Files

In this lesson, we will cover reading from and writing to files.

WE'LL COVER THE FOLLOWING

Need for file handling

We have seen in the previous lesson that the standard input and output streams can be redirected to and from files and other programs with the `>`, `<` and `|` operators on the terminal. Despite being very powerful, these tools are not suitable in every situation because in many cases programs can not complete their tasks simply by reading from their input and writing to their output.
For example, a program that deals with student records may use its standard output to display the program menu. Such a program may need to search, update or delete specific records for later use. Therefore, this information needs to be stored in files.

Fundamental concepts

Files are represented by the `File` struct of the `std.stdio` module. Since structs have not been introduced yet, we will have to accept the syntax of struct construction as is for now.

Let us go through some fundamental concepts about files before we delve into the example code.

The producer and the consumer

Files that are created on one platform may not be readily usable on other platforms. Merely opening a file and writing data to it may not be sufficient for that data to be available on the consumer's side. The producer and the consumer of the data must have already agreed on the format of the data that is in the file. For example, if the producer has written the id and the name of the student records in a certain order, the consumer must read the data back in the same order.

Additionally, the code samples below do not write a *byte order mark* (**BOM**) to the beginning of the file. This may make your files incompatible with systems that require a BOM. The BOM specifies in what order the UTF code units of characters are arranged in a file.

Access rights

File systems present files to programs under certain access rights. Access rights are important for both data integrity and performance.
When it comes to reading, allowing multiple programs to read from the same file can improve performance, because the programs will not have to wait for each other to perform the read operation. On the other hand, when it comes to writing, it is often beneficial to prevent concurrent accesses to a file, even when only a single program wants to write to it. By locking the file, the operating system can prevent other programs from reading partially written files, from overwriting each other's data and so on.

Opening a file

The standard input and output streams `stdin` and `stdout` are already open when programs start running. They are ready to be used.
On the other hand, normal files must first be opened by specifying the name of the file and the access rights that are needed. As we will see in the examples below, creating a `File` object is sufficient to open the file specified by its name:

```
File file = File("student_records", "r");
```

Closing a file

Any file that has been opened by a program must be closed when the program finishes using that file. In most cases the files need not be closed explicitly; they are closed automatically when the `File` objects cease to exist or are out of the scope of the program.

```
if (aCondition) {
// Assume a File object has been created and used here.
// ...
} // ← The actual file would be closed automatically here when leaving thi
s scope. No need to close explicitly.
```

In some cases, a `File` object may need to be reopened to access a different file or the same file with different access rights. In such cases the file must be closed and reopened with new parameters:

```
file.close();
file.open("student_records", "w");
```

Writing to and reading from files

Since files are character streams, input and output functions `writeln`, `readf`, `readln` etc. are used exactly the same way as with `stdin` and `stdout`. The only difference is that the name of the `File` variable and a dot must be typed with `writeln`:

```
writeln("hello"); // writes to File type variable file
stdout.writeln("hello"); // same as above
file.writeln("hello"); // writes to the specified file
```

`eof()` to determine the end of a file

The `eof()` member function determines whether the end of a file has been reached while reading from a file. It returns `true` if the end of the file has been reached otherwise it returns `false`.
For example, the following loop will be active until the end of the file:

```
while (!file.eof()) {
    // ...
}
```

In the next lesson, we will explore the contents of the std.stdio.File struct in D.