Introduction to Systematic Programming
Unit 20 - Input and Output using Files

20.1 Input and output using files

So far our programming examples have either been interactive (ie. taking the input from a keyboard and sending the output to a VDU screen) or they have taken their input from a file and/or sent the output to another file. In the latter case, we have used the process of **input/output re-direction** from within Ada by using the procedures `OpenInput` and `OpenOutput` from the library package `CS_File_IO`.

So far `OpenInput` and `OpenOutput` have been adequate for our non-interactive I/O programming needs in this course. However, these procedures have one major limitation: although as we saw in the programming example in Unit 11, input in a program could be taken from one file and then later from another file, *at any point* during the execution of the program input could only be taken from one source. The only way that the program could take input from a second file would be to close the first input file with `CloseInput` and then open the second file with another call of `OpenInput`. Similarly at any given point in the program output may only be sent to one destination.

Unix input/output re-direction is even more limited since input can be taken from only one file, and output sent to only one file, for the whole execution life-time of the program. Moreover there is no indication in the Ada program itself as to whether it is expecting to perform its input/output interactively from a keyboard/VDU screen, or by reading/writing values using files.

What happens if the program needs to input data values from more than one source? For example, a program might need to process a list of starting and finishing times of a number of competitors in a time-trial race, the starting times might have been recorded in one file by an automatic timing device and the finishing times recorded in another file by a second device. The program might need to copy the start and finish times into one output file and output a list of the race-times to yet another file. This is an example where the program needs alternately input from two files, and output to another two files. Another example is where a program is processing a large amount of data and writing results to a file whilst displaying various informatory messages on the progress of the file processing on a VDU screen.

We see that repeated calls to `OpenInput` to switch input from one file to another are not adequate to cope with these situations since each call of `OpenInput` resets the 'read' position to the start of the file in question. Similarly each time `OpenOutput` is called it deletes any existing contents of the specified file and starts writing the file afresh.

20.1.1 Defining file variables

Ada lets us deal with situations like the ones above by permitting us to define **file variables** in our program, which are used to represent the files on which the program needs to perform input or output operations. A separate file variable is used for each physical disc file used by the program. We may then associate these file variables with actual physical disc files, and perform input and output steps by specifying with which file variable the input or output is to be performed.

To illustrate the details of what is involved, let us see how we might arrange for the input/output requirements of the timing problem described above.

We start by declaring the four file variables (corresponding to the two input and two output files) needed by the program. We would do this using the declarations:

```
StartTimes  : File_Type;  -- For the starting times
FinishTimes : File_Type;  -- For the finishing times
Merged      : File_Type;  -- For the merged start and finish times
Timings     : File_Type;  -- For race times
```

Notice that we should choose identifiers for these four file variables which give some indication of their use, ie. choose meaningful identifiers. The type `File_Type` is not pre-defined in Ada, but rather, like all other means of performing input/output is provided by a package (in this case the package `Ada.Text_IO`). Thus `File_Type` needs to be 'imported' by including, at the head of the program, the context clauses:

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
```

20.1.2 Associating file variables with physical disc files

The package Ada.Text_IO also provides the means by which the file variables can be associated with physical disc files. It does so through the provision of the procedure Open which may be called as follows:

```
Open(File => StartTimes,  Mode => In_File, Name => "starts.dat");
Open(File => FinishTimes, Mode => In_File, Name => "finishes.dat");
```

In the above, the first actual parameter (of mode IN OUT) is the file variable, the second actual parameter (of mode IN) specifies the way in which the file is to be used by the program; In_File indicates that the file is to be used for input only. The third actual parameter (also of mode IN) is a string giving the name of the actual physical disc file to be associated with the file variable. In this example StartTimes and FinishTimes are both to be used for input only, and have now been 'connected' to the physical disc files starts.dat and finishes.dat respectively. Of course these two disc files must already exist and contain appropriate data values.

We need to do something similar for the output files, namely if the two files already exist we must open them for output with calls to the procedure Open:

```
Open(File => Merged,  Mode => Out_File, Name => "merged.res");
Open(File => Timings, Mode => Out_File, Name => "timings.res");
```

which will 'connect' the file variables Merged and Timings to the existing physical disc files merged.res and timings.res respectively. Note that both these file are to be used for output only, and this has been indicated by specifying the Mode parameter as Out_File. Note that the existing contents of the two output files are completely erased by the calls to Open; nevertheless, Ada requires that the output files should already exist.

If the disc files to be used for output do not already exist, new files must be created and opened for output with calls to the procedure Create:

```
Create(File => Merged,  Mode => Out_File, Name => "merged.res");
Create(File => Timings, Mode => Out_File, Name => "timings.res");
```

This will create two new (empty) physical files on disc with the specified names and 'connect' the file variables Merged and Timings to these two new files.

As we have seen above the existing contents of a file are erased when the file is opened for output using the mode parameter Out_File. Sometimes we may wish to retain the original contents of a file and simply append extra output to the end of the file. To achieve this we open the file timings.res (say) for output with a call to the procedure Open as follows:

```
Open(File => Timings, Mode => Append_File, Name => "timings.res");
```

This call will 'connect' the file variable Timings to the existing physical disc file timings.res. The file is opened for output without erasing its current contents by specifying the Mode parameter as Append_File.

20.1.3 Input/output using file variables

Now values may be input from, or output to, files with new versions of the procedures Get and Put which are imported from Ada.Text_IO, CS_Int_IO or CS_Flt_IO as appropriate. These versions include an additional file variable parameter of mode IN to indicate where the Get or Put action is to take effect. Thus to input the next pair of start and finish times from the files starts.dat and finishes.dat we would write (assuming the prior declaration of Integer variables Start and Finish):

```
Get(File => StartTimes,  Item => Start);
Get(File => FinishTimes, Item => Finish);
```

The first of these steps causes the next Integer value from the disc file associated with the file variable StartTimes (ie. from the disc file starts.dat) to be input into the variable Start. The second causes the next Integer value in the file finishes.dat (since it is associated with the file variable FinishTimes) to be input into the variable Finish.

Similarly, to output values to the files merged.res and timings.res we would write :

```
Put(File => Merged,  Item => Start,  Width => 10);
Put(File => Merged,  Item => Finish, Width => 10);
Put(File => Timings, Item => Finish - Start);
```

The first two steps output the values of the variables `Start` and `Finish` to the disc file associated with the file variable `Merged` (ie. to the disc file `merged.res`) whereas the third outputs the `Integer` value `Finish - Start` to the disc file `timings.res` as this is associated with the file variable `Timings`.

Note that we can produce formatted output by using the `Width` parameter in the standard way. Similarly we may use the parameters `Fore` and `Aft` for formatted output of values of type `Float`. These versions of the procedure `Put` for outputting `Integer` and `Float` values to files have the same default values for their formal parameters `Width`, `Fore` and `Aft` as their interactive counterparts.

We may also start new lines in our output files by using a different version of the procedure `New_Line` which has a file variable parameter of mode `IN` to indicate *in which file* the new line is to be output. Thus the procedure calls:

```
New_Line(File => Merged);
New_Line(File => Timings, Spacing => 3);
```

would output an e-o-l marker to the disc file `merged.res` and three e-o-l markers to the disc file `timings.res` respectively. Similarly we may also move to the start of a new line in our input files by using a different version of the procedure `Skip_Line` which has a file variable parameter of mode `IN` to indicate *in which file* the action is to take place. Thus the procedure calls:

```
Skip_Line(File => StartTimes);
Skip_Line(File => FinishTimes, Spacing => 4);
```

would skip to just after the next e-o-l marker in the disc file `starts.dat` and to just after the fourth e-o-l marker in the disc file `finishes.dat` respectively.

20.1.4 Detecting end of line and end of file

The package `Ada.Text_IO` also provides the means by which end of line and end of file may be detected. Extra versions of the functions `End_Of_Line` and `End_Of_File`, which have a file variable parameter of mode `IN` to indicate *which file* to test, are provided. For example to display the file `finishes.dat` on the VDU screen we could write (assuming the prior declaration of a variable `Ch` of type `Character`):

```
WHILE NOT End_Of_File(File => FinishTimes) LOOP
   WHILE NOT End_Of_Line(File => FinishTimes) LOOP
      Get(File => FinishTimes, Item => Ch);
      Put(Ch);
   END LOOP;
   Skip_Line(File => FinishTimes);
   New_Line;
END LOOP;
```

In fact all I/O procedures you have used so far in this course have versions with a file parameter.

20.1.5 What to do when the program has finished with a file

The package `Ada.Text_IO` also provides the means by which a file variable can be disconnected from its physical disc file when the program has finished reading from, or writing to the file. It does so through the provision of the procedure `Close` taking a single file variable parameter of mode `IN OUT` which may be called as follows:

```
Close(File => StartTimes);
Close(File => FinishTimes);
Close(File => Merged);
Close(File => Timings);
```

The above will have the effect of 'disconnecting' all the physical disc files used by our example program.

20.1.6 Using file variables

We need not worry about the form of the values stored in variables of type `File_Type` and, indeed, these are likely to vary from one type of computer to the next. We simply use them with the procedures `Open`, `Close`, `Get` and `Put` etc. as a means of specifying (in a machine-independent manner) the files to be used in our I/O operations.

In fact Ada prevents us from looking at the 'inner structure' of the type `File_Type` as it is a 'private' type. This means that its definition is private to the body of the library package `Ada.Text_IO` which exports it and furthermore the operations that can be performed on variables of this type are limited to those provided by `Ada.Text_IO` and other I/O library packages. Clearly it would not make much sense to attempt to perform arithmetic operations on file variables, but Ada even prevents us from assigning values directly to variable of type `File_Type`, thus, for example:

```
File1, File2 : File_Type;
.................
Open(File => File1, Mode => In_File, Name => "somefile.dat");
File2 := File1;   -- !? illegal in Ada to copy file values
```

Similarly we are not allowed to compare two file variables for equality or inequality. In fact essentially the only thing we may do with variables of type `File_Type` is to use them as parameters to the procedures provided by `Ada.Text_IO`, `CS_Int_IO` and `CS_Flt_IO`. The rationale behind this is that indiscriminate use of these operations (particularly assignment) could perhaps compromise the integrity of the I/O system.

20.2 More on input and output redirection in Ada

The general file I/O in Ada described in [20.1] is very flexible and allows us to have several files open at once for input and to interleave input steps from the various files. Similarly we may have several files open for output and alternately send output to each of these files. In such cases it is clearly necessary to specify a file parameter when calling the I/O procedures; how else could the computer determine whence to take input and whither to send output?

However, in certain data processing applications the majority of `Get` steps might input data from a single file and the majority of `Put` steps might output results to just one file. Occasionally we might also wish to take input from another source file and/or send output to another destination file before resuming I/O operations with the original files from the positions where we left off. In such cases it is rather inconvenient to have to specify a file parameter in each and every I/O step.

To overcome this sort of inconvenience the library package `Ada.Text_IO` provides two procedures, namely:

```
Set_Input
```

and:

```
Set_Output
```

which, when called with a file parameter, have the effect of making the associated physical file the default source of input or the default destination for output respectively. To use these procedures we must open a physical file for input or output in the normal way using a call to `Open`, for example:

```
Open(File => MainInput,   Mode => In_File,  Name => "main.dat");
Open(File => MainOutput,  Mode => Out_File, Name => "main.res");
Open(File => OtherInput,  Mode => In_File,  Name => "other.dat");
Open(File => OtherOutput, Mode => Append_File, Name => "other.res");
```

Then we call `Set_Input` or `Set_Output`, as appropriate, to redirect I/O to the specified file and can subsequently use the forms of `Get`, `Put` etc. without file parameters for our I/O operations:

```
-- Do some input and output with "main" files
Set_Input(MainInput);   Set_Output(MainOutput);
Get(SomeData);
..............
Put(SomeValue);

-- Now do some I/O with the "other" files
Set_Input(OtherInput);   Set_Output(OtherOutput);
Get(SomeData);
..............
Put(SomeValue);

-- Revert to input and output with "main" files
Set_Input(MainInput);  Set_Output(MainOutput);
Get(SomeData);
..............
Put(SomeValue);
```

This technique is really only appropriate if we intend to perform a considerable number of I/O operations on the subsidiary files before reverting to I/O with the main files. If we intend to take input from two or more files in rapid succession, it is generally better to use the form of `Get` with an explicit file variable parameter.

Suppose that, after redirecting I/O to files as above, we wish temporarily to revert to interactive I/O. For example we might wish to prompt a user on the VDU screen to select between a number of processing options. The user would then input the desired options using the keyboard. We need to 'tell' the `Get` and `Put` procedures to perform interactive I/O. How can this be done?

The package `Ada.Text_IO` provides two parameter-less functions to facilitate this, namely:

```
Standard_Input
```

and:

```
Standard_Output
```

These functions return values of type `File_Type` giving the original default input source (normally the computer keyboard) and the original default output destination (usually the computer's VDU screen) respectively[1]. Thus to revert temporarily to interactive I/O we could write:

```
-- Do some input and output with "main" files
Set_Input(MainInput);  Set_Output(MainOutput);
Get(SomeData);
.............
Put(SomeValue);

-- Now do some interactive I/O
Put(File => Standard_Output,
    Item => "Continue? Please type Y or N: ");
Get(File => Standard_Input, Item => Char);
Skip_Line(File => Standard_Input);

-- Revert to input and output with "main" files
Get(SomeData);
.............
Put(SomeValue);
```

Of course if we intend to perform a considerable number of interactive I/O operations, it might be more convenient to temporarily make interactive I/O the default with the procedure calls:

```
Set_Input(Standard_Input);
Set_Output(Standard_Output);
```

and then use the simple forms of `Put` and `Get` for interactive I/O, as we did in the first example in this section, before reverting to I/O with the main files with the procedure calls:

```
Set_Input(MainInput);
Set_Output(MainOutput);
```

20.3 The Library Package `CS_File_IO`

As its name might suggest, the package `CS_File_IO` is not a standard Ada package; instead it was written to provide simple file I/O facilities specially for the ISP course. Using the procedures provided by `CS_File_IO` it is possible to redirect input and output without needing to worry about the complexities of using file variables and the procedures `Open`, `Create`, `Close`, `Set_Input` and `Set_Output` discussed above. However, these procedures (from `Ada.Text_IO`) are used in the package body of `CS_File_IO` to implement the procedures: `OpenInput`, `OpenOutput`, `CloseInput` and `CloseOutput`.

Two versions of `OpenInput` and of `OpenOutput` are provided: the first version of each takes a string parameter of mode `IN` which is used to specify the name of the physical file to be opened. The second version takes no parameters; when this second version is called the user is prompted to enter the name of the file interactively. The file variables `Input` and `Output` used to handle file input and output are encapsulated within `CS_File_IO` and so their existence is hidden from the user. The

[1] When input and output are redirected to files at operating system level (using the Unix redirection operators > and <), this operation is 'invisible' to Ada and so in this case `Standard_Input` and `Standard_Output` return values which refer to these files rather than to the keyboard and VDU.

package body of CS_File_IO is presented below; for the sake of brevity only the coding of the input redirection procedures is given, that of the output procedures is similar[2].

```
WITH Ada.Text_IO;

PACKAGE BODY CS_File_IO IS

   MaxPathLen : CONSTANT Positive := 1024;
   SUBTYPE PathString IS String(1 .. MaxPathLen);
   Input, Output : Ada.Text_IO.File_Type;

   PROCEDURE OpenInput (FileName : IN String) IS
      -- Opens the file FileName for input and then sets it
      -- to be the default input stream.
   BEGIN
      Ada.Text_IO.Open(File => Input,
                       Mode => Ada.Text_IO.In_File,
                       Name => FileName);
      Ada.Text_IO.Set_Input(File => Input);
   END OpenInput;

   PROCEDURE OpenInput IS
      -- Prompts the user for a file name, opens it for input and
      -- then sets it to be the default input stream.
      PathLen  : Natural;
      FileName : PathString;
   BEGIN
      Ada.Text_IO.Put(File => Ada.Text_IO.Standard_Output,
                      Item => "Input file> ");
      Ada.Text_IO.Get_Line(File => Ada.Text_IO.Standard_Input,
                           Item => FileName,
                           Last => PathLen);
      Ada.Text_IO.Open(File => Input,
                       Mode => Ada.Text_IO.In_File,
                       Name => FileName(1 .. PathLen));
      Ada.Text_IO.Set_Input(File => Input);
   END OpenInput;

   PROCEDURE CloseInput IS
      -- Closes the file associated with the current default input
      -- stream.
      -- The default input stream reverts to Standard_Input.
   BEGIN
      Ada.Text_IO.Set_Input(File => Ada.Text_IO.Standard_Input);
      Ada.Text_IO.Close(File => Input);
   END CloseInput;

END CS_File_IO;
```

20.4 Programming example - Merging several sorted files

Suppose it is required to merge several lists of integers each of which is already sorted into ascending numerical order into a single ordered list. Thus, for example, the three lists:

First list	**Second list**	**Third list**	should produce:	**Merged list**
2345	1356	2217		1356
3456	6521	2244		2217
4567				2244
				2345
				3456
				4567
				6521

[2] Actually OpenOutput is rather more complex as it first attempts to open the file for output with Open; if that fails (presumably because the file does not exist), it attempts to open the file for output using Create; this uses the technique of exception handling which is beyond the scope of this introductory course. The full package CS_File_IO can be inspected in the directory ~barnesa/IOlib on Sparcs or in the sub-folder CS_IOlib of the folder CS110 (AB) on the public_info volume on Apple-Server.

We could read the contents of all the files, one after another, into a single array and then sort the array however, this is rather inefficient. We need a better algorithm which utilises the fact that the lists to be merged are already sorted.

20.4.1 Outline Algorithm

i) Input first number from each input file.

ii) Determine the smallest of these numbers.

iii) Output this number to the output file.

iv) Replace the number just output with the next number from the file from which the number originated (or close the file if all numbers in this file have been processed).

v) Repeat from step (ii) until we have processed all the data in the input files.

20.4.2 Data Structures

There are a number of ways of determining whether all the data in the input files has been processed. The simplest is to keep a count of the number of files still containing unprocessed data (`OpenFileCount` say). If all the values in a file have been processed we decrement this count by 1 as part of step (iv) of the outline algorithm. For simplicity we will also suppose that the number of files to be merged (`NumFiles` say) is known in advance and so we need initialise to `OpenFileCount` to `NumFiles`.

We will need an array of records to store the following information about each input file. Each record will contain three fields: the file variable associated with the file when it is opened (this will be supplied as a parameter in calls to `Get` to cause data to be input from the appropriate input file); the value currently being processed from the file; and a boolean field to indicate when all values from the file have been processed. This boolean field will be set during step (iv) of the outline algorithm if all values in the file have been processed.

```
TYPE FileRecord IS
             RECORD
                File     : File_Type; -- Imported from Ada.Text_IO.
                NextNum  : Integer;   -- To hold current number in file.
                Done  : Boolean := False;
             END RECORD;

TYPE FileRecs IS ARRAY(1 .. NumFiles) OF FileRecord;
FileList : FileRecs;
```

20.4.3 Top-level Design

```
Open all input files ..................................................... A
Open the output file ..................................................... B
OpenFileCount := NumFiles
Read first value from each file and store in FileList array ..... C
WHILE OpenFileCount > 0 LOOP
   Determine smallest of the available values in FileList
      array and the Index of the file from which it came ..... D
   Output this smallest value ..................................... E
   Read next value from Index^th file and store in NextNum
      field of corresponding file record, or if file is
      exhausted close the file and set Done field of file
      record to True and decrement OpenFileCount ................. F
END LOOP
Close the output file .................................................... G
```

20.4.4 Refinements

D: This is essentially a straightforward algorithm to find a smallest value in an array and its location. However we must be careful not to process 'garbage' values from files which are already exhausted.

```
      NumToOutput := Integer'Last;   -- Largest possible integer.
      FOR I IN 1 .. NumFiles LOOP
           IF NOT FileList(I).Done
                      AND THEN FileList(I).NextNum <= NumToOutput THEN
                NumToOutput := FileList(I).NextNum;
                Index := I;
           END IF;
      END LOOP;
F:    IF End_Of_File(File => FileList(Index).File) THEN
           Close(File => FileList(Index).File);
           FileList(Index).Done := True;
           OpenFileCount := OpenFileCount - 1;
      ELSE
           Get(File => FileList(Index).File,
               Item => FileList(Index).NextNum);
      END IF;
```

C: We may use a loop:

```
      FOR I IN 1 .. NumFiles LOOP
```

with a body consisting of essentially the same steps as **F** above. Note that this correctly caters for the case when one or more of the input files is empty. This suggests that we procedurise step **F** as a procedure (`ReadNextInput` say).

The remaining refinements are quite straightforward and are omitted.

20.4.5 Full Program

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
WITH CS_Int_IO;    USE CS_Int_IO;

PROCEDURE Merge IS
   -- Example Program in Unit 20 of ISP course.
   -- Written by A Barnes, August 1996
   -- Merges a number of sorted files of integers into a single file.

   NumFiles : CONSTANT Positive:= 6;        -- Number of files to merge.
   SUBTYPE FileRange IS Positive RANGE 1 .. NumFiles;
   SUBTYPE FileCount IS Natural  RANGE 0 .. NumFiles;

   TYPE FileRecord IS
              RECORD
                 File     : File_Type; -- Imported from Ada.Text_IO.
                 NextNum  : Integer;   -- To hold current number in file.
                 Done     : Boolean := False;
                                       -- Set to True when all values in
                                       -- the file have been processed.
              END RECORD;

   -- Array of records to hold details of each input file.
   TYPE FileRecs IS ARRAY(FileRange) OF FileRecord;

   -- Constants and types for use in procedures for opening i/o files.
   MaxPathLen : CONSTANT Positive:= 1024;
                                    -- Longest allowed Unix file name.
   SUBTYPE PathType IS String(1 .. MaxPathLen);

   ------------------------------------------------------------------------

   PROCEDURE OpenInputFile(FileID : OUT File_Type) IS
      FileName : PathType;
      Length   : Natural;  -- To hold length of name of an I/O file.
   BEGIN
      Put_Line("Please enter the name of the next file to be merged");
      Get_Line(Item => FileName, Last => Length);
      Open(File => FileID, Mode => In_File, Name => FileName(1..Length));
   END OpenInputFile;
```

```
   PROCEDURE OpenOutputFile(FileID : OUT File_Type) IS
      FileName : PathType;
      Length : Natural;      -- to hold length of PathName of I/O files
   BEGIN
     Put_Line("Please enter the file name for the merged output");
     Get_Line(Item => FileName, Last => Length);
     Open(File => FileID, Mode => Out_File, Name => FileName(1..Length));
   END OpenOutputFile;

   ------------------------------------------------------------------------
   PROCEDURE ReadNextInput(FileRec : IN OUT FileRecord;
                           NumOpen : IN OUT FileCount) IS
   BEGIN
      IF End_Of_File(File => FileRec.File) THEN
         Close(File => FileRec.File);
         FileRec.Done := True;
         NumOpen := NumOpen - 1;
      ELSE
         Get(File => FileRec.File, Item => FileRec.NextNum);
      END IF;
   END ReadNextInput;

   -- Main program variables ----------------------------------------------

   OpenFileCount : FileCount := NumFiles;
                       -- Number of input files still open.
   NumToOutput   : Integer;
   Index : FileRange; -- Index of file from which NumToOutput originated.
   FileList : FileRecs;
   Output   : File_Type;  -- Imported from Ada.Text_IO.

BEGIN
   -- Open all the input files.
   FOR I IN FileRange LOOP
       OpenInputFile(FileID => FileList(I).File);
   END LOOP;

   OpenOutputFile(FileID => Output);

   -- Input first data item from each file.
   FOR I IN FileRange LOOP
      ReadNextInput(FileRec => FileList(I), NumOpen => OpenFileCount);
   END LOOP;

   WHILE OpenFileCount > 0 LOOP
      -- Find the (smallest) number to be output next.
      NumToOutput := Integer'Last;      -- Initialise to largest Integer.
      FOR I IN FileRange LOOP
         IF NOT FileList(I).Done
                     AND THEN FileList(I).NextNum <= NumToOutput THEN
            NumToOutput := FileList(I).NextNum;
            Index := I;
         END IF;
      END LOOP;

      -- Output this (smallest) number.
      Put(File => Output, Item => NumToOutput);
      New_Line(File => Output);

      -- Read next data item from the appropriate input file.
      ReadNextInput(FileRec => FileList(Index),
                    NumOpen => OpenFileCount);

   END LOOP;

   Close(File => Output);

END Merge;
```