Raw input and output

We've recently seen how C11 employs arrays of characters to represent strings, treating the NULL-byte with special significance.

At the lowest level, an operating system will only communicate using *bytes*, not with higher-level integers or floating-point values. C11 employs arrays of characters to hold the bytes in requests for *raw input and output*.

File descriptors - reading from a file

Unix-based operating systems provide *file descriptors*, simple integer values, to identify 'communication channels' - such as files, interprocess-communication pipes, (some) devices, and network connections (sockets).

In combination, our C11 programs will use integer file descriptors and arrays of characters to request that the operating system performs input and output on behalf of the process - see *man 2 open*.

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

#define  MYSIZE      10000

void read_using_descriptor(char filename[])
{
//  ATTEMPT TO OPEN THE FILE FOR READ-ONLY ACCESS
    int fd    = open(filename, O_RDONLY);

//  CHECK TO SEE IF FILE COULD BE OPENED
    if(fd == -1) {
        printf("cannot open '%s'\n", filename);
        exit(EXIT_FAILURE);
    }

//  DEFINE A CHARACTER ARRAY TO HOLD THE FILE'S CONTENTS
    char   buffer[MYSIZE];
    size_t got;

//  PERFORM MULTIPLE READs OF FILE UNTIL END-OF-FILE REACHED
    while((got = read(fd, buffer, sizeof buffer)) > 0) {
        .....
    }

//  INDICATE THAT THE PROCESS WILL NO LONGER ACCESS FILE
    close(fd);
}
```

Note that the functions *open*, *read*, and *close* are not C11 functions but operating system**system-calls**, providing the interface between our user-level program and the operating system's implementation.

File descriptors - writing to a file

Similarly, we use integer file descriptors and arrays of characters to write data to a file. We require a different file descriptor for each file - the descriptor identifies the file to use and the operating system (internally) remembers the requested (permitted) form of access.

Copying a file using file descriptors

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

#define  MYSIZE      10000

int copy_file(char destination[], char source[])
{
//  ATTEMPT TO OPEN source FOR READ-ONLY ACCESS
    int fd0    = open(source, O_RDONLY);
//  ENSURE THE FILE COULD BE OPENED
    if(fd0 == -1) {
        return -1;
    }

//  ATTEMPT TO OPEN destination FOR WRITE-ONLY ACCESS
    int fd1    = open(destination, O_WRONLY);
//  ENSURE THE FILE COULD BE OPENED
    if(fd1 == -1) {
        close(fd0);
        return -1;
    }

//  DEFINE A CHARACTER ARRAY TO HOLD THE FILE'S CONTENTS
    char   buffer[MYSIZE];
    size_t got;

//  PERFORM MULTIPLE READs OF FILE UNTIL END-OF-FILE REACHED
    while((got = read(fd0, buffer, sizeof buffer)) > 0) {
        if(write(fd1, buffer, got)) != got) {
            close(fd0); close(fd1);
            return -1;
        }
    }

    close(fd0); close(fd1);
    return 0;
}
```

Reading and writing *text files*

We'll next focus on reading and writing from human-readable *text files*. C11 provides additional support*above* the operating systems's system-calls to provide more efficient *buffering* of I/O operations, and treating text files as a sequence of lines (as strings).

We open a text file using C's *fopen()* function.
To this function we pass the name of the file we wish to open (as a character array), and describe how we wish to open, and later access, the file.

The returned value is a *FILE pointer*, that we use in all subsequent operations with that file.

```
#include <stdio.h>

#define DICTIONARY      "/usr/share/dict/words"

....
//  ATTEMPT TO OPEN THE FILE FOR READ-ACCESS
    FILE   *dict = fopen(DICTIONARY, "r");

//  CHECK IF ANYTHING WENT WRONG
    if(dict == NULL) {
        printf( "cannot open dictionary '%s'\n", DICTIONARY);
        exit(EXIT_FAILURE);
    }

//  READ AND PROCESS THE CONTENTS OF THE FILE
    ....

//  WHEN WE'RE FINISHED, CLOSE THE FILE
    fclose(dict);
```

If *fopen()* returns the special value *NULL*, it indicates that the file may not exist, or that the operating system is not giving us permission to access it as requested.

Declaring how the file will be used

In different applications, we may need to open the file in different ways.

We pass different strings as the second parameter to *fopen()* to declare how we'll use the file:

"r"	open for reading
"r+"	open for reading *and* writing
"w"	create or truncate file, then open for writing
"w+"	create or truncate file, then open for reading *and* writing
"a"	create if necessary, then open for appending (at the end of the file)
"a+"	create if necessary, then open for reading *and* appending

All future operations to read and write an open file are checked against the initial *file access mode*. Future operations will *fail* if they do not match the initial declaration.

NOTE - File access mode flag *"b"* can optionally be specified to open a file in *binary mode* (described later). This flag has effect only on Windows systems, and is ignored on Linux and macOS. This flag has no effect when reading and writing text files.

Reading a text file, one line at a time

Having opened the file (for read access), we now wish to read it in - *one line at a time*.
We generally don't need to store each line of text; we just check or use it as we traverse the file:

The text data in the file is (unsurprisingly) also stored as a sequence of characters. We can use C's character arrays to store each line as we read it:

```
#include <stdio.h>

....
    FILE   *dict;
    char   line[BUFSIZ];

    dict = fopen( ..... );
    ....

//  READ EACH LINE FROM THE FILE,
//  CHECKING FOR END-OF-FILE OR AN ERROR
    while( fgets(line, sizeof line, dict) != NULL ) {
        ....
        ....     // process this line
        ....
    }
//  AT END-OF-FILE (OR AN ERROR), CLOSE THE FILE
    fclose(dict);
```

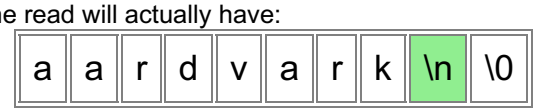
Of note in this code:

- we pass to *fgets()* our character array, into which each line will be read.
- we indicate the maximum size of our array, so that *fgets()* doesn't try to read in too much.
- We use the **sizeof** operator to indicate the maximum size of our character array. Using **sizeof** here, rather than just repeating the value *BUFSIZ*, means that we can change the size of *line* at any time, by only changing the size at its definition.
- We pass our file pointer, *dict*, to our file-based functions to indicate which file to read or write.
  It's possible to have many files open at once.
- The *fgets()* functions returns the constant *NULL* when it "fails". When reading files, this indicates that the end-of-file has been reached, or some error detected (e.g. USB key removed!).
- Assuming that we've reached end-of-file, and that we only need to read the file once, we close the file pointer (and just assume that the closing has succeeded).

What did we just read from the file?

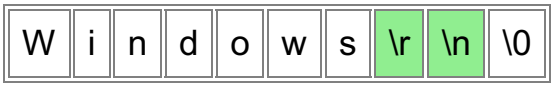
Our call to the *fgets()* function will have read in *all* characters on each line of the dictionary but, if we're interested in processing the characters as simple strings, we find that we've got "too much".

Each line read will actually have:



The character '\n', the familiar *newline* character often used in *print()*, is silently added to text lines by our text editors.

In fact on Windows' machines, text files *also* include a *carriage-return* character before the newline character.



As we know, we can simply turn this into a more manageable string by replacing the newline or carriage-return character by the *null-byte*.

Trimming end-of-line characters from a line

To make future examples easy to read, we'll write a function, named *trim_line()*, that receives a line (a character array) as a parameter, and "removes" the first carriage-return or newline character that it finds.

It's very similar to functions like *my_strlen()* that we've written in laboratory work:

```
//  REMOVE ANY TRAILING end-of-line CHARACTERS FROM THE LINE
void trim_line(char line[])
{
    int i = 0;

//  LOOP UNTIL WE REACH THE END OF line
    while(line[i] != '\0') {

//  CHECK FOR CARRIAGE-RETURN OR NEWLINE
        if( line[i] == '\r' || line[i] == '\n' ) {
            line[i] = '\0'; // overwrite with null-byte
            break;          // leave the loop early
        }
        i = i+1;            // iterate through character array
    }
}
```

We note:

- we simply overwrite the unwanted character with the *null-byte*.
- the function will actually modify the *caller's* copy of the variable.
- we do not return any value.

Writing text output to a file

We've used *fgets()* to 'get' a line of text (a string) from a file;
we similarly use *fputs()* to 'put' (write) a line of text.

The file pointer passed to *fputs()* must previously have been opened for *writing* or appending.

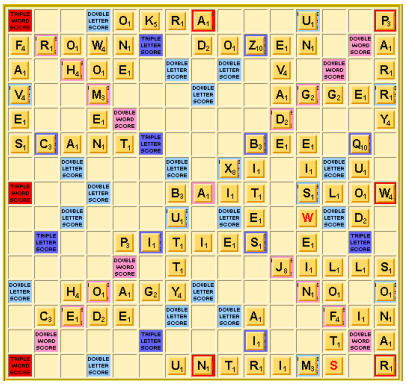
Copying a text file using file pointers

We now have all the functions necessary to copy one text file to another, one line line at a time:

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

void copy_text_file(char destination[], char source[])
{
    FILE        *fp_in   = fopen(source, "r");
    FILE        *fp_out  = fopen(destination,  "w");

//  ENSURE THAT OPENING BOTH FILES HAS BEEN SUCCESSFUL
    if(fp_in != NULL && fp_out != NULL) {
        char    line[BUFSIZ];

        while( fgets(line, sizeof line, fp_in) != NULL) {
            if(fputs(line, fp_out) == EOF) {
                printf("error copying file\n");
                exit(EXIT_FAILURE);
            }
        }
    }
//  ENSURE THAT WE ONLY CLOSE FILES THAT ARE OPEN
    if(fp_in != NULL) {
        fclose(fp_in);
    }
    if(fp_out != NULL) {
        fclose(fp_out);
    }
}
```

More text file reading - the game of Scrabble

Let's quickly consider another example employing reading a text file.



In the game of Scrabble, each letter tile has a certain value - the rarer a letter is in the English language, the higher the value of that letter ('E' is common and has the value 1, 'Q' is rare and has the value 10).

Can we write C functions to determine:

- the value of a word in Scrabble?
- the word in a dictionary with the highest value?
- the best word to choose from a set of tiles?

In writing these functions we won't consider all of the rules of Scrabble.

Another consideration (which we'll ignore) is that there are fixed tile frequencies - for example, there are 12 'E's but only 1 'Q'.

Refer to Wikipedia for the actual distributions.

The value of a word in Scrabble

To answer our Scrabble questions, we'll develop two simple helper functions:

```
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

//  ENSURE THAT A WORD CONTAINS ONLY LOWERCASE CHARACTERS
bool valid_word(char word[])
{
    int i = 0;

//  IF NOT A LOWERCASE CHARACTER, THE FUNCTION RETURNS false
    while(word[i] != '\0') {
 if( ! islower( word[i] )) {   //  if not islower ...
            return false;
        }
        i = i+1;
    }
//  WE'VE REACHED THE END OF THE WORD - IT'S ALL LOWERCASE
    return true;
}

//  CALCULATE THE SCRABBLE VALUE OF ANY WORD
int calc_value(char word[])
{
//  AN ARRAY TO PROVIDE THE VALUE OF EACH LETTER, FROM 'a' TO 'z'
    int values[] = {  1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1,
                      1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10 };
    int total    = 0;
    int i        = 0;

//  TRAVERSE THE WORD DETERMINING THE VALUE OF EACH LETTER
    while(word[i] != '\0') {
        total   = total + values[ word[i] - 'a' ];
        i = i+1;
    }
    return total;
}
```

Letter	Value
A	1
B	3
C	3
D	2
E	1
F	4
G	2
H	4
I	1
J	8
K	5
L	1
M	3
N	1
O	1
P	3
Q	10
R	1
S	1
T	1
U	1
V	4
W	4
X	8
Y	4
Z	10

The word with the highest value

Can we find which *valid* word from a dictionary has the highest value?

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

#define DICTIONARY      "/usr/share/dict/words"
#define LONGEST_WORD    100

//  FIND THE WORD WITH THE BEST VALUE
void findbest( char filename[] )
{
    FILE        *fp  = fopen(filename, "r");

//  ENSURE THAT WE CAN OPEN (WITH READ-ACCESS) THE FILE
    if(fp != NULL) {
        char    bestword[LONGEST_WORD];
        int     bestvalue       = 0;
        char    thisword[LONGEST_WORD];
        int     thisvalue       = 0;

//  READ EACH LINE OF THE FILE
        while( fgets(thisword, sizeof thisword, fp) != NULL ) {
//  REPLACE THE NEWLINE CHARACTER WITH A NULL-BYTE
            trim_line( thisword );

//  ENSURE THAT THIS WORD IS VALID (previously defined)
            if( valid_word(thisword) ) {
                thisvalue = calc_value( thisword );

//  IS THIS WORD BETTER THAN THE PREVIOUSLY BEST?
                if(bestvalue < thisvalue) {
                    bestvalue   = thisvalue;    // save current details
                    strcpy(bestword, thisword);
                }
            }
        }
        fclose(fp);
        printf("best word is %s = %i\n", bestword, bestvalue);
    }
}

int main(int argc, char *argv[])
{
    findbest( DICTIONARY );
    return 0;
}
```

Full solution here.

Reading and writing files of *binary data*

To date, our use of files has dealt exclusively with lines of text, using *fgets()* and *fputs()* to perform our I/O.

This has provided a good introduction to file input/output (I/O) as textual data is easy to "see", and printing it to the screen helps us to verify our functions:

- the standard *fgets* function manages the differing lengths of input lines by reading until the '\n' or '\r' character is found,
- *fgets* terminates input lines by appending a *null-byte* to them, 'turning' them into C strings, and
- the *null-byte* is significant when later managing (copying, printing, ...) strings.

Reading and writing files of binary data, *continued*

However, when managing files of arbitrary data, possibly including *null-bytes* as well, we must use different functions to handle binary data:

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

void copyfile(char destination[], char source[])
{
    FILE        *fp_in   = fopen(source, "rb");
    FILE        *fp_out  = fopen(destination,  "wb");

//  ENSURE THAT OPENING BOTH FILES HAS BEEN SUCCESSFUL
    if(fp_in != NULL && fp_out != NULL) {

        char    buffer[BUFSIZ];
        size_t  got, wrote;

        while( (got = fread(buffer, 1, sizeof buffer, fp_in)) > 0) {
            wrote = fwrite(buffer, 1, got, fp_out);
            if(wrote != got) {
                printf("error copying files\n");
                exit(EXIT_FAILURE);
            }
        }

    }

//  ENSURE THAT WE ONLY CLOSE FILES THAT ARE OPEN
    if(fp_in != NULL) {
        fclose(fp_in);
    }
    if(fp_out != NULL) {
        fclose(fp_out);
    }
}
```

NOTE - The access mode flag *"b"* has been used in both calls to *fopen()* as we're anticipating opening *binary* files. This flag has effect only on Windows systems, and is ignored on Linux and macOS. This flag has no effect when reading and writing text files.

While we might *request* that *fread()* fetches a known number of bytes, *fread()* might not provide them all!

- we might be reading the last "part" of a file, or
- the data may be arriving (slowly) over a network connection, or
- the operating system may be too busy to provide them all right now.

Reading and writing *binary data structures*

The *fread()* function reads an indicated number of elements, each of which is the same size:

```
size_t  fread(void *ptr, size_t eachsize, size_t nelem, FILE *stream);
```

This mechanism enables our programs to read arbitrary sized data, by setting *eachsize* to one (a single byte), or to read a known number of data items each of the same size:

```
#include <stdio.h>

int   intarray[ N_ELEMENTS ];
int   got, wrote;

//  OPEN THE BINARY FILE FOR READING AND WRITING
    FILE  *fp = fopen(filename, "rb+");
    ....

    got = fread( intarray, sizeof int, N_ELEMENTS, fp);
    printf("just read in %i ints\n", got);

//  MODIFY THE BINARY DATA IN THE ARRAY
    ....

//  REWIND THE FILE TO ITS BEGINNING
    rewind(fp);

//  AND NOW OVER-WRITE THE BEGINNING DATA
    wrote = fwrite( intarray, sizeof int, N_ELEMENTS, fp);
    ....

    fclose(fp);
```

Reading and writing binary data structures, *continued*

When reading and writing arbitrary binary data, there is an important consideration - different hardware architectures (the computer's CPU and data circuitry) store and manipulate their data in different formats.

The result is that when binary data *written* via one architecture (such as an Intel Pentium) is*read* back on a different architecture (such as an IBM PowerPC), the "result" will be different!

Writing on a 32-bit Intel Pentium:

```
#include <stdio.h>

#define N     10

  int array[N];

  for(int n=0 ; n < N ; ++n) {
      array[n]  =  n;
  }

  fwrite(array, N, sizeof int, fp_out);
```

Reading on a 32-bit PowerPC:

```
#include <stdio.h>

#define N     10

  int array[N];

  fread(array, N, sizeof int, fp_in);

  for(int n=0 ; n < N ; ++n) {
      printf("%i ", array[n]);
  }
  printf("\n");
```

Prints the output:

```
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512 134217728 150994944
```

The problems of reading and writing of binary data, to and from different architectures and across networks, are discussed and solved in later units, such as *CITS3002 Computer Networks*.

Jonathan Swift's Gulliver's Travels, published in 1726, provided the earliest literary reference to computers, in which a machine would write books. This early attempt at artificial intelligence was characteristically marked by its inventor's call for public funding and the employment of student operators. Gulliver's diagram of the machine actually contained errors, these being either an attempt to protect his invention or the first computer hardware glitch.

The term endian is used because of an analogy with the story Gulliver's Travels, in which Swift imagined a never-ending fight between the kingdoms of the Big-Endians and the Little-Endians (whether you were Lilliputian or Brobdignagian), whose only difference is in where they crack open a hard-boiled egg.