return the absolute value of `i`.
`ceil(d)`	double	Round up to the next integer value (the smallest integer that is greater than or equal to `d`).
`cos(d)`	double	Return the cosine of `d`.
`cosh(d)`	double	Return the hyperbolic cosine of `d`.
`exp(d)`	double	Raise *e* to the power `d` (*e* = 2.7182818 · · · is the base of the natural (Naperian) system of logarithms).
`fabs(d)`	double	Return the absolute value of `d`.
`floor(d)`	double	Round down to the next integer value (the largest integer that does not exceed `d`).
`fmod(d1,d2)`	double	Return the remainder (i.e., the noninteger part of the quotient) of `d1/d2`, with same sign as `d1`.
`getchar()`	int	Enter a character from the standard input device.
`log(d)`	double	Return the natural logarithm of `d`.
`pow(d1,d2)`	double	Return `d1` raised to the `d2` power.
`printf(...)`	int	Send data items to the standard output device (arguments are complicated — see Chap. 4).
`putchar(c)`	int	Send a character to the standard output device.

LIBRARY FUNCTIONS

`rand()`	int	Return a random positive integer.
`sin(d)`	double	Return the sine of `d`.
`sqrt(d)`	double	Return the square root of `d`.
`srand(u)`	void	Initialize the random number generator.
`scanf(...)`	int	Enter data items from the standard input device (arguments are complicated – see Chap. 4).
`tan(d)`	double	Return the tangent of `d`.
`toascii(c)`	int	Convert value of argument to ASCII.
`tolower(c)`	int	Convert letter to lowercase.
`toupper(c)`	int	Convert letter to uppercase.

Type refers to the data type of the quantity that is returned by the function.

`c` denotes a character-type argument

`i` denotes an integer argument

`d` denotes a double-precision argument

`u` denotes an unsigned integer argument

Write a C program to solve roots of quadratic equation

$$ax^2 + bx + c = 0$$

using the well-known quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
main()   /* solution of a quadratic equation */
{
    double a,b,c,root,x1,x2;

    /* read values for a, b and c */

    root = sqrt(b * b - 4 * a * c);
    x1 = (-b + root) / (2 * a);
    x2 = (-b - root) / (2 * a);

    /* display values for a, b, c, x1 and x2 */
}
```

Program to convert lowercase character in to uppercase

```
/* read a lowercase character and display its uppercase equivalent */

#include <stdio.h>
#include <ctype.h>

main()
{
    int lower, upper;

    lower = getchar();
    upper = toupper(lower);
    putchar(upper);
}
```

Data Input and Output

- C has no of functions for input and output.
- 6 functions
- getchar, putchar, scanf, printf , gets and puts.

SINGLE CHARACTER INPUT- getchar()

- The getchar function is a part of the standard C I/O library.
- It returns a single character from a standard input device (typically a keyboard).
- In general terms, a function reference would be written as

character variable = getchar ( ) ;

where character variable refers to some previously declared character variable.

Eg:-

```
char c;
. . . . .
c = getchar();
```

SINGLE CHARACTER OUTPUT – putchar()

- The putchar function, like getchar, is a part of the standard C I/O library.
- It transmits a single character to a standard output device (typically a TV monitor).
- In general, a function reference would be written as

  putchar (*character variable)*
- where *character variable refers to some previously declared character variable.*
- EXAMPLE
- A C program contains the following statements.
- char c;
- c= ‘a’;
- putchar(c);

stdin, stdout, stderr

- When your C program begins to execute, three input/output devices are opened automatically.
- stdin
  - The “standard input” device, usually your keyboard
- stdout
  - The “standard output” device, usually your monitor
- stderr
  - The “standard error” device, usually your monitor
- Some C library I/O functions automatically use these devices

Formatted Console Output

- In C formatted output is created using the `printf( )` function.
- `printf( )` outputs text to `stdout`
- The basic function call to `printf( )` is of the form

  ```
  printf( format, arg1, arg2, … );
  ```

  where the format is a string containing
  - conversion specifications
  - literals to be printed

printf( ) conversions

Conversions specifications begin with % and end with a conversion character.

Between the % and the conversion character MAY be, in order

- A minus sign specifying left-justification
- The minimum field width
- A period separating the field width and precision
- The precision that specifies
  - Maximum characters for a string
  - Number of digits after the decimal for a floating point
  - Minimum number of digits for an integer
- An h for “short” or an l (letter ell) for long

Conversion specifier	Description
`d`	Display as a signed decimal integer.
`i`	Display as a signed decimal integer. [*Note:* The `i` and `d` specifiers are different when used with `scanf`.]
`o`	Display as an unsigned octal integer.
`u`	Display as an unsigned decimal integer.
`x` or `X`	Display as an unsigned hexadecimal integer. `X` causes the digits `0-9` and the letters `A-F` to be displayed and `x` causes the digits `0-9` and `a-f` to be displayed.
`h` or `l` (letter `l`)	Place before any integer conversion specifier to indicate that a `short` or `long` integer is displayed, respectively. Letters `h` and `l` are more precisely called **length modifiers**.

Fig. 9.1 | Integer conversion specifiers.

```c
/* Fig 9.2: fig09_02.c */
/* Using the integer conversion specifiers */
#include <stdio.h>

int main( void )
{
   printf( "%d\n", 455 );
   printf( "%i\n", 455 ); /* i same as d in printf */
   printf( "%d\n", +455 );
   printf( "%d\n", -455 );
   printf( "%hd\n", 32000 );
   printf( "%ld\n", 2000000000L ); /* L suffix makes literal a long */
   printf( "%o\n", 455 );
   printf( "%u\n", 455 );
   printf( "%u\n", -455 );
   printf( "%x\n", 455 );
   printf( "%X\n", 455 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.2 | Using integer conversion specifiers.

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

Fig. 9.2 | Using integer conversion specifiers.

Only the minus sign prints; plus signs are suppressed.
Also, the value -455, when read by %u (line 15), is converted to the unsigned value 4294966841.

Conversion specifier	Description
`e` or `E`	Display a floating-point value in exponential notation.
`f`	Display floating-point values in fixed-point notation. [*Note:* In C99, you can also use `F`.]
`g` or `G`	Display a floating-point value in either the floating-point form `f` or the exponential form `e` (or `E`), based on the magnitude of the value.
`L`	Place before any floating-point conversion specifier to indicate that a `long double` floating-point value is displayed.

Fig. 9.3 | Floating-point conversion specifiers.

Values are printed with e (E) if, after conversion to exponential notation, the value's exponent is less than -4, or the exponent is greater than or equal to the specified precision (six significant digits by default for g and G).

Printing Floating-Point Numbers

- The %E, %e and %g conversion specifiers cause the value to be rounded in the output and the conversion specifier %f does not.
- Conversion specifier g (or G) prints in either e (E) or f format with no trailing zeros (1.234000 is printed as 1.234).
- The precision for conversion specifiers g and G indicates the maximum number of significant digits printed, including the digit to the left of the decimal point.
- The value 1234567.0 is printed as 1.23457e+06, using conversion specifier %g
- With some compilers, the exponent in the outputs will be shown with two digits to the right of the + sign.

```c
/* Fig 9.4: fig09_04.c */
/* Printing floating-point numbers with
   floating-point conversion specifiers */

#include <stdio.h>

int main( void )
{
   printf( "%e\n", 1234567.89 );
   printf( "%e\n", +1234567.89 );
   printf( "%e\n", -1234567.89 );
   printf( "%E\n", 1234567.89 );
   printf( "%f\n", 1234567.89 );
   printf( "%g\n", 1234567.89 );
   printf( "%G\n", 1234567.89 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.4 | Using floating-point conversion specifiers.

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

Fig. 9.4 | Using floating-point conversion specifiers.

9.6 Printing Strings and Characters

- The `c` and `s` conversion specifiers are used to print individual characters and strings, respectively.
- Conversion specifier `c` requires a `char` argument.
- Conversion specifier `s` requires a pointer to `char` as an argument.
- Conversion specifier `s` causes characters to be printed until a terminating null (`'\0'`) character is encountered.
- The program shown in Fig. 9.5 displays characters and strings with conversion specifiers `c` and `s`.

```c
/* Fig 9.5: fig09_05c */
/* Printing strings and characters */
#include <stdio.h>

int main( void )
{
   char character = 'A'; /* initialize char */
   char string[] = "This is a string"; /* initialize char array */
   const char *stringPtr = "This is also a string"; /* char pointer */

   printf( "%c\n", character );
   printf( "%s\n", "This is a string" );
   printf( "%s\n", string );
   printf( "%s\n", stringPtr );
   return 0; /* indicates successful termination */
} /* end main */
```

```
A
This is a string
This is a string
This is also a string
```

Fig. 9.5 | Using the character and string conversion specifiers.

9.7 Other Conversion Specifiers

- The three remaining conversion specifiers are **p**, **n** and **%** (Fig. 9.6).
- The **conversion specifier %n** stores the number of characters output so far in the current **printf**—the corresponding argument is a pointer to an integer variable in which the value is stored—nothing is printed by a **%n**.
- The conversion specifier **%** causes a percent sign to be output.

Conversion specifier	Description
`p`	Display a pointer value in an implementation-defined manner.
`n`	Store the number of characters already output in the current `printf` statement. A pointer to an integer is supplied as the corresponding argument. Nothing is displayed.
`%`	Display the percent character.

Fig. 9.6 | Other conversion specifiers.

9.7 Other Conversion Specifiers (Cont.)

- Figure 9.7's %p prints the value of ptr and the address of x; these values are identical because ptr is assigned the address of x.
- Next, %n stores the number of characters output by the third printf statement (line 15) in integer variable y, and the value of y is printed.
- The last printf statement (line 21) uses %% to print the % character in a character string.

9.7 Other Conversion Specifiers (Cont.)

- **Every printf call returns a value—either the number of characters output, or a negative value if an output error occurs.**
- **[*Note: This example will not execute in Microsoft Visual C++ because %n has been disabled by Microsoft "for security reasons." To execute the rest of the program, remove lines 15–16.*]**

```c
/* Fig 9.7: fig09_07.c */
/* Using the p, n, and % conversion specifiers */
#include <stdio.h>

int main( void )
{
   int *ptr; /* define pointer to int */
   int x = 12345; /* initialize int x */
   int y; /* define int y */

   ptr = &x; /* assign address of x to ptr */
   printf( "The value of ptr is %p\n", ptr );
   printf( "The address of x is %p\n\n", &x );

   printf( "Total characters printed on this line:%n", &y );
   printf( " %d\n\n", y );

   y = printf( "This line has 28 characters\n" );
   printf( "%d characters were printed\n\n", y );

   printf( "Printing a %% in a format control string\n" );
   return 0; /* indicates successful termination */
} /* end main */
```

```
The value of ptr is 0012FF78
The address of x is 0012FF78

Total characters printed on this line: 38

This line has 28 characters
28 characters were printed

Printing a % in a format control string
```

9.8 Printing with Field Widths and Precision

- The exact size of a field in which data is printed is specified by a **field width**.
- If the field width is larger than the data being printed, the data will normally be right justified within that field.
- An integer representing the field width is inserted between the percent sign (%) and the conversion specifier (e.g., %4d).
- Figure 9.8 prints two groups of five numbers each, right justifying those containing fewer digits than the field width.
- The field width is increased to print values wider than the field and that the minus sign for a negative value uses one character position in the field width.
- Field widths can be used with all conversion specifiers.

```c
/* Fig 9.8: fig09_08.c */
/* Printing integers right-justified */
#include <stdio.h>

int main( void )
{
   printf( "%4d\n", 1 );
   printf( "%4d\n", 12 );
   printf( "%4d\n", 123 );
   printf( "%4d\n", 1234 );
   printf( "%4d\n\n", 12345 );

   printf( "%4d\n", -1 );
   printf( "%4d\n", -12 );
   printf( "%4d\n", -123 );
   printf( "%4d\n", -1234 );
   printf( "%4d\n", -12345 );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.8 | Right justifying integers in a field. (

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

More about Printf()

- **A *minimum* field width can be specified by preceding the conversion character by an unsigned integer**

```
#include <stdio.h>

main()        /* minimum field width specifications */

{
    int i = 12345;
    float x = 345.678;

    printf("%3d %5d %8d\n\n", i, i, i);
    printf("%3f %10f %13f\n\n", x, x, x);
    printf("%3e %13e %16e", x, x, x);
}
```

output

```
 12345 12345    12345

 345.678000 345.678000    345.678000

 3.456780e+02  3.456780e+02     3.456780e+02
```

```
#include <stdio.h>

main()        /* minimum field width specifications */

{
    int i = 12345;
    float x = 345.678;

    printf("%3d %5d %8d\n\n", i, i, i);
    printf("%3g %10g %13g\n\n", x, x, x);
    printf("%3g %13g %16g", x, x, x);
}
```

```
12345 12345    12345

345.678    345.678       345.678

345.678       345.678          345.678
```

MORE ABOUT THE printf()

- learned how to specify a minimum field width in a p r i n t f function.
- It is also possible to specify the maximum number of decimal places for a floating-point value, or the maximum number of characters for a string. This specification is known as *precision.*
- The precision is an unsigned integer that is always preceded by a decimal point.

EXAMPLE

- #include <stdio.h>
- main() /* display a floating-point number with several different precisions */
- {
- float x = 123.456;
- printf("%7f %7.3f %7.1f \n\n", x, x, x) ;

}

- When this program is executed, the following output is generated.

```
123.456000 123.456   123.5
```

```
#include <stdio.h>

main()

{
    char line[12];

    . . . . .

    printf("%10s %15s %15.5s %.5s", line, line, line, line);
}
```

the string **hexadecimal** is assigned to the character array **l i n e**

```
   hexadecimal      hexadecimal           hexad hexad
```

9.8 Printing with Precision

- Function **printf** also enables you to specify the precision with which data is printed.
- Precision has different meanings for different data types.
- When used with integer conversion specifiers, precision indicates the minimum number of digits to be printed.
- If the printed value contains fewer digits than the specified precision and the precision value has a leading zero or decimal point, zeros are prefixed to the printed value until the total number of digits is equivalent to the precision.
- If neither a zero nor a decimal point is present in the precision value, spaces are inserted instead.

9.8 Printing with Precision (Cont.)

- The default precision for integers is 1.
- When used with floating-point conversion specifiers **e**, **E** and **f**, the precision is the number of digits to appear after the decimal point.
- When used with conversion specifiers **g** and **G**, the precision is the maximum number of significant digits to be printed.
- When used with conversion specifier **s**, the precision is the maximum number of characters to be written from the string.
- To use precision, place a decimal point (.), followed by an integer representing the precision between the percent sign and the conversion specifier.

9.8 Printing with Precision (Cont.)

- A floating-point number will be *rounded* if it must be shortened to conform to a precision specification

```c
/* Fig 9.9: fig09_09.c */
/* Using precision while printing integers,
   floating-point numbers, and strings */
#include <stdio.h>

int main( void )
{
   int i = 873; /* initialize int i */
   double f = 123.94536; /* initialize double f */
   char s[] = "Happy Birthday"; /* initialize char array s */

   printf( "Using precision for integers\n" );
   printf( "\t%.4d\n\t%.9d\n\n", i, i );

   printf( "Using precision for floating-point numbers\n" );
   printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );

   printf( "Using precision for strings\n" );
   printf( "\t%.11s\n", s );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.9 | Using precis

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+002
        124

Using precision for strings
        Happy Birth
```

9.8 Printing with Field Widths and Precision

- **The field width and the precision can be combined by placing the field width, followed by a decimal point, followed by a precision between the percent sign and the conversion specifier, as in the statement**
  - `printf( '%9.3f', 123.456789 );`

  which displays 123.457 with three digits to the right of the decimal point right justified in a nine-digit field.
- **It's possible to specify the field width and the precision using integer expressions in the argument list following the format control string.**

9.8 Printing with Field Widths and Precision (Cont.)

- To use this feature, insert an asterisk (*) in place of the field width or precision (or both).
- The matching int argument in the argument list is evaluated and used in place of the asterisk.
- A field width's value may be either positive or negative (which causes the output to be left justified in the field as described in the next section).
- The statement
  - printf('%*.*f', 7, 2, 98.736);

  uses 7 for the field width, 2 for the precision and outputs the value 98.74 right justified.

9.9 Using Flags in the printf Format Control String

- Function **printf** also provides flags to supplement its output formatting capabilities.
- Five flags are available for use in format control strings (Fig. 9.10).
- To use a flag in a format control string, place the flag immediately to the right of the percent sign.
- Several flags may be combined in one conversion specifier.

Flag	Description
- (minus sign)	Left justify the output within the specified field.
+ (plus sign)	Display a plus sign preceding positive values and a minus sign preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix `0` to the output value when used with the octal conversion specifier `o`.
	Prefix `0x` or `0X` to the output value when used with the hexadecimal conversion specifiers `x` or `X`.
	Force a decimal point for a floating-point number printed with `e`, `E`, `f`, `g` or `G` that does not contain a fractional part. (Normally the decimal point is printed only if a digit follows it.) For `g` and `G` specifiers, trailing zeros are not eliminated.
0 (zero)	Pad a field with leading zeros.

Fig. 9.10 | Format control string flags.

9.9 Using Flags in the printf Format Control String (Cont.)

- **Figure 9.11 demonstrates right justification and left justification of a string, an integer, a character and a floating-point number.**

```c
/* Fig 9.11: fig09_11.c */
/* Right justifying and left justifying values */
#include <stdio.h>

int main( void )
{
   printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
   printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
     hello         7         a  1.230000

hello     7         a         1.230000
```

Fig. 9.11 | Left justifying strings in a field.

9.9 Using Flags in the printf Format Control String (Cont.)

- **Figure 9.12 prints a positive number and a negative number, each with and without the + flag.**
- **The minus sign is displayed in both cases, but the plus sign is displayed only when the + flag is used.**

```c
/* Fig 9.12: fig09_12.c */
/* Printing numbers with and without the + flag */
#include <stdio.h>

int main( void )
{
   printf( "%d\n%d\n", 786, -786 );
   printf( "%+d\n%+d\n", 786, -786 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
786
-786
+786
-786
```

Fig. 9.12 | Printing positive and negative numbers with and without the + flag.

9.9 Using Flags in the printf Format Control String (Cont.)

- **Figure 9.13 prefixes a space to the positive number with the space flag.**
- **This is useful for aligning positive and negative numbers with the same number of digits.**
- **The value -547 is not preceded by a space in the output because of its minus sign.**

```c
/* Fig 9.13: fig09_13.c */
/* Printing a space before signed values
   not preceded by + or - */
#include <stdio.h>

int main( void )
{
   printf( "% d\n% d\n", 547, -547 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
 547
-547
```

Fig. 9.13 | Using the space flag.

9.9 Using Flags in the printf Format Control String (Cont.)

- **Figure 9.14 uses the # flag to prefix 0 to the octal value and 0x and 0X to the hexadecimal values, and to force the decimal point on a value printed with g.**

```c
/* Fig 9.14: fig09_14.c */
/* Using the # flag with conversion specifiers
   o, x, X and any floating-point specifier */
#include <stdio.h>

int main( void )
{
   int c = 1427; /* initialize c */
   double p = 1427.0; /* initialize p */

   printf( "%#o\n", c );
   printf( "%#x\n", c );
   printf( "%#X\n", c );
   printf( "\n%g\n", p );
   printf( "%#g\n", p );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 9.14 | Using the # flag. (Part 1 of 2.)

```
02623
0x593
0X593

1427
1427.00
```

Fig. 9.14 | Using the # flag. (Part 2 of 2.)

9.9 Using Flags in the printf Format Control String (Cont.)

- **Figure 9.15 combines the + flag and the 0 (zero) flag to print 452 in a 9-space field with a + sign and leading zeros, then prints 452 again using only the 0 flag and a 9-space field.**

```c
/* Fig 9.15: fig09_15.c */
/* Printing with the 0( zero ) flag fills in leading zeros */
#include <stdio.h>

int main( void )
{
   printf( "%+09d\n", 452 );
   printf( "%09d\n", 452 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
+00000452
000000452
```

Fig. 9.15 | Using the 0 (zero) flag.

9.10 Printing Literals and Escape Sequences

- Most literal characters to be printed in a **printf** statement can simply be included in the format control string.
- However, there are several "problem" characters, such as the quotation mark (') that delimits the format control string itself.
- Various control characters, such as newline and tab, must be represented by escape sequences.
- An escape sequence is represented by a backslash (\), followed by a particular escape character.
- Figure 9.16 lists the escape sequences and the actions they cause.

Escape sequence	Description
`\'` (single quote)	Output the single quote (') character.
`\"` (double quote)	Output the double quote (") character.
`\?` (question mark)	Output the question mark (?) character.
`\\` (backslash)	Output the backslash (\) character.
`\a` (alert or bell)	Cause an audible (bell) or visual alert.
`\b` (backspace)	Move the cursor back one position on the current line.
`\f` (new page or form feed)	Move the cursor to the start of the next logical page.
`\n` (newline)	Move the cursor to the beginning of the next line.
`\r` (carriage return)	Move the cursor to the beginning of the current line.
`\t` (horizontal tab)	Move the cursor to the next horizontal tab position.
`\v` (vertical tab)	Move the cursor to the next vertical tab position.

Fig. 9.16 | Escape sequences.

printf( ) Examples

```
int anInt = 5678;
double aDouble = 4.123;

/* what is the output from each printf( )
   statement? */
printf ("%d is a large number\n", anInt);
printf ("%8d is a large number\n", anInt);
printf ("%-8d is a large number\n", anInt);
printf ("%10.2f is a double\n", aDouble);
printf( "The sum of %d and %8.4f is %12.2f\n",
    anInt, aDouble, anInt + aDouble);
```

Formatted Output Example

- **Use field widths to align output in columns**

```
int i;
for (i = 1 ; i < 5; i++)
  printf("%2d %10.6f %20.15f\n",
             i,sqrt(i),sqrt(i));
```

```
12 1234567890 12345678901234567890
 1   1.000000    1.000000000000000
 2   1.414214    1.414213562373095
 3   1.732051    1.732050807568877
 4   2.000000    2.000000000000000
```

Keyboard Input

In C, keyboard input is accomplished using the `scanf( )` function. `scanf` reads user input from `stdin`.

Calling `scanf( )` is similar to calling `printf( )`

```
scanf( format, arg1, arg2, ... )
```

The format string has a similar structure to the format string in `printf( )`. The arguments are the *addresses* of the variables into which the input is store.

scanf( ) format string

The `scanf( )` format string usually contains conversion specifications that tell `scanf( )` how to interpret the next "input field". An input field is a string of non-whitespace characters.

The format string usually contains

- Blanks or tabs which are ignored
- Ordinary characters which are expected to match the next (non-whitespace) character input by the user
- Conversion specifications usually consisting
  - % character indicating the beginning of the conversion
  - An optional h, l (ell) or L
  - A conversion character which indicates how the input field is to be interpreted.

Conversion specifier	Description
Integers	
`d`	Read an optionally signed decimal integer. The corresponding argument is a pointer to an `int`.
`i`	Read an optionally signed decimal, octal or hexadecimal integer. The corresponding argument is a pointer to an `int`.
`o`	Read an octal integer. The corresponding argument is a pointer to an unsigned `int`.
`u`	Read an unsigned decimal integer. The corresponding argument is a pointer to an unsigned `int`.
`x` or `X`	Read a hexadecimal integer. The corresponding argument is a pointer to an unsigned `int`.
`h` or `l`	Place before any of the integer conversion specifiers to indicate that a `short` or `long` integer is to be input.

Fig. 9.17 | Conversion specifiers for `scanf`. (Part 1 of 3.)

Conversion specifier	Description
Floating-point numbers	
`e`, `E`, `f`, `g` or `G`	Read a floating-point value. The corresponding argument is a pointer to a floating-point variable.
`l` or `L`	Place before any of the floating-point conversion specifiers to indicate that a `double` or `long double` value is to be input. The corresponding argument is a pointer to a `double` or `long double` variable.
Characters and strings	
`c`	Read a character. The corresponding argument is a pointer to a `char`; no null `('\0')` is added.
`s`	Read a string. The corresponding argument is a pointer to an array of type `char` that is large enough to hold the string and a terminating null `('\0')` character—which is automatically added.
Scan set	
[*scan characters*]	Scan a string for a set of characters that are stored in an array.

Fig. 9.17 | Conversion specifiers for `scanf`. (Part 2 of 3.)

Conversion specifier	Description
Miscellaneous	
`p`	Read an address of the same form produced when an address is output with `%p` in a `printf` statement.
`n`	Store the number of characters input so far in this call to `scanf`. The corresponding argument is a pointer to an `int`.
`%`	Skip a percent sign (`%`) in the input.

Fig. 9.17 | Conversion specifiers for `scanf`. (Part 3 of 3.)

9.11 Reading Formatted Input with scanf (Cont.)

- **Conversion specifier %i is capable of inputting decimal, octal and hexadecimal integers.**

```c
/* Fig 9.18: fig09_18.c */
/* Reading integers */
#include <stdio.h>

int main( void )
{
   int a;
   int b;
   int c;
   int d;
   int e;
   int f;
   int g;

   printf( "Enter seven integers: " );
   scanf( "%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );

   printf( "The input displayed as decimal integers is:\n" );
   printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Fig. 9.18 | Reading input with integer conversion specifiers. (Part 2 of 2.)

Fig. 9.18 | Reading input with integer conversion specifiers. (Part 1 of 2.)

9.11 Reading Formatted Input with scanf (Cont.)

- When inputting floating-point numbers, any of the floating-point conversion specifiers **e**, **E**, **f**, **g** or **G** can be used.
- Figure 9.19 reads three floating-point numbers, one with each of the three types of floating conversion specifiers, and displays all three numbers with conversion specifier **f**.
- The program output confirms the fact that floating-point values are imprecise—this is highlighted by the third value printed.

```c
/* Fig 9.19: fig09_19.c */
/* Reading floating-point numbers */
#include <stdio.h>

/* function main b
int main( void )
{
   double a;
   double b;
   double c;

   printf( "Enter three floating-point numbers: \n" );
   scanf( "%le%lf%lg", &a, &b, &c );

   printf( "Here are the numbers entered in plain\n" );
   printf( "floating-point notation:\n" );
   printf( "%f\n%f\n%f\n", a, b, c );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06
Here are the numbers entered in plain
floating-point notation:
1.279870
1279.870000
0.000003
```

Fig. 9.19 | Reading input with floating-point conversion specifiers. (Part 2 of 2.)

Fig. 9.19 | Reading input with floating-point conversion specifiers. (Part 1 of 2.)

9.11 Reading Formatted Input with scanf (Cont.)

- Characters and strings are input using the conversion specifiers `c` and `s`, respectively.
- Figure 9.20 prompts the user to enter a string.
- The program inputs the first character of the string with `%c` and stores it in the character variable `x`, then inputs the remainder of the string with `%s` and stores it in character array `y`.

```c
/* Fig 9.20: fig09_20.c */
/* Reading characters and strings */
#include <stdio.h>

int main( void )
{
   char x;
   char y[ 9 ];

   printf( "Enter a string: " );
   scanf( "%c%s", &x, y );

   printf( "The input was:\n" );
   printf( "the character \"%c\" ", x );
   printf( "and the string \"%s\"\n", y );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter a string: Sunday
The input was:
the character "S" and the string "unday"
```

Fig. 9.20 | Inputting characters and strings.

9.11 Reading Formatted Input with scanf (Cont.)

- A sequence of characters can be input using a **scan set**.
- A scan set is a set of characters enclosed in square brackets, [], and preceded by a percent sign in the format control string.
- A scan set scans the characters in the input stream, looking only for those characters that match characters contained in the scan set.
- Each time a character is matched, it's stored in the scan set's corresponding argument—a pointer to a character array.
- The scan set stops inputting characters when a character that is not contained in the scan set is encountered.

9.11 Reading Formatted Input with scanf (Cont.)

- If the first character in the input stream does not match a character in the scan set, only the null character is stored in the array.
- Notice that the first seven letters of the input are read.
- The eighth letter (h) is not in the scan set and therefore the scanning is terminated.

```c
/* Fig 9.21: fig09_21.c */
/* Using a scan set */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   char z[ 9 ]; /* define array z */

   printf( "Enter string: " );
   scanf( "%[aeiou]", z ); /* search for set of characters */

   printf( "The input was \"%s\"\n", z );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter string: ooeeooahah
The input was "ooeeooa"
```

Fig. 9.21 | Using a scan set.

9.11 Reading Formatted Input with scanf (Cont.)

- The scan set can also be used to scan for characters not contained in the scan set by using an **inverted scan set**.
- To create an inverted scan set, place a **caret (^)** in the square brackets before the scan characters.
- This causes characters not appearing in the scan set to be stored.
- When a character contained in the inverted scan set is encountered, input terminates.
- Figure 9.22 uses the inverted scan set **[^aeiou]** to search for consonants—more properly to search for "nonvowels."

```c
/* Fig 9.22: fig09_22.c */
/* Using an inverted scan set */
#include <stdio.h>

int main( void )
{
   char z[ 9 ];

   printf( "Enter a string: " );
   scanf( "%[^aeiou]", z ); /* inverted scan set */

   printf( "The input was \"%s\"\n", z );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter a string: String
The input was "Str"
```

Fig. 9.22 | Using an inverted scan set.

9.11 Reading Formatted Input with scanf (Cont.)

- A field width can be used in a **scanf** conversion specifier to read a specific number of characters from the input stream.
- Figure 9.23 inputs a series of consecutive digits as a two-digit integer and an integer consisting of the remaining digits in the input stream.

```c
/* Fig 9.23: fig09_23.c */
/* inputting data with a field width */
#include <stdio.h>

int main( void )
{
   int x;
   int y;

   printf( "Enter a six digit integer: " );
   scanf( "%2d%d", &x, &y );

   printf( "The integers input were %d and %d\n", x, y );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Fig. 9.23 | Inputting data with a field width.

9.11 Reading Formatted Input with scanf (Cont.)

- **Often it's necessary to skip certain characters in the input stream.**
- **For example, a date could be entered as**
  - 11-10-1999
- **Each number in the date needs to be stored, but the dashes that separate the numbers can be discarded.**
- **To eliminate unnecessary characters, include them in the format control string of scanf (white-space characters—such as space, newline and tab—skip all leading white-space).**

9.11 Reading Formatted Input with scanf (Cont.)

- For example, to skip the dashes in the input, use the statement
  - scanf('%d-%d-%d', &month, &day, &year);
- Although, this **scanf** does eliminate the dashes in the preceding input, it's possible that the date could be entered as
  - 10/11/1999
- In this case, the preceding **scanf** would not eliminate the unnecessary characters.
- For this reason, **scanf** provides the **assignment suppression character *.**

9.11 Reading Formatted Input with scanf (Cont.)

- The assignment suppression character enables **scanf** to read any type of data from the input and discard it without assigning it to a variable.
- The argument lists for each **scanf** call do not contain variables for the conversion specifiers that use the assignment suppression character.
- The corresponding characters are simply discarded.

```c
/* Fig 9.24: fig09_24.c */
/* Reading and discarding characters from the input stream */
#include <stdio.h>

int main( void )
{
   int month1;
   int day1;
   int year1;
   int month2;
   int day2;
   int year2;

   printf( "Enter a date in the form mm-dd-yyyy: " );
   scanf( "%d%*c%d%*c%d", &month1, &day1, &year1 );

   printf( "month = %d  day = %d  year = %d\n\n", month1, day1, year1 );

   printf( "Enter a date in the form mm/dd/yyyy: " );
   scanf( "%d%*c%d%*c%d", &month2, &day2, &year2 );

   printf( "month = %d  day = %d  year = %d\n", month2, day2, year2 );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter a date in the form mm-dd-yyyy: 11-18-2003
month = 11  day = 18  year = 2003

Enter a date in the form mm/dd/yyyy: 11/18/2003
month = 11  day = 18  year = 2003
```

Fig. 9.24 | Reading and discarding characters from the input stream. (Part 2 of 2.)

Fig. 9.24 | Reading and discarding characters from the input stream. (Part 1 of 2.)

scanf( ) examples

```
int age;
double gpa;

printf("Input your age: ");
scanf( "%d", &age );                /* note &
  */
printf(" input your gpa: ");
scanf ("%lf", &gpa );
```

Multiple dataitems

```
#include <stdio. h>
main( )
char item[20];
int partno;
float cost;
scanf(“   %s  %d  %f', item, &partno, &cost);
```

- If two or more data items are entered, they must be separated by whitespace characters.
- Notice the blank space that precedes **%s.**
- **Exception:**
- **scanf("%s%d%f", item, &partno, &cost); //valid due to c type conversions**

- **A null character (\O) will then automatically be added to the end of the string.**

```
#include <stdio.h>

main()

{
    char line[80];

    . . . . .

    scanf(" %[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);

    . . . . .

}
```

This means that only the characters specified in the brackets are allowed in the input string.

EXAMPLE

- It is useful to precede the characters within the square brackets by a ***Circumflex (i.e., ^)***.
- characters specified after ^ are not allowed in the input string.

```
#include <stdio.h>
main( )
{
char  line [ 8 0 ] ;
scanf (“  %[ ^\n ] “  , line ) ;
. . . . .
```

- **Note: the blank space preceding %[^\n] ,to ignore any unwanted characters that may have been entered previously.**
- **When the scanf function is executed, a string of undetermined length (but not more than 79 characters) will be entered from the standard input device and assigned to l i n e .**

MORE ABOUT THE scanf ()

- Can specify maximum fieldwidth of a data item .
- To do so, an unsigned integer indicating the field width is placed within the control string, between the percent sign (%) and the conversion character.
- The data item may contain fewer characters than the specified field width.
- However, the number of characters in the actual data item cannot exceed the specified field width.
- Any characters that extend beyond the specified field width will not be read.
- Such leftover characters may be incorrectly interpreted as the components of the next data item.

```
#include <stdio.h>

main()
{
    int a, b, c;

    . . . . .

    scanf("%3d %3d %3d", &a, &b, &c);

    . . . . .
}
```

Suppose the input data items are entered as **1 2 3**

Then the following assignments will result:

a = 1, b = 2, c = 3

If the data had been entered **as** 123 456 789
Then the assignments would be a = 123, b = 456, c = 789

suppose that the data had been entered **as** 123456789
Then the assignments would be a = 123, b = 456, c = 789

Finally, suppose that the data had been entered **as** 1234 5678 9
The resulting assignments would now be a = 123, b = 4, c = 567
The remaining two digits (8 and 9) would be ignored

```c
#include <stdio.h>

main()

{
    int i;
    float x;
    char c;

    . . . . .

    scanf("%3d %5f %c", &i, &x, &c);

    . . . . .
}
```

If the data items are entered **as** 10 256.875 **T**
10 will be assigned to i
256.8 will be assigned to x and
the character 7 will be assigned to c.

The remaining two input characters (5 and T) will be ignored.

```
#include <stdio.h>

main()

{
     short ix,iy;
     long lx,ly;
     double dx,dy;

     . . . . .

     scanf("%hd %ld %lf", &ix, &lx, &dx);

     . . . . .

     scanf("%3ho %7lx %15le", &iy, &ly, &dy);

     . . . . .
}
```

single-letter ***prefix***,

an l (lowercaseL) - either a signed or unsigned long integer argument, or a double-precision argument.
h - a signed or unsigned short integer.
L - a long double.

```
#include <stdio.h>

main()

{
    char item[20];
    int partno;
    float cost;

    . . . . .

    scanf(" %s %*d %f", item, &partno, &cost);

    . . . . .
}
```

If the corresponding data items are
fastener 12345 0.05

then **fastener will be assigned to item and 0.05 will be assigned to cost.**
12345 will not be assigned to partno because of the asterisk

```
#include <stdio.h>
main ( )
{
char c l , c2, c3;
scanf(" %c%c%c", &cl, &c2, &c3);
```

- If the data items are entered as a b c
- then the following assignments would result:

  cl = a, c2 = <blankspace>, c3 = b

- scanf(" %c%1s%1s", &cl, &c2, &c3)
- then the same input data would result in the following assignments: cl = a, c2 = b, c3 = c

- We could have written the scanf function as
- scanf (' %c %c %c", &cl, &c2, &c3);

  or we could have used the original scanf function but written the input data as consecutive characters without blanks; i.e., abc.

gets(),puts()

```
#include <stdio.h>

main()          /* read and write a line of text */

{
    char line[80];

    gets(line);
    puts(line);
}
```

- **Write an interactive C program that reads in a student's name and three exam scores, and then calculates an average score.**

```c
#include <stdio.h>

main()     /* sample interactive program */

{
    char name[20];
    float score1, score2, score3, avg;

    printf("Please enter your name: ");          /* enter name */
    scanf(" %[^\n]", name);

    printf("Please enter the first score:  ");  /* enter 1st score */
    scanf("%f", &score1);

    printf("Please enter the second score: ");  /* enter 2nd score */
    scanf("%f", &score2);

    printf("Please enter the third score:  ");  /* enter 3rd score */
    scanf("%f", &score3);

    avg = (score1+score2+score3)/3;             /* calculate avg */

    printf("\n\nName: %-s\n\n", name);          /* write output */
    printf("Score 1: %-5.1f\n", score1);
    printf("Score 2: %-5.1f\n", score2);
    printf("Score 3: %-5.1f\n\n", score3);
    printf("Average: %-5.1f\n\n", avg);
}
```

```
Please enter your name: Robert Smith
Please enter the first score:  88
Please enter the second score: 62.5
Please enter the third score:  90

Name: Robert Smith

Score 1:        88.0
Score 2:        62.5
Score 3:        90.0

Average:        80.2
```