Chapter 15
File Input & Output

15.1 Chapter Overview
Input/Output operations in C are carried out through function calls, which are included in the library header file <stdio.h>.

15.2 Files
Files contain a collection of related data, treated as a unit. Files contain text or binary types of data, and are stored on storage devices (e.g. hard drives). With text files, data is stored as human-readable characters, and each line of data ends with a '\n' character.

Standard Files
Three special `FILE *`'s defined in `<stdio.h>` are automatically opened by the system when a program is run (they must not be declared nor opened in your program):

- ***stdin*** - standard input - normally associated with the keyboard (but may be redirected with <)
- ***stdout*** - standard output - normally associated with the terminal output, i.e. the display screen (but may be redirected with >)
- ***stderr*** - standard error - normally associated with the terminal output; this is where most of the error messages produced by the system are written. This is very useful when the program's output is being redirected to a file - the normal output will be written to the file but any system error messages will still appear in your window (the error message are separate from the normal output so they are not also written to the output file).

User Files
Users may also create and/or use other files. These files must be explicitly opened in the program before any I/O operations can occur.

The `fopen()` function is used to open a file for processing. This function makes the connection between the external file and the program. The signature looks like the following: `fopen("filename", "mode")` where `filename` refers to the name of the file, e.g. `input.txt` or `scores.dat`, and `mode` refers to the type of operation that will be performed with the file:

MODE	MEANING
r	Open file for reading • if file exists, the marker is positioned at the beginning • if the file does not exist, an error is returned
w	Open text file for writing • if file exists, it is emptied • if file does not exist, it is created
a	Open text file for append • if file exists, the marker is positioned at the end • if the file does not exist, it is created

There are a few other modes explained in Chapter 16, page 364 ("r+", "w+", and "a+").

15.3 Examples

```
/* open a file for reading */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

      FILE *inFile;
      int number;

      inFile = fopen("input.dat", "r");
      if (inFile == NULL) {
            fprintf(stderr, "File open error.  Exiting program\n");
            exit(1);    // or  exit(EXIT_FAILURE);
      }

      fscanf(inFile, "%d", &number);
      printf("The number from the file is:  %d\n", number);

      fclose(inFile);
      return 0;
}
```

```
/* open a file for writing */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

      FILE *outFile;
      int number;

      outFile = fopen("output.dat", "w");
      if (outFile == NULL) {
            fprintf(stderr, "Output failure.  Exiting program\n");
            exit(2);
      }

      fprintf(stdout, "Enter a number");
      fscanf(stdin, "%d", &number);
      fprintf(stdout, "The number entered was: %d\n", number);

      fprintf(outFile, "%d", number);

      fclose(outFile);
      return 0;
}
```

15.4 The `exit` Function

The standard header file `<stdlib.h>` defines exit statuses `EXIT_FAILURE` and `EXIT_SUCCESS` as integer values that you can use to indicate if the program has failed or has succeeded. The `exit(n)` function takes an integer `n` as an argument and terminates the program. Any open files are automatically closed by the system in the event that the program exits. In the examples above, you can see that a good use for this is when checking to see if there was a failure with opening a file; if so, you can exit or terminate the program using any non-zero value for the exit (to indicate failure) or using the predefined `EXIT_FAILURE`. If you exit with a zero value, that is the same as `EXIT_SUCCESS`, which indicates to the system a successful termination (same as `return 0` from main).

15.5 File and I/O Functions

`fopen("filename", "mode")`

- opens the specified file with the type of operation to be performed with the file given by `mode`
- returns a pointer to the file, or null if unsuccessful

`fclose(file_pointer)`

- closes the file when no longer needed
- prevents the associated file from being accessed again
- guarantees that all the data stored in the stream buffer is written to the file
- releases the file structure so that it can be used with another file
- frees system resources, such as buffer space

`fprintf()` and `fscanf()`

- can be used to write to/read from stdout or stderr/stdin, or to/from a specified file (specified by the file pointer)

getting and putting characters

`int getchar()` - reads the next character from the standard input stream and returns its value; `eof` is returned upon error or end of file; it returns an `int`, being the ASCII code of the character, but you can assign the result to a `char` variable. Example:

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

`int getc(FILE *fpIn)` - reads the next character (byte) from the file; `eof` is returned upon error or end of file; can also assign the result to a `char` variable:

```
char c;
FILE *inFile = fopen("input", "r");
c = getc(inFile);
```

`int fgetc(FILE *fpIn)` - reads the next character (byte) from the file; `eof` is returned upon error or end of file; (same as `getc()` function)

`int putchar(int outChar)` - writes the character to standard output; if successful, returns the character written, or `eof` on failure

```
putchar('H');  // will print the letter H to the screen
```

`int putc(int oneChar, FILE *fpOut)` - writes value of the character to the specified file; if successful, character is returned, or `eof` on failure

```
FILE *outFile = fopen("output", "w");
putc('H', outFile);
```

`int fputc(int oneChar, FILE *fpOut)` - writes value of the character to the file; if successful, character is returned, or `eof` on failure (same as `putc` function)

NOTE: `getc()/fgetc()` and `putc()/fputc()` are inefficient for reading or writing a large volume of data and should never be used for large volumes of data, e.g. photographic image files.

getting and putting strings

`char *gets(void *buffer)` - reads a line (terminated by a `\n`) from standard input; `\n` is not read into the buffer, but `\0` is added to the end of the line; if successful, returns the string and puts it into the array pointed to by `buffer`; otherwise a null pointer if returned; this is a deprecated function and gcc will warn you against using `gets()`

```
char buffer[80];
gets(buffer);
```

`char *fgets(void *buffer, int buffSize, FILE *fp)` - reads a line (terminated by a `\n`) from the specified file up until the `\n` character or up to the `buffSize` (i.e. `buffSize-1`); appends `\0` at the end of the string (after the `\n`); if successful, returns the string, otherwise a null pointer if returned

```
char buffer[80];
FILE *inFile = fopen("input", "r");
fgets(buffer, 80, inFile);
```

```
/* using fgets() and sscanf() */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

      char id[3] = {0, 0, 0};
      long values[3];
      int count = 0;
      char buff[256];

      FILE *inFile = fopen ("image.ppm", "r");
      if (inFile == NULL) {
            fprintf(stderr, "File open error.  Exiting program\n");
            exit(1);
      }

      fgets(buff, 256, inFile);
      id[0] = buff[0];
      id[1] = buff[1];

      count = sscanf(&buf[2], "%d %d %d", &values[0], &values[1], &values[2]);

      fprintf(stderr, "obtained %d values\n", count);
      fprintf(stderr, "%li %li %li\n", values[0], values[1], values[2]);

      fclose (inFile);
      return 0;
}
```

`int puts(void *buffer)` - takes a null-terminated string from memory (the `buffer`), or a string literal, and writes it to standard output until a null character is encountered; a `\n` is automatically written as the last character; if successful, returns a nonzero integer, otherwise it returns an `eof`

```
char buffer[] = "Hello";
puts(buffer);
```

OR

```
puts("Hello");
```

`int fputs(void *buffer, FILE *fp)` - takes a null-terminated string from memory (the `buffer`) and writes it to the specified file until the terminating `\0` of buffer is reached (it does not write the `\0` character to the file); it is the programmer's responsibility to make sure the `\n` is present at the appropriate place; if successful, returns a nonzero integer, otherwise it returns an `eof`

```
char buffer[] = "Hello";
FILE *outFile = fopen("output", "w");
fputs(buffer, outFile);
```

end of file

`EOF` is a special flag that is defined in `<stdio.h>` to indicate if the end of the file has been reached. An end of file condition exists when the final piece of data has been read from a file. Most of the functions from `<stdio.h>` return a special flag to indicate when a program has reached the end of the file.

```
while ( (c = getc(inFile)) != EOF ) {
        ...
```

`int feof(FILE *fp)` - function to check if the end of the file has been reached. returns a nonzero integer if the file has reached the end of the file, returns zero otherwise; can be used in a while loop, e.g.:

```
// read before the loop
while (!feof(filePtr)) {
        ...
        // read again
}
```

fread and fwrite for large amounts of data

`int fread(void *buffer, int elementSize, int count, FILE *fp)` - most efficient way to read large blocks of data from a file; returns the number of items read

- `buffer` - a pointer to the input area in memory
- `elementSize` - size of a basic data item, often specified using the *sizeof* operator
- `count` - number of data items
- `fp` - file pointer of an open file

--- reads `count` items of data from the file pointed to by `fp` into `buffer`; each item of data is `elementSize` bytes in length

--- for example: `numread = fread(text, sizeof(char), 80, inFile);` will read 80 characters from the file identified by `inFile` and stores them into the array pointed to by `text`; the function returns the number of data items read

`int fwrite(void *buffer, int elementSize, int count, FILE *fp)` - most efficient way to write large blocks of data to a file; returns the number of items written

- `buffer` - a pointer to the output area in memory
- `elementSize` - size of a basic data item, often specified using the *sizeof* operator
- `count` - number of data items
- `fp` - file pointer of an open file

--- copies `elementSize*count` bytes from the address specified by `buffer` to the file pointed to by `fp`

```
/* using fread() and fwrite() */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

      FILE *inFile;
      FILE *outFile;
      int itemsRead;
      int count = 500;
      int data[500];

      inFile = fopen("input.dat", "r");
      if (!inFile) {
            fprintf(stderr, "File open error.  Exiting program\n");
            exit(1);
      }

      outFile = fopen("output.dat", "w");

      while(!feof(inFile)) {
            itemsRead = fread(data, sizeof(int), count, inFile);
            if (ferror(inFile) {
                  fprintf(stderr, "Read error\n");
            }
            fwrite(data, sizeof(int), itemsRead, outFile);
            /* if writing to stdout, you can do the following two lines:
               (fflush() will make sure everything is written)   */
            // fwrite(data, sizeof(int), itemsRead, stdout);
            // fflush(stdout);
      }

      return 0;
}
```

The following example is similar to the previous one except that it is reading characters (instead of integers) from an input file and storing them into a character array, and then writing that character array back out to an output file. If you create an input file adding text to it, or use one that you already have, then you can play around with this program. Change the value of count to see that it reads & writes a different number of characters.

```
/* using fread() and fwrite() */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

	FILE *inFile;
	FILE *outFile;
	int itemsRead;
	int count = 100;
	char data[count];

	inFile = fopen("input.dat", "r");
	if (!inFile) {
		fprintf(stderr, "File open error.  Exiting program\n");
		exit(1);
	}

	outFile = fopen("output.dat", "w");


	itemsRead = fread(data, sizeof(char), count, inFile);
	if (ferror(inFile) {
		fprintf(stderr, "Read error\n");
	}
	fwrite(data, sizeof(char), itemsRead, outFile);

	return 0;
}
```