6 Files and Streams

Large amounts of data are usually stored in data files which are most of the time located in *secondary storage*, typically a read/write type device as a hard disk or a read-only device such as a CD or DVD. Of course, files can also be located and accessed in other types of memory devices or through a network connection. No matter where and how a file is physically stored, in C, all types of files are accessed through the use of an abstraction known as a *stream*, which offer sequential access to the contents of the file.

6.1 Using `fopen` and `fclose`

You access streams in your programs using a pointer to a special structure of type `FILE`. In order to read or write to a file we first need to associate the physical file located in you with a `FILE` pointer, which is done, for example, in the case of a file in your hard disk, by invoking the function `fopen`. The following excerpt shows how to do that:

```
FILE *stream1, *stream2;

stream1 = fopen("input.dat", "r") ;
stream2 = fopen("output.dat", "w");
```

In it we declare two `FILE` pointer variables `stream1` and `stream2` and use `fopen` to *open* the files *input.dat* and *output.dat*, which are the first arguments to `fopen`. These files will be located in the directory from where you're running your program. The second argument to `fopen` is string that specifies how the file should be opened. In this case `"r"` specifies that *input.dat* will only be read and `"w"` specifies that we will only write to *output.dat*.

IMPORTANT: Your program must open a file using `fopen` before it can access it! If file cannot be opened for any reason then the `NULL` pointer will be returned. This occurs, for instance, if the input file is not in the current directory, or your program has no permission to open the file.

It is good practice to always check the value returned by `fopen` to be sure that it is not NULL, as in the next excerpt:

```
FILE *stream1;

if ( (stream1 = fopen("input.dat", "r") == NULL ) {
  printf("Error opening the file \"input.dat\"\n");
```

```
    exit(1);
  }
```

Use the `stdlib` function `exit` to make sure *all* open streams are closed before exiting the program. The argument of `exit` is returned to the parent process, which can be used to check whether your program completed succesfully. Any value different than 0 indicates an error.

The second argument to `fopen` is an access character (mode value), which is usually one of:

Mode	Meaning
r	To open a text file for reading
w	To create a text file for writing
a	To append to a text file
rb	To open a binary file for reading
wb	To create a binary file for writing
ab	To append to a binary file
r+	To open a text file for read/write
w+	To create a text file for read/write
a+	To append or create a text file for read/write
r+b	To open a binary file for read/write
w+b	To create a binary file for read/write
a+b	To append a binary file for read/write

To close a file after we are done with modifying or reading it, use `fclose`. The return value of `fclose` is the integer 0 if successful, and `EOF` (end of file) if an error occurs. The excerpt

```
FILE *seismo1;
...
seismo1 = fopen("earthquake_data", "r");
...
fclose(seismo1);
```

opens the file `"earthquake_data"` for reading then closes it.

IMPORTANT: You must always close your files, because changes you have made may not be committed to the actual file until you close it.

Some types of files and their associated file system support repositioning the stream pointer at certain locations. For example, you may *rewind* the file pointed by stream `stream1` by using:

```
rewind(stream1);
```

which brings the stream back to the first character of the first line of the file. This operation may not be supported for certain types of files and can fail. Related functions are `fseek`, `fsetpos` which will not be discussed in this class.

6.2 Using `fscanf` and `fprintf`

Once an input file has been opened, we can use `fscanf` function to read information from the file or `fprintf` to write information on the file. These functions are identical to `scanf` and `printf`, except that the first argument is a file pointer that specifies the file to be read or written. For example:

```
fscanf(stream1, "%lf %lf", &time, &vel);
fprintf(stream2, "%f %f", time, vel);
```

will read two doubles from `stream1` and write two doubles to `stream2`. If the end of file is reached or an error occurs before any conversion, `fscanf` returns `EOF`. Here is a complete example:

```
/* Program w6-1.c */

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

main() {

  /* declare variables and file pointers */
  int i;
  FILE *fp_in, *fp_out;

  /* open file for reading */
  if ( (fp_in = fopen("input.dat", "r")) == NULL ) {
    printf("Error in opening input file\n");
    exit(1);
  }

  /* read an integer from the input file */
  fscanf(fp_in, "%d", &i);

  /* print the integer on the screen */
  printf("i = %d\n", i);

  /* open file for writing */
  if ( (fp_out = fopen("output.dat", "w")) == NULL ) {
    printf("Error in opening output file\n");
```

```
    exit(1);
  }

  /* write integer on output file */
  fprintf(fp_out, "I read the number '%d'.\n", i);

  /* close files */
  fclose(fp_in);
  fclose(fp_out);

  exit(0);
}
```

The input file `input.dat` must exists in the current directory with an integer in the first line. If the input file is in another directory, use a qualified path name. The program will create a file named `output.dat`.

Here is an example that actually does something with the data in the file:

```
/* Program w6-3.c */

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

main() {

  /* declare variables */
  int nlines, i;
  double x, y, r, theta;
  FILE *fp1, *fp2;

  if ( (fp1 = fopen("input.dat", "r")) == NULL ) {
    printf("Error in opening input file\n");
    exit(1);
  }

  if ( (fp2 = fopen("output.dat", "w")) == NULL ) {
    printf("Error in opening output file\n");
    exit(1);
  }

  /* print message */
  printf("Processing input file...\n");

  /* read number of lines */
  fscanf(fp1, "%d", &nlines);
```

```
  /* write it on the output */
  fprintf(fp2, "%d\n", nlines);

  /* read remaining lines with pairs of doubles */
  i = 0;
  while ( fscanf(fp1, "%lf %lf", &x, &y) != EOF ) {

    /* compute something */
    r = sqrt(x * x + y * y);
    theta = atan2(y, x);

    /* write it on the output */
    fprintf(fp2, "%g %g\n", r, theta);

    /* increment line counter */
    i++;

  }

  if ( i != nlines ) {
    printf("Warning: number of lines on input file is not correct!\n");
  }

  /* close files */
  fclose(fp1);
  fclose(fp2);

  /* print message */
  printf("Done!\n");

  exit(0);
}
```

Can you tell what it does? Note that we are checking the result of `scanf` to determine when we hit the end of the file.

6.3 Using `fgets` and `sscanf`

We now meet the function `fgets` for the second time. This function is used to read text from an input file. If end of file is reached or an error occurs, `fgets` returns `NULL`. The syntax of `fgets` is:

```
  fgets(string_pointer, number_of_characters, file_pointer);
```

so

```
  fgets(line, 81, file1)
```

will read 81 characters from `file1` and store them on the character array `line`. When we used `fgets` earlier we used `stdin` for the file pointer. `stdin` is a true file pointer that points to the current terminal input. Likewise, `stdout` is a file pointer to the current terminal screen. Note that in this way, the terminal and the screen can be treated just like a regular file in C.

The next program reads names from a file and prints them on the screen:

```
/* Program  w6-5.c */

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

main() {

  /* declare line size */
  const int MAX_LINE_SIZE = 120 + 1;

  /* declare variables */
  char line[MAX_LINE_SIZE];
  char *ptr;

  FILE *input;

  /* open input file */
  if ( (input = fopen("names.dat", "r")) == NULL ) {
    printf("Failed to open input file.\n");
    exit(1);
  }

  /* read all lines */
  while ( (ptr = fgets(line, MAX_LINE_SIZE, input)) != NULL ) {

    /* remove leading spaces */
    for ( ; *ptr &&
            (*ptr == ' ' || *ptr == '\t' || *ptr == '\n'); ptr++ );

    while ( *ptr ) {

      /* print name */
      while ( *ptr &&
              !(*ptr == ' ' || *ptr == '\t' || *ptr == '\n' ) )
        printf("%c", *ptr++);

      /* remove trailing spaces */
      for ( ; *ptr &&
```

```
            (*ptr == ' ' || *ptr == '\t' || *ptr == '\n'); ptr++ );

      if ( *ptr )
        printf(" ");

    }

    printf("\n");

  }

  /* close file */
  fclose(input);

}
```

Note that it also trims the names as they are printed.

When one has numbers and strings mixed on the same line it may be harder to distinguish between the entries. Consider for example the file `input.csv` where entries in the same row are separated by commas (aka CSV or Comma Separated Values format):

```
Kate Mills, 09/07/2010, 8.40
Robert Haven, 10/08/2010, 10.00
Kathy Wells, 10/21/2010, 3.20
Marie Jordan, 11/03/2010, 33.00
Robert Haven, 11/01/2010, 12.00
Robert Haven, 11/15/2010, 2.00
Kate Mills, 11/17/2010, 3.25
```

If we use `fgets` to read a line, we will not be able to parse the last entry, which is to be operated as a double, using `fscanf`. One alternative is then to parse this numeric value *after* we read the line. This can be done with the function `sscanf`, which will parse character strings, rather then files. It works pretty much in the same way as `scanf`. For example, in the next program we will use:

```
  sscanf(ptr, "%lf", &value);
```

in order to parse character string `ptr` into the double variable `value`. The following program will use this construction in order to process the above CSV file and calculate a percentage of the numeric input:

```
/* Program  w6-6.c */

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

```c
#include <string.h>

main() {

  /* declare line size */
  const int MAX_LINE_SIZE = 120 + 1;

  /* declare variables */
  char line[MAX_LINE_SIZE];
  char *ptr;

  FILE *input, *output;

  int i;
  char *name;
  char *date;
  double value;

  double tax, total;

  /* open input file */
  if ( (input = fopen("input.csv", "r")) == NULL ) {
    printf("Failed to open input file.\n");
    exit(1);
  }

  /* open output file */
  if ( (output = fopen("output.csv", "w")) == NULL ) {
    printf("Failed to open output file.\n");
    exit(1);
  }

  /* read all lines */
  while ( (ptr = fgets(line, MAX_LINE_SIZE, input)) != NULL ) {

    /* read name */
    name = line;
    for ( ; *ptr && *ptr != ',' ; ptr++ );

    /* mark end of name */
    *ptr++ = '\0';

    /* remove leading spaces */
    for ( ; *ptr &&
            (*ptr == ' ' || *ptr == '\t' || *ptr == '\n' );
          ptr++ ) ;
```

```
    /* read date */
    date = ptr;
    for ( i = 0; *ptr && *ptr != ',' && i++ < 10; ptr++ );

    /* mark end of date */
    *ptr++ = '\0';

    /* read value */
    sscanf(ptr, "%lf", &value);

    /* compute */
    tax = 0.075*value;
    total = value + tax;

    /* write to output file */
    fprintf(output, "%s, %s, %.2f, %.2f, %.2f\n",
            name, date, value, tax, total);

    /* print summary */
    printf("> %s\n", name);
    printf("  Date: %s\n", date);
    printf("     $: %10.2f\n", value);
    printf("   Tax: %10.2f\n", tax);
    printf("    $$: %10.2f\n\n", total);


  }

  /* close files */
  fclose(input);
  fclose(output);

  exit(0);

}
```

A converse function, `sprintf`, also exists, but it is not a safe function as the user has no way to tell `sprintf` the size of the available string buffer. If used, users should be very careful and limit its use to formats with predictable display length.