C FILE Type

Accessing a stream requires a pointer variable of type `FILE`.

C provides three standard streams, which require no special preparation other than the necessary `include` directive:

`stdin`	standard input	keyboard
`stdout`	standard output	console window
`stderr`	standard error	console window

Alternatively, you can declare `FILE` pointers and manage their connections to disk files.

header file: <stdio.h>

Writing Output with `printf()`

The Standard Library provides the `printf()` function which can be used to write formatted output to the standard output stream, `stdout`.

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

The first parameter is a string literal that specifies the formatting to be used.

The remaining parameters, if any, specify the values to be written.

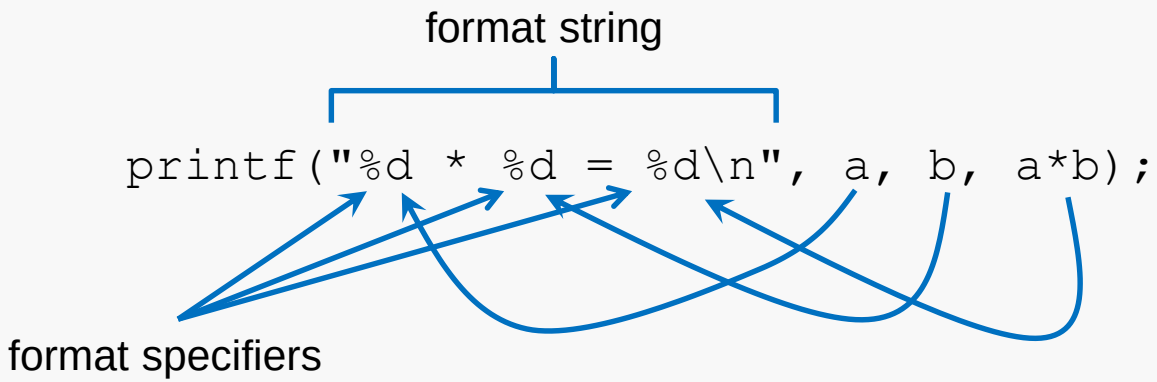
The return value is the number of characters that were written, or a negative value if an error was encountered.

Printing Integer Values

The specification of formatting is mildly complex:

```
int a = 42, b = -17;

printf("%d * %d = %d\n", a, b, a*b);
```

```
42 * -17 = -714
```



`%d` means print the corresponding value as a base-10 integer

Printing Integer Values

The specification of formatting is mildly complex:

<stdlib.h>

```
int a = 42, b = -17;

printf("The absolute value of %d is %d.\n", b, abs(b) );
```

```
The absolute value of -17 is 17.
```

```
printf("%d\n+ %d\n= %d\n", a, b, a + b);
```

```
42
+ -17
= 25
```

The alignment of the last one isn't ideal; fortunately, we can specify field widths...

Aligning Output

`%Wd` means print the corresponding value as a base-10 integer, right-aligned in W columns:

```
printf("%7d\n+ %5d\n= %5d\n", a, b, a + b);
```

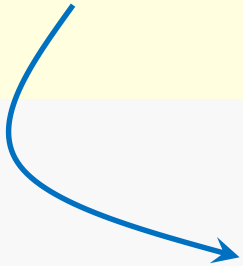
If the value requires more than the specified number of columns, then the width specifier is ignored and the entire value is printed.

Printing Floating-point Values

There are different format specifiers for different data types:

```
double r  = 2.0;
double pi = 3.141592;
```

```
printf("The area of a circle of radius %.2f is about %.4f\n",
        r, pi * r * r);
```



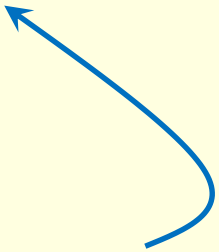
```
The area of a circle of radius 2.00 is about 12.5664
```

`%f` means print the corresponding value as a base-10 decimal value

`%W.Pf` means print the corresponding value right-aligned in W columns and show P digits after the decimal point

Printing a Table

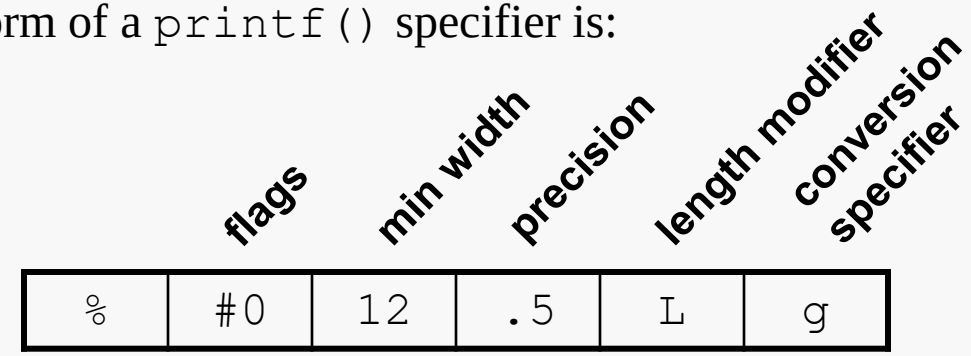
```
#include <stdio.h>
#include <math.h>

int main() {


   printf("  x   sqrt(x)\n");
   printf("-------------\n");

   for (int i = 2; i < 20; i++) {

       printf("%3d%10.4f\n", i, sqrt(i));
   }

   return 0;
}
```



```
 x    sqrt(x)
-------------
  2    1.4142
  3    1.7321
  4    2.0000
  5    2.2361
  6    2.4495
  7    2.6458
  8    2.8284
  9    3.0000
 10    3.1623
 11    3.3166
 12    3.4641
 13    3.6056
 14    3.7417
 15    3.8730
 16    4.0000
 17    4.1231
 18    4.2426
 19    4.3589
```

More on `printf()` Format Specifiers

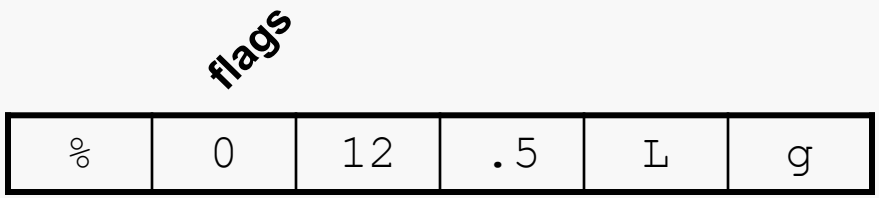
The general form of a `printf()` specifier is:



Format specifiers begin with the percent sign `'%'`.

All fields except the **conversion specifier** are optional.

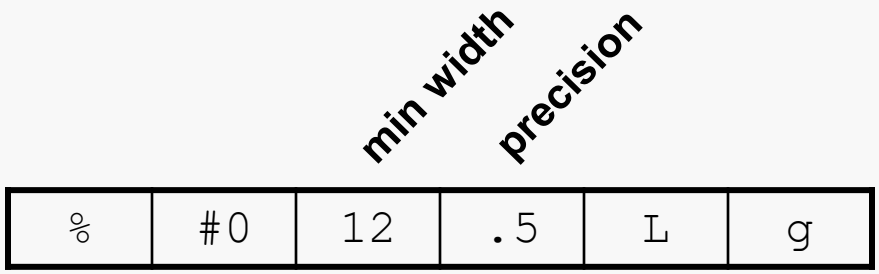
More on `printf()` Format Specifiers



flags: optional, more than one allowed

–	left-justify within field; right-justification is default for numbers
+	always show leading sign (for numeric output)
space	precede non-negative numbers with a space, if first character is not a sign
#	see reference, rarely used
0	pad with zeros to fill field; cannot use a different padding character (alas)

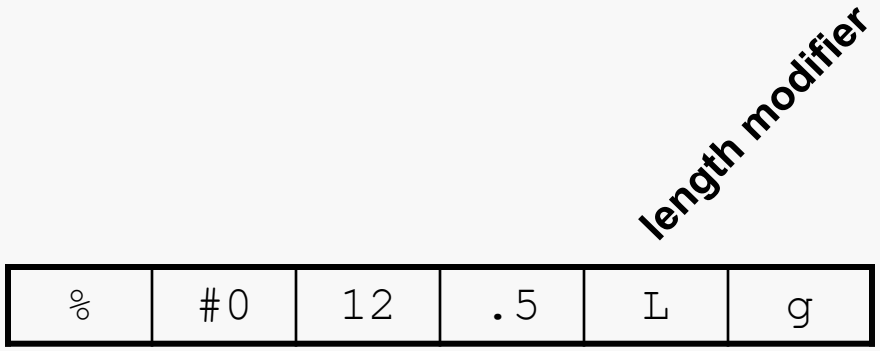
More on `printf()` Format Specifiers



min width: optional, pad with spaces if field not filled, ignored if insufficient

precision: optional, number of digits for integer values, number of digits after decimal point for `float/double`

More on `printf()` Format Specifiers

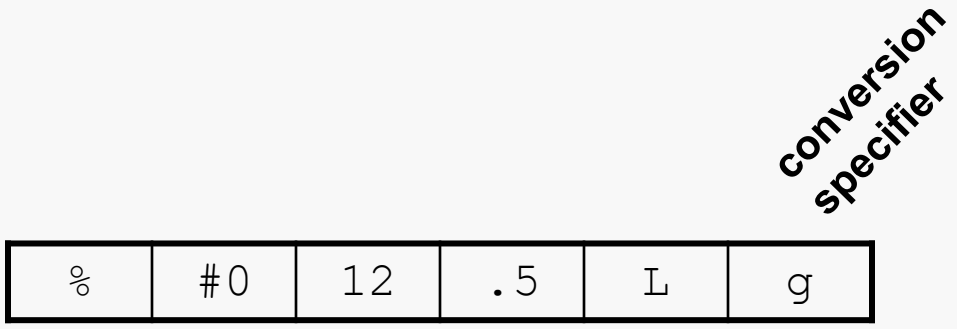


length modifier: optional, indicates value to be printed has a type that's longer or shorter than is normal for a particular conversion specifier (see following)

`d`	normally used for `int` values
`u`	normally used for `unsigned int` values
`hd`	normally used for `short int` values
`ld`	normally used for `long int` values

see reference for more…

More on `printf()` Format Specifiers



conversion specifier: mandatory

`d, i`	converts `int` to text decimal form (base-10)
`u`	converts `unsigned int` to text decimal form (base-10)
`f, F`	converts `float` to text decimal form; default precision 6
`c`	converts byte to `char` or `unsigned char`
`x, X`	converts `unsigned int` to text hexadecimal (base-16)
`s`	convert `char*` to text string (assumes C-string)
`p`	convert pointer value (address) to text hexadecimal

see reference for more…

Example

```
#include <stdio.h>

int GCD(int N, int M) {

   printf("N = %5d and M = %5d\n", N, M);
   if ( M == 0 ) return N;
   return GCD(M, N % M);
}

int main(int argc, char** argv) {

   if ( argc != 3 ) {
      printf("gcd X Y\n");
      return 1;
   }

   int X = atoi(argv[1]);
   int Y = atoi(argv[2]);

   printf("GCD is %d\n", GCD(X, Y));
   return 0;
}
```

```
centos> gcd 24060 87123
N = 24060 and M = 87123
N = 87123 and M = 24060
N = 24060 and M = 14943
N = 14943 and M =  9117
N =  9117 and M =  5826
N =  5826 and M =  3291
N =  3291 and M =  2535
N =  2535 and M =   756
N =   756 and M =   267
N =   267 and M =   222
N =   222 and M =    45
N =    45 and M =    42
N =    42 and M =     3
N =     3 and M =     0
GCD is 3
```

Example

```
#include <stdio.h>
#include <math.h>    // use -lm to compile

   . . .
   double PI = 4.0 * atan(1.0);

   printf("PI = %12.10f\n", PI);

   printf("PI = %12.9f\n", PI);

   printf("PI = %12.8f\n", PI);

   printf("PI = %.4f\n", PI);
   . . .
```

```
PI = 3.1415926536
PI =  3.141592654
PI =   3.14159265
PI = 3.1416
```

Format Specifiers for `<stdint.h>`

The basic integer format codes will work with `int32_t` and `uint32_t` (with compiler warnings), but are not reliable with `int64_t` and `uint64_t`.

The header file `<inttypes.h>` provides specialized format codes for the new integer types.

Here's a very brief description; see your C reference for details.

PRId*N*	for signed integer types, *N* = 8, 16, 32, 64
PRIu*N*	for unsigned integer types

For example:

```
uint64_t K = 123456789012345;
printf("%15"PRIu64"\n", K);  // note use of quotes!!
```

Example

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main() {

   int32_t N = INT32_MAX;
   uint8_t nDivisions = 0;

   while ( (N = N / 2) > 0 ) {

      ++nDivisions;

      printf("%2"PRIu8"%12"PRId32"\n",
                nDivisions, N);
   }

   return 0;
}
```

```
 1  1073741823
 2   536870911
 3   268435455
 4   134217727
 5    67108863
 6    33554431
 7    16777215
...
23         255
24         127
25          63
26          31
27          15
28           7
29           3
30           1
```

Input with `scanf()`

The Standard Library provides the `scanf()` function which can be used to read formatted input from the standard input stream, `stdin`.

```
int scanf(const char * restrict format, . . .);
```

The first parameter is a string literal that specifies the formatting expected in the input stream.

The remaining parameters, if any, specify the variables that will receive the values that are read.

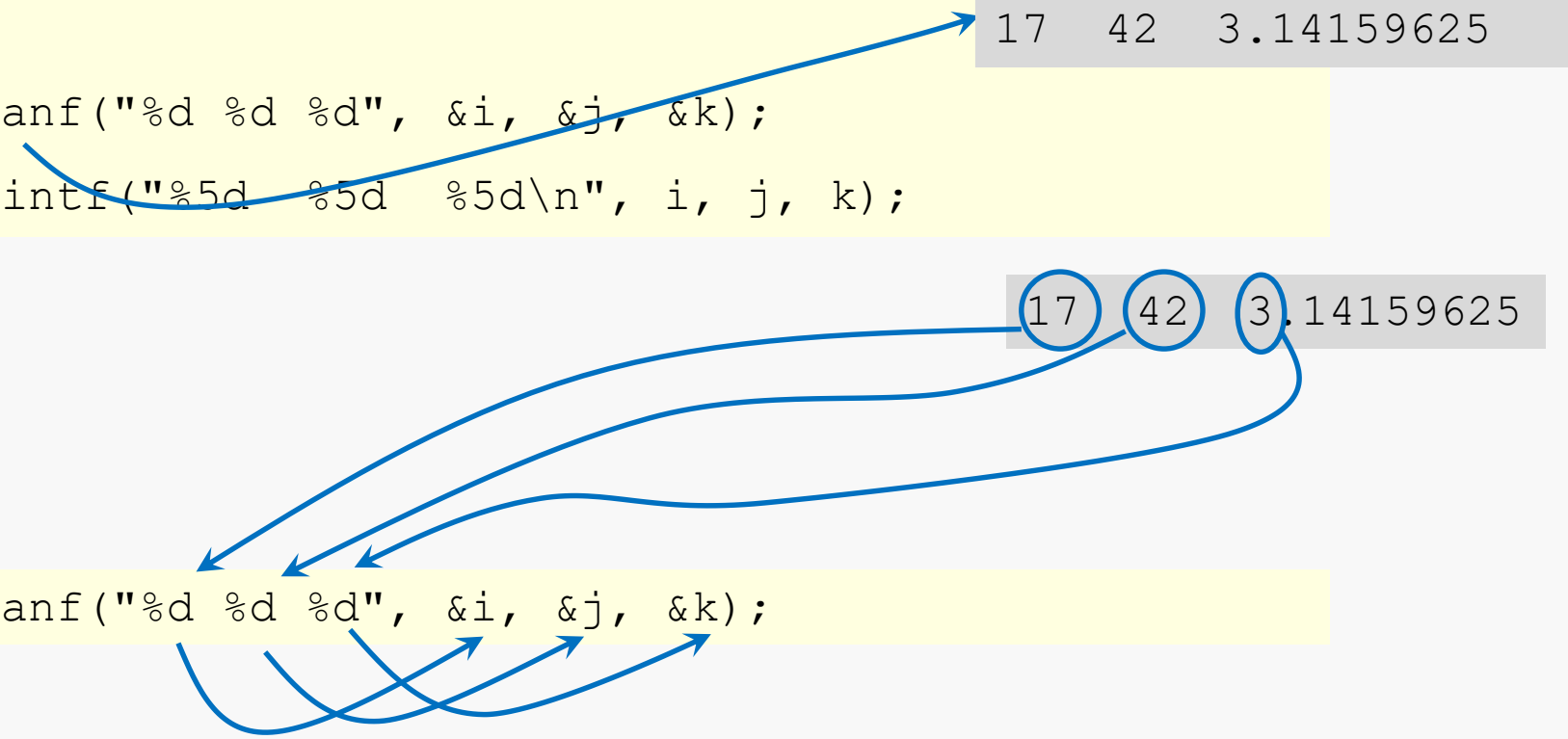
The return value is the number of values that were read, or the value of `EOF` if an input failure occurs.

Simple Integer Input

Suppose we have an input stream of the following form:

```
17   42   3.14159625
```

```
int i = 1, j = 1;
double k = 1.0;

scanf("%d %d %d", &i, &j, &k);
printf("%5d  %5d  %5d\n", i, j, k);
```



Floating-point Input

Suppose we have an input stream of the following form:

```
17, 42, 3.14159625
```

```
int i = 1, j = 1;
double x = 1.5;
scanf("%d, %d, %f", &i, &j, &x);
printf("%5d  %5d  %7.4f\n", i, j, x);
```

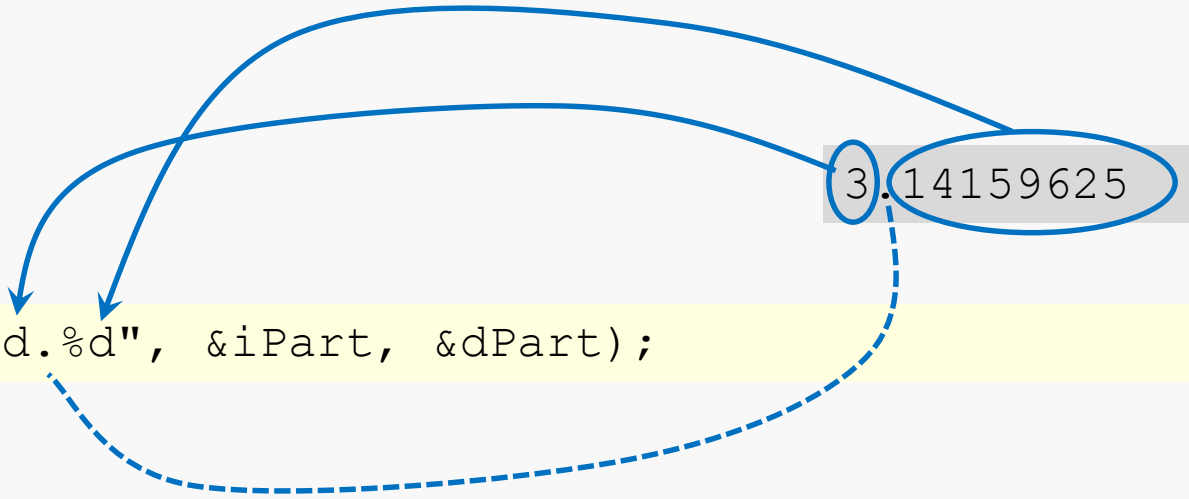
```
   17   42 3.1416
```

Formatted Input

Suppose we have an input stream of the following form: `3.14159625`

```
int iPart = 1, dPart = 1;

scanf("%d.%d", &iPart, &dPart);

printf("%d\n%d\n", iPart, dPart);
```

```
3
14159625
```

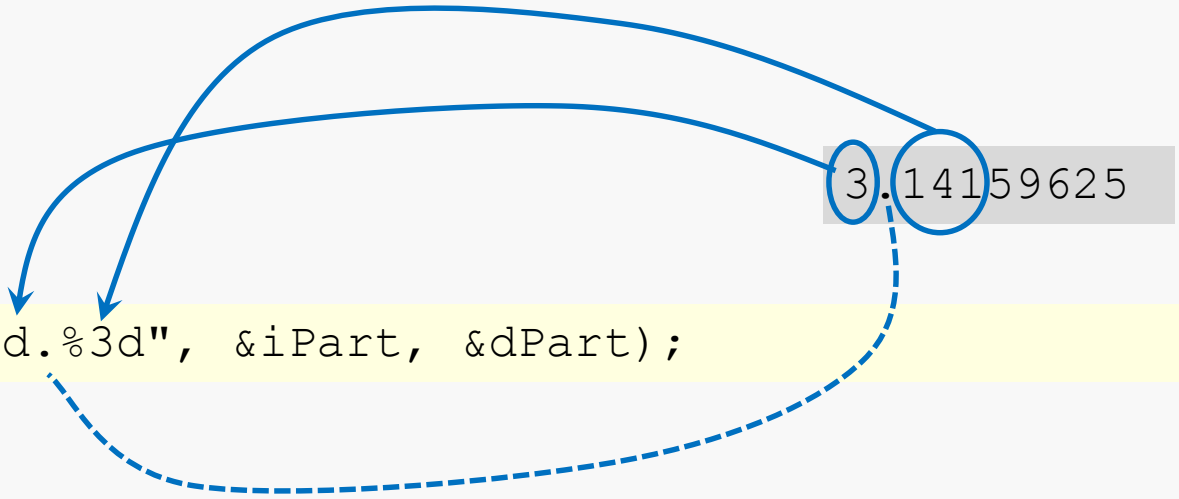


Input with Width Specifier

Suppose we have an input stream of the following form: `3.14159625`

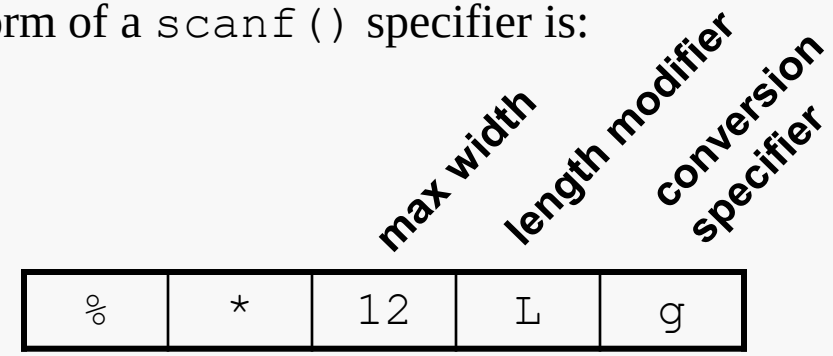
```
int iPart = 1, dPart = 1;

scanf("%d.%3d", &iPart, &dPart);

printf("%d  %d\n", iPart, dPart);
```

```
3  141
```



More on `scanf()` Format Specifiers

The general form of a `scanf()` specifier is:

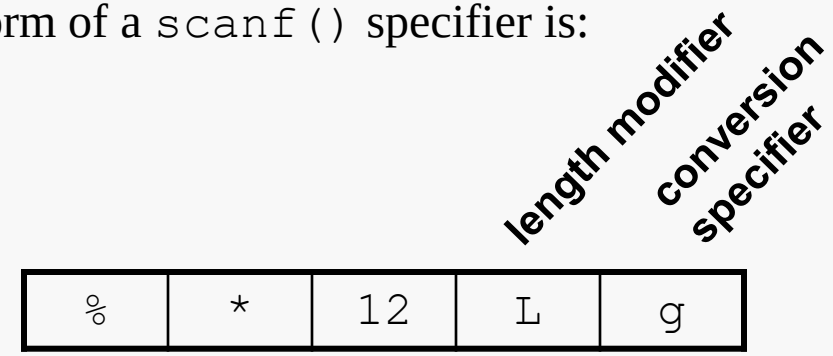


*****: optional, read but do not assign value to an object

max width: optional, leading whitespace doesn't count

More on `scanf()` Format Specifiers

The general form of a `scanf()` specifier is:



length modifier: optional, indicates object that will receive value has a type that's longer or shorter than is normal for a particular conversion specifier (see following)

see reference for more…

conversion specifier:mandatory

see reference for more…

Example

```
const int secondsPerMinute = 60;
const int minutesPerHour   = 60;

void sumTimes() {

   int sumOfSeconds = 0;
   int minutes;
   int seconds;

   while ( scanf("%d:%d\n", &minutes, &seconds) == 2 ) {
      sumOfSeconds += seconds + secondsPerMinute*minutes;
   }

   int totalSeconds = sumOfSeconds % secondsPerMinute;
   sumOfSeconds /= secondsPerMinute;
   int totalMinutes = sumOfSeconds % minutesPerHour;
   int totalHours = sumOfSeconds / minutesPerHour;

   printf("Total time: %d:%02d:%02d\n", totalHours,
                                        totalMinutes,
                                        totalSeconds);
}
```

```
14:32
 7:42
29:17
10:00
```

```
Total time: 1:01:31
```

Opening Files

File I/O is almost identical to I/O with `stdin` and `stdout`.

You must make a call in order to associate a `FILE` pointer with a particular file:

```
FILE *fopen(const char* restrict filename,
            const char* restrict mode);
```

`filename`	path/name of file to be opened			
`mode`	`"r"`	`"w"`	`"a"`	
	`"rb"`	`"wb"`	`"ab"`	
	`"r+"`	`"w+"`	`"a+"`	see reference for more

Return value is valid `FILE` pointer if successful and `NULL` otherwise.

File I/O

File I/O is accomplished using variations of `printf()` and `scanf()`:

```
int fprintf(FILE * restrict stream,
            const char * restrict format, . . .);

int fscanf(FILE * restrict stream,
           const char *  restrict format, . . .);
```

These are used in the same way as their counterparts, aside from taking a `FILE*`.

Closing Files

When done with a file, you should close the file:

```
int fclose(FILE *stream);
```

Any unwritten buffered data will be delivered to the host environment.

Any unread buffered data will be discarded.

Returns `0` on success and `EOF` otherwise.

Example: Caesar Cipher

The Caesar Cipher is an ancient, weak scheme for encrypting text.

The basic idea is quite simple: create the *ciphertext* by replacing each letter in the unencrypted text (*plaintext*) with a letter that is a fixed position from it in the alphabet, wrapping around the ends of the alphabet as necessary.

For example, using a shift of 3 positions, we'd use the following substitution table:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
d e f g h i j k l m n o p q r s t u v w x y z a b c
```

So,

```
computer organization
```

would be encrypted as

```
frpsxwhu rujdqlcdwlrq
```

Example: Analysis of Problem

Let's consider implementing a C program that will apply the Caesar Cipher to a given text sample.

Let's assume the user will want to specify the shift amount and the text to be encrypted.

Let's also assume the user will want the case of the original text to be preserved, but that only letters should be changed (in accord with what we know about the original approach).

Let's mandate the interface: `caesar <shift amount> <plaintext file>`

Example: Procedural Decomposition

I identify the following tasks that need to be carried out:

- validate the command-line parameters
- process the input file
- exit

- verify number of parameters supplied
- verify shift amount is sensible
- verify input file exists

- open input file
- read next input char until no more exist
  + if char is a letter compute its shift target
  + write shift target (where?)
- close input file

Example: Front End

```
int main(int argc, char** argv) {

   int ckStatus;
   if ( ( ckStatus = checkParams(argc, argv) ) != 0 ) {
      return ckStatus;
   }

   int shiftAmt = setShiftAmt(argv[1]);

   printf("Shifting alphabetic input text by %d positions.\n",
          shiftAmt);

   int charsShifted = processFile(shiftAmt, argv[2]);

   printf("Shifted %d alphabetic characters.\n", charsShifted);

   return 0;
}
```

```
int setShiftAmt(char* src) {

   char *p;
   int shiftAmt = strtol(src, &p, 10);
   return shiftAmt;
}
```

Example: Comments

```
. . .
int ckStatus;
if ( ( ckStatus = checkParams(argc, argv) ) != 0 ) {
   return ckStatus;
}
. . .
```

The test in the **if** statement is classic C idiom:

- assign the result of a function call to a local variable
- test the value of the assignment to determine whether to enter the **if**
- the value of an assignment is the value that's copied to the left side

Example: Validating Parameters

```
int checkParams(int nparams, char** params) {

   if ( nparams != 3 ) {
      printf("Invoke as: caesar <shift distance> <file name>\n");
      return WRONG_NUMBER_OF_PARAMS;
   }

   if ( !checkShiftAmt(params[1]) ) {
      return INVALID_SHIFT_SPECIFIED;
   }

   FILE* fp;
   if ( (fp = fopen(params[2], "r") ) == NULL ) {
      printf("The file %s could not be found.\n", params[2]);
      return FILE_NOT_FOUND;
   }
   else {
      fclose(fp);
   }
   return 0;
}
```

```
#define WRONG_NUMBER_OF_PARAMS    1
#define INVALID_SHIFT_SPECIFIED   2
#define FILE_NOT_FOUND            3
```

Example: Processing the File

```
int processFile(int shiftAmt, char* fileName) {

   int nChars = 0;
   FILE *In = fopen(fileName, "r");

   char nextIn, nextOut;
   while ( fscanf(In, "%c", &nextIn) == 1 ) {

      if ( isalpha(nextIn) ) {
         ++nChars;
         nextOut = applyShift(nextIn, shiftAmt);
      }
      else
         nextOut = nextIn;

      printf("%c", nextOut);
   }

   fclose(In);
   return nChars;
}
```

```
char applyShift(char Original,
                int shiftAmt) {

   char Modified = Original;
   . . .
   return Modified;
}
```

Executing the Caesar Cipher Program

```
centos> caesar 3 AMansAManForAThat.txt
Shifting alphabetic input text by 3 positions.
D Pdq'v d Pdq iru D' Wkdw

Lv wkhuh iru krqhvw Sryhuwu
   Wkdw klqjv klv khdg, dq' d' wkdw;
Wkh frzdug vodyh zh sdvv klp eu,
   Zh gduh eh srru iru d' wkdw!
```

```
A Man's a Man for A' That

Is there for honest Poverty
   That hings his head, an' a' that;
The coward slave we pass him by,
   We dare be poor for a' that!
. . .
```

```
D Pdq'v d Pdq iru D' Wkdw

Lv wkhuh iru krqhvw Sryhuwu
   Wkdw klqjv klv khdg, dq' d' wkdw;
Wkh frzdug vodyh zh sdvv klp eu,
   Zh gduh eh srru iru d' wkdw!
. . .
```