CS141: Lab 5

1. Overview

How To: We'll be dealing with files and exceptions. Your program will read color codes from one file, turn them into a number, and write the colors and number to another file. Some of the lines may have bad data in them, so we'll also be using exception handlers to recover from errors. You can start with a copy of Lab 2 if you like. You don't need to use any functions, procedures, or packages (unless you really want to.)

We'll also be dealing with automatic testing using input files. This is a useful technique where a set of inputs and their expected outputs are put in separate files. For the given input, the program should always produce the same output. This type of testing process makes it easier to find bugs based on the discrepancies.

Prerequisites: *Read these notes carefully first.* This document contains brief introductions to files and exceptions. It will explain how to open, close, read, and write files. It will also explain how to catch errors (the Constraint Errors and Data Errors that you've seen all along.) Since this program uses the same color code as Lab 2, it will be a lot easier if you finished and understood that lab. You should be prepared to refer to previous labs for help with writing simple programs, input, output, user-defined types, and loops.

Objectives: Understand the following:

- Opening and closing files.
- How to read and write data files that are organized into lines.
- Recover from errors while reading and detect the end of a file.
- How to test a program using input and output files.

Why This Is Important: Using only the skills in labs 1-5, you can write practically useful data processing tools based on the *filter model*. These are programs that use loops, custom types, and exception handlers to read an input file in one format and write output in another format, possibly with some very complex calculation in between, and possibly when the input data is not 100% conformant (eg, *dirty data*.)

You can also organize your programs around the concept of *pipelines*, where each program (or even packages in the same program) does a particular type of operation. The pipelines snap together like building blocks. It is particularly common in UNIX-like operating systems (Linux, MacOS) to construct complex programs as '*pipelines of filters*'.

Further, the filter is an easy pattern to follow, and also easy to test and debug. If you can set up a files with a set of known inputs and outputs, you can compare your program's behavior and automatically test its correctness. This is a lot better than testing by hand (or not testing at all.)

Keep In Mind:

- Ada maintains a *pointer* into every file you are reading and writing. The pointer moves forward every time you get or put something. You can move the pointer to a new line using `skip_line` (if you are reading) and `new_line` (if you are writing.)
- Reading and writing from files is a lot like reading and writing from the keyboard and screen in previous labs. The procedure calls are the same, except that you pass the additional parameter `FILE => (your file handle here).`
- *Files* are things on the disk. *File Handles* are things in your program. The handle variable names do not necessarily have the same name as the file you are reading or writing!
- You can think of the screen and keyboard as the default file that gets used if you don't specify another one. That means that if your program is sitting there and not moving, you might have forgotten to pass a `FILE` parameter to a `GET`, and it is waiting you to type something on the keyboard!

2. Assignment and Grading

Color	Digit	Color	Digit
Black	0	Green	5
Brown	1	Blue	6
Red	2	Violet	7
Orange	3	Grey	8
Yellow	4	White	9

Write a program that opens the file `lab5_in.txt`, reads it in a loop, and converts the colors to numbers using the ten-color code from Lab 2. As in Lab 2, each code is four colors, separated by spaces. You should write the original colors and the resulting number to `lab5_out.txt`. You may also write the output to the screen if you want, but you do not have to, and should not confuse output to the screen with output to a file.

You should also tolerate mistakes in the input file, including (but not limited to) colors that aren't in the code. Your program should simply skip to the next line and continue. You do not need to write anything to the output for the "damaged lines". You should also ignore extra colors if any single line has more than four colors on it.

You should keep a count of the number of lines that were successfully processed and a separate count of the number of lines with errors. You should print both of these counts to the screen as the last thing your program does.

While the color code may specify numbers above two billion, you are guaranteed in this lab that none of the inputs will be that large, and so you are welcome to use integer math.

Reading and writing from files is very similar to reading and writing from the screen. See the files given in the Assignments directory. Use `lab5_in.txt` to test your program. The file that your program writes should exactly match `lab5_out.txt`, including spacing and digits after the decimal point. Some examples of input and output appear below.

```
red green green red
orange cherry lemon
blue red blue yellow black
```

This input should generate exactly this output. Note that the line with the non-colors got skipped, and only the first four colors on any line were used.

```
RED GREEN GREEN RED 25500
BLUE RED BLUE YELLOW 6260000
```

This example shows that your program should also gracefully handle bad input data. In our previous programs, that strategy has always been to abort the program with an error message. Ada provides a mechanism, called *exceptions*, that allow you to gracefully recover from these errors. Simply skip the line and try to read another value/unit pair.

Be aware that loops and data files are often used together, and every loop must have a termination condition (eg, a time when it stops.) Your program should detect when it has reached the end of the file (using the `Ada.Text_IO.End_Of_File` procedure) and gracefully exit.

Grading:

A: Correctly handles all examples in the input file, including skipping the lines with errors. Exits gracefully at the end of the file (it should not print any error messages to the screen.) Format of output file matches the example output file. Correct success and failure counts.

B: Minor math or formatting errors, but still runs in a loop, and does most conversions. Broken counters.

C: Doesn't handle exceptions or correctly calculate, but still does some meaningful input and output.

D: Programs that have good source structure and commenting, but which don't compile or run.

Important Note: *Be very careful not to accidentally destroy your `lab5.adb` file either with Ada's Create or Windows' notepad.*

3. Lab Procedure

3.1 Working With Files

It's assumed that you have an idea of what a 'file' in a computer is. `Ada.Text_IO` (and its related packages) provide a means of accessing files sequentially, one symbol at a time. This is sometimes called a *stream*. Streams can be input or output, which means that either symbols are read one at a time, or added to the end of the file, one at a time.[1]

You can make new input files, or inspect the contents of output files, using Notepad, or some other text editor. For sake of simplicity, name these files with a `.txt` type unless you have a good reason to do otherwise.

Streams work much like the screen and keyboard (collectively called the *console*); or more accurately, the screen and keyboard are just another pair of streams. The `GET` and `PUT` procedures in the `Ada.Text_IO` packages work exactly the same way for files as they do for the console, except that you must include a `File=>` parameter. You could also think of `GET` and `PUT` as assuming that `File` will point to the console if you don't specify another object.

Each file you are using must have a *file handle object* (or just file object) inside your program. This is a scratchpad of type `Ada.Text_IO.File_Type` that contains information about the open file. Like other scratchpads, it will have a name, but this name does not have to match the name of the file that will open.

You must open the file once before using it. This tells the computer's operating system to grant access to the file, and sets up a *pointer* to the start of the file. As the name implies, the pointer keeps track of your position in the file. Remember that streams are always read and written one symbol at a time; the pointer ensures that this order is preserved.

The following program gives an example for a file of integers, reading the first one, and printing it. Output files behave similarly, using `PUT` statements instead of `GET`.

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
PROCEDURE Demo1 IS
  MyFile : Ada.Text_IO.File_Type;
  i : Integer;
BEGIN
  Ada.Text_IO.Open(File => MyFile,
    Name => "file_of_numbers.txt", Mode => Ada.Text_IO.In_File);
  Ada.Integer_Text_IO.Get(Item => i, File => MyFile);
  Ada.Text_IO.Put("The first number in the file is: ");
  Ada.Integer_Text_IO.Put(Item => i);
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Close(File => MyFile);
END Demo1;
```

Note that the `Name` passed to `Open` is different from the name of the file scratchpad. The net effect of this program is to advance the pointer exactly enough to read a single integer, then store that integer in `i`. The instructions for printing the number just read should be self-explanatory, but note that the console still works even when other files are open. The program also demonstrates the close procedure, which is not strictly necessary, but it is good practice, especially when dealing with a lot of open files in a single program.

Also note the `Mode` parameter. This may be `Ada.Text_IO.In_File` or `Ada.Text_IO.Out_File`.

1 Ada also allows some files to be in and out at the same time, but generally you won't need to use it.

As the names imply, `In_File` is used when reading a file, and `Out_File` is used when writing.[2] The `Out_File` mode is not usually used with `File_Open`. It is more commonly used with `Create`, which is discussed in the next section.

Loops and files are often used together. For example, the input file in this lab has multiple measurements, so a loop is an obvious way to repeat the read/convert/print loop until the pointer reaches the end of the file. There is a convenient function for testing if the end of the file has been reached, which is suitable for using inside a loop. You can use it like this:

```
WHILE NOT Ada.Text_IO.End_Of_File(File => MyFile) LOOP
  -- Your loop body here
END LOOP;
```

3.2 Creating Files and Output To Files

The examples in the previous section cover reading input from files. In keeping with Ada's cautious design goals, attempting to use `File_Open` on an nonexistent file will raise an exception, even if the file is for output and would be destroyed anyway. It is therefore more convenient to use the `Create` procedure. This will create the file on disk (destroying it if it already existed) and open it.

The following example will read one line from the keyboard and write it to a file, creating that file as it goes. (Please be aware that this is an example and you will not use `Get_Line` or Strings in this lab.)

```
WITH Ada.Text_IO;
PROCEDURE Demo2 IS
  MyFile : Ada.Text_IO.File_Type;
  s : String(1..60); -- We'll talk about this in another lab
  l : Integer;
BEGIN
  s := (Others => ' '); -- Clear before using it (also another lab)
  Ada.Text_IO.Put("Please tell me your name: ");
  Ada.Text_IO.Get_Line(Item => s, Last => l); -- Also another lab
  Ada.Text_IO.Create(File => MyFile, Mode => Ada.Text_IO.Out_File,
    Name => "myname.txt");
  Ada.Text_IO.Put(File => MyFile, Item => s);
  Ada.Text_IO.Close(File => MyFile);
  Ada.Text_IO.Put("I just put your name in the file myname.txt");
END Demo2;
```

Be aware that creating a file will destroy whatever file was there before, without warning. Do not create a file unless you are sure you do not need its previous contents, and especially do not overwrite a program source file (eg, do NOT create "`lab5.adb`"). Your first file-enabled programs will probably use hard-coded file names, which means that they will destroy the results of their last run every time you run them again. If you want to save an output file from one of your programs, use Windows to copy the file to a new name before running the program again.

3.3 Exceptions

As mentioned, *exceptions* indicate an unexpected problem. *Exception Handlers* are a means of recovering from errors. Exceptions commonly occur when reading input from text files, especially in a strongly-typed language like Ada. You have encountered such exceptions before, when putting 'bad data' into previous assignments. If the format of the input data does not match what is expected, then Ada's I/O packages will raise an exception. If you do not define any handlers, then the default behavior of an Ada program is to

2 The notion of input and output streams may remind you of `IN` and `OUT` parameters in procedures. The semantics are similar, in that each file is a 'window' in or out of the program.

simply abort and print the name of the exception.

You may set up a new `BEGIN/EXCEPTION/END` block anywhere in the program, inside another `BEGIN/END` block (procedure or function body, or even inside another exception handler.) It is possible to catch particular types of exceptions by naming them. Alternatively, you can catch any exception (or any other exception) with a `WHEN OTHERS =>` clause.

As soon as any error occurs, no matter where it occurs inside the block, the execution stops and skips to the most recently defined `Exception` handler. This is true even if the program is inside a procedure, function, loop, or other block at the time the program will 'back out' as far as it needs to find an exception handler. If it gets all the way to the top level (your main program) and finds no exception handler, the program exits with an error on the screen.

Part of the Demo1 program is rewritten below, with exception handlers added around the `Get` statement. `Data_Error` is defined in `Ada.Text_IO`, and so the fully-qualified name of the exception is given here.

```
BEGIN
  Ada.Text_IO.Open(File => MyFile,
    Name => "file_of_numbers.txt", Mode => Ada.Text_IO.In_File);
  BEGIN
    Ada.Integer_Text_IO.Get(File => MyFile, Item => i);
    Ada.Text_IO.Put("The first number in the file is: ");
    Ada.Integer_Text_IO.Put(Item => i);
    Ada.Text_IO.New_Line();
  EXCEPTION
    WHEN Ada.Text_IO.Data_Error =>
      Ada.Text_IO.Put(Item => "A Data Error occurred");
      Ada.Text_IO.skip_line(File => MyFile);
    WHEN OTHERS =>
      Ada.Text_IO.Put(Item => "Unknown error occured");
      Ada.Text_IO.skip_line(File => MyFile);
  END;
  Ada.Text_IO.Close(File => MyFile);
END Demo1;
```

Notice the calls to `skip_line`. When an exception is encountered while reading a stream, Ada does not advance the pointer. That means that a naïve loop may read the same bad data over and over again, producing the same exception every time. When reading a text file, you will usually want to skip the bad line and go on to the next line. This has the effect of advancing the pointer to immediately after the next newline in the file. You can do this with the `skip_line` procedure, as shown.

When writing an exception handler, keep in mind what the recovery strategy should be. It is acceptable, for instance, if the exception handler runs inside a loop , to simply try to keep going. The following is an example. Note that the `BEGIN/EXCEPTION/END` block is inside the loop. If the loop was within the `BEGIN/END`, then the `EXCEPTION` would skip to outside the loop, and the loop would not repeat.

```
WHILE (still_alive) LOOP
  BEGIN
    -- Do whatever the loop is going to do here
  EXCEPTION WHEN OTHERS =>
    -- Do nothing! The loop will go again.
    NULL;
  END;
END LOOP;
```

Exceptions may also occur because of errors in the operating system; for example, trying to write a file when the disk is full, or running out of memory. Such errors are unlikely to arise in these labs, but you should be aware that they can occur, especially when dealing with large or complicated programs.

3.4 RAISE Statements (Not used in this lab)

(RAISE statements are part of exceptions, but they are not necessary in Lab 5. It is covered here for sake of completeness. You may safely skip this section.)

It is also possible to make exceptions happen yourself. This is referred to as *raising an exception.* It may be a useful way of 'breaking out' of several layers of procedure calls, or to skip a large block of code when `IF/THEN` statements are inconvenient. The exception type is first defined, similar to a scratchpad, for later use by a `RAISE` instruction. You can select whatever name is most useful for the error, and then use it in a `WHEN` clause later, as shown here:

```
PROCEDURE Demo2 IS
  My_Error: EXCEPTION;
  somthing_bad_happened: boolean;
  error_counter: integer := 0;
BEGIN
  IF (something_bad_happened) THEN
    raise My_Error;
  END IF;
EXCEPTION WHEN My_Error =>
  error_counter := error_counter + 1;
END;
```

4. Testing and Submitting the Program

Run the program. It should automatically read the input file and write the output file, and the output file should exactly match the example `lab5_out.txt` from the assignment directory. Please match the format and spacing exactly. You can check by navigating to your working folder under Windows and opening the file in notepad. Please submit only your `lab5.adb` file. You do not need to submit your `lab5_in.txt` or `lab5_out.txt` files.

5. Looking Ahead

Believe it or not, the Ada you have seen so far (and the analogous constructs in other imperative languages, like C and Java) is sufficient for any programming problem that has a solution at all. However, it is not always convenient or elegant to use only files for data storage. The remainder of the course will deal with *arrays* and *records,* which are the rudiments of *data structures*. Data structures allow you to store a lot of data inside a program at once, in whatever form you decide is most convenient.