Intermediary COBOL

“*COBOL is for morons*.”

Edsger Dijkstra

This unit will cover the following topics

1. I/O Statements
2. I/O Example
3. Making Decisions
4. Modular Programs and Subprograms
5. Loops
6. Program Blocks
7. Unconditional Control Transfer
8. Terminating Programs
9. File I/O of Structures

1. I/O Statements

There are four aspects of I/O in Cobol:

1. The **input-output** section of the **environment division**.
2. The **file section** of the **data division**.
3. The editing records of the **data division**.
4. The **open**, **close**, **read** and **write** statements in the **procedure division**.

The **input-output** section of the **environment division** is used to assign symbolic file names to the devices used for input and output. These assignments are achieved using **select** statements. The general form of the **select** statement is:

```
    select file-name assign to system-name.
```

Here, **file-name** is any legal Cobol identifier, and **system-name** is the name given to some I/O device. For example:

```
    select ifile assign to "test.dat".
```

The **file section** of the **data division** is used to define the files and records that are associated with the symbolic file names that were assigned to the various system devices.

```
    fd file-name
        label records are omitted.
        data record is record-name-list.
```

where **record-name-list** is a list consisting of the names of the records associated with the **file-name**. The records must be defined immediately following the file definition. The definition of records occurs directly after the file definition. Consider the following sample record:

```
    01 input-record.
       05 filler     pic  x(6).
       05 data-item  pic  9(5)V9(2).
       05 filler     pic  x(12).
       05 last-item  pic  x(55).
```

Characters 7-13 and 26-80 are referenced by the variable names **data-item** and **last-item**.

The fourth aspect of Cobol I/O involves the **open**, **close**, **read** and **write** statements. Firstly the file must be opened, using an open statement of the form:

```
    open input input-file-list, output output-file-list.
```

For example:

```
open input ifile.
```

After the last I/O statements, the files are closed, using a close statement of the form:

```
close file-name-list.
```

The **read** statement is used to assign input values to variable names. Its general form is:

```
read file-name into record-name
    at end statement
end-read.
```

The **write** statement is used to output the contents of a record to a file. Its general form is:

```
write record-name from output-record
    after advancing constant line[s].
```

2. I/O Example

Consider a file which contains a series of two-digit numbers of the form:

```
     23
     45
     56
     72
     ...
```

A program to read in these numbers and calculate their sum is explained as follows. Firstly the **environment division** is created:

```
environment division.
input-output section.
file-control.
select ifile assign to "test.dat"
    organization is line sequential.
```

This assigns the symbolic file **ifile** to the physical ASCII file **test.dat**, which contains a series of numbers, each on a separate line. The last line specifies that the file is organized as **line sequential**, meaning that the program reads one record after another in the same order in which the records were entered when the file was created.

The data division specifies how the data is set up.

```
data division.
file section.
fd ifile.
01 input-record.
   05  num        pic  9(2).

working-storage section.
77 eof-switch    pic  9 value 1.
77 numsum        pic  999.
01 out-record.
   05  out1      pic  X(8) value "number =".
   05  filler    pic  X.
   05  out2      pic  99.
```

Here a file descriptor in the form of a record is defined for the file associated with **ifile**. The record **input-record** has one entry, a variable **num** which, in which each value read from the file is stored during processing - the 9(2) specifies a 2-digit numeric picture string. In the **working-storage section**, a variable **eof-switch** is created with an initial value of 1 - it is used to flag when the end-of-file occurs. The variable **numsum** holds the cumulative sum of the numbers in the file. Finally the record out-record holds the formatting for the display of the numeric values as they are read in from the file. It is of the form:

```
     “number =” + “  “ + “00”      = “number = 00”
```

Next is the **procedure division**. This does the actual processing.

```
procedure division.

    open input ifile.

    move 0 to numsum.

    perform sum-numbers
        until eof-switch = 0.

    close ifile.
    display "Sum = " numsum.

stop run.
```

The first statement opens the file associated with **ifile** for input. Next the value of **numsum** is set to zero. Then a loop is used to execute the “paragraph” or subprogram **sum-numbers**, until the variable **eof-switch** is set to zero. The file **ifile** is then closed, and the value of **numsum** is displayed. The paragraph **sum-numbers** is explained below.

```
sum-numbers.
    read ifile into input-record
        at end move zero to eof-switch
    end-read.
    if eof-switch is not equal to zero
        add num to numsum
        move num to out2
        display out-record
    end-if.
```

Firstly, a **read** statement is used to input the next piece of data from the file into the record **input-record**, in this case storing the value in **num**. If EOF is encountered then the variable **eof-switch** is set to zero. An **if** statement is then used to determine if EOF has been encountered. If not, then the value of **num** is added to **numsum**, the value of **num** is copied to **out2**, from the record **out-record**, and **out-record** is displayed.

3. Making Decisions

3.1 if-else

The basic form of the Cobol **if** statement (Cobol74) is:

```
if condition
     sentence-1
else
     sentence-2.
```

The **condition** is any conditional expression constructed using one of the relational operators, **sentence-1** and **sentence-2** are any Cobol sentences (which has its ending period omitted for **sentence-1**).

For example, to derive the maximum value of two variables:

```
if a is greater than b
     move a to max-val
else
     move b to max-val.
```

Blocks of sentences can also be used:

```
if denom is less than 0
     move 0 to result
     add 1 to n
else
     divide numer by denom giving result
     add 1 to m.
```

To continue execution when a condition is false, use the **next sentence** clause.

```
if denom is greater than 0 then
     divide numer by denom giving result
else
     next sentence.
```

⚠ In modern compilers (Cobol85), the use of the term **next sentence** is archaic. The compiler will produce a warning of the form:

```
Warning: NEXT SENTENCE is archaic in OpenCOBOL
```

Instead the term **end-if** can be used. In early versions of Cobol an **if** statement was terminated with a period. Cobol 85 introduced the use of the **end-if** statement. Cobol85 modified the **if** statement, to the form:

```
if condition then
     sentence-1
else
     sentence-2
end-if.
```

Statements that make up the then-part or else-part of an if statement should not end with a period.

if statements can also be *nested*, for example:

```
if a is equal to 0 then
    if b is equal to 0 then
        add 1 to c
    end-if
end-if.
```

This can be replaced with a compound condition of the form:

```
if a is equal to 0 and b is equal to 0 then
    add 1 to c
end-if.
```

Cobol 74	**Cobol 85**

`if number < 0`
`    move “-” to numsign`
`else if number > 0`
`    move “-” to numsign`
`else`
`    next sentence.`
	`if number < 0`
	`    move “-” to numsign`
	`else if number > 0`
	`         move “-” to numsign`
	`     end-if`
	`end-if.`

Examples:

```
if time greater than norm-time
    compute overtime = time - norm-time.

if time > norm-time
    compute overtime = time - norm-time
    compute extra-salary = overtime * hourly-pay
```

```
      else move 0 to extra-salary.
```

For example:

Cobol 74	**Cobol 85**

if humidity > 70
    move 0.483199 to eqMC
else
    move 0.160107 to eqMC.
	if humidity > 70
	    move 0.483199 to eqMC
	else
	    move 0.160107 to eqMC
	end-if.

3.2 Relational Conditions

The relational operators are:

Relational Operator	Cobol expression	Example

`equal`	`is equal to`
	`is =`
		a **is equal to** be

`not equal`	`is not equal to`
	`is not =`
		a **is not =** 17

`less than`	`is less than`
	`is <`
		(a-b) **is <** 0

`not less than`	`is not less than`
	`is not <`
		humid **not <** 0

`greater than`	`is greater than`
	`is >`
		(a-b) **is >** 0

`not greater than`	`is not greater than`
	`is not >`
		humid **not >** 100

`less than or equal to (Cobol85)`	`is less than or equal to`
	`is < =`
		humid **is < =** 100

`greater than or equal to (Cobol85)`	`is greater than or equal to`
	`is > =`
		humid **is > =** 0

The keywords **is, to** and **than** are optional and are only included for readability.

3.3 Combined Conditions

The logical operators are:

```
not, and, or
```

3.4 Abbreviated Conditions

⚠ Sometimes it is possible to abbreviate conditions. For example:

```
prcnt > 0 or prcnt < 100
```

could be written as:

```
prcnt = > 0 or < 100
```

3.5 Sign Conditions

⚠ It is also possible to test the sign of an arithmetic quantity, using the conditions **positive**, **negative** and **zero**. For example:

```
denominator is not zero
```

is equivalent to:

```
denominator is not = 0
```

3.6 Class Conditions

In addition there are class conditions used to determine whether the contents of a field are numeric or alphabetic. For example:

```
if variable is numeric
```

There are four class conditions: **numeric**, **alphabetic**, **alphabetic-lower** (C85), and **alphabetic-upper** (C85).

3.7 case in Cobol85

The **case** structure of other languages is implemented in Cobol 85 in the form of **evaluate**.

```
evaluate num_r
    when 0 perform even_n
    when 1 perform odd_n
    when other perform error.
```

This is equivalent to the **go to depending** of Cobol74.

```
odd-even.
    go to even_n odd_n
        depending on num_r.
    go to error-code.
even-n.
    ...
    go to end-dec-pnt.
odd-n.
    ...
end-dec-pnt.
```

4. Modular Programs and Subprograms

A procedure division paragraph is a subprogram , but paragraph oriented program design is not the only way to achieve modularity. Subprograms can be classified as internal (part of the main program) or external (coded and compiled independently).

There are three types of sub-program:

1. Internal subprograms as paragraphs or sections.
2. Internal subprograms as nested programs (C85).
3. External subprograms.

4.1 Internal procedures

Paragraphs are subprograms designed to execute specific tasks. The **perform** statement is used to invoke a paragraph. The **perform** statement transfers control. For example:

```
perform get-inputs.
perform do-calculations.
perform put-input.

get-inputs.
    ....

do-calculations.
    ....
```

This could have also been coded as:

```
perform get-inputs through do-calculations.
```

or

```
perform get-inputs thru do-calculations.
```

5. Loops

As mentioned in the previous section, paragraphs and sections found in the **procedure division**, are in fact modules designed to execute specific tasks.

5.1 Perform-until loops

The structure used to produce loops in Cobol is called the **perform** statement, and is of the form:

```
perform p-1 [through p-2] until condition.
```

The **condition** is a simple and compound condition and **paragraph-name** is the name of the paragraph that contains the statement sequence to be repeated. This is almost like calling a function numerous times. An example of using this is:

```
move 1 to n.
perform the-loop
    until n is greater than 100.
...

the-loop.
    ...
    add 1 to n.
```

This is a pre-test iterative structure - *somewhat equivalent to a do-while* loop.

5.2 Perform-times loops

There is a version of the **perform** loop used to iterate through the perform statement a prescribed number of times.

```
perform p-1 [through p-2] value times.
```

For example:

```
perform do-calculation 10 times.
```

This loop is *somewhat equivalent to a for* loop.

5.3 Perform-varying-until loops

There is also a more traditional form of the **perform** statement. It is of the form:

```
perform p-1 [through p-2]
    varying index from initial by increment
    until condition.
```

where **index** is a numeric variable that can assume integer values only, and **initial**/**increment** must be integer-valued numeric constants or numeric variables. For example:

```
move 0 to sum.
perform loop-1
    varying i from 1 by 1
    until i is greater than 10.
loop-1.
    add i to sum.
```

This loop sums the values from 1 to 10, and is equivalent to:

```
move 0 to sum.
move 1 to i.
perform loop-1
    until i is greater than 10.
loop-1.
    add i to sum.
    add 1 to i.
```

Therefore the **varying** clause eliminates the need for one **move** and one **add** statement.

5.4 Ending loops

Similarly to the **if** statement, loops in Cobol85 can be terminated with an **end-perform** statement.

Cobol 74	**Cobol 85**
move 0 to sum.	move 0 to sum.
move 1 to i.	move 1 to i.
perform loop-1	**perform** loop-1
until i is greater than 10.	until i is greater than 10
loop-1.	**end-perform.**
add i to sum.	loop-1.
add 1 to i.	add i to sum.
	add 1 to i.

5.5 Nested loops

Nested loops and 2D arrays always go hand in hand, but in Cobol, they don't appear visually as they would in other programming languages. A nested loop might look something like:

```
perform outer-loop
    until condition1.

outer-loop.
    * some code here
    perform inner-loop
        until condition2.
   * some code here

inner-loop.
   * some code here
```

For example:

```
perform outer-loop
    varying i from 1 by 1
    until i is greater than 5.

outer-loop.
    * some code here
    perform inner-loop
        varying j from 1 by 1
        until j is greater than 8.
    * some code here

inner-loop.
    * some code here
```

This code performs the outer loop 5 times, and the inner loop 8 times, for a total iteration of 40 - for every one time it performs the outer loop, it performs the inner loop 8 times.

⚠ This is not the conventional way of thinking about loops, so derive you algorithm carefully.

6. Program Blocks

Cobol85 introduced the notion of block structuring, accomplished using in-line forms of **perform**. They exist in the form:

```
perform
    * statements
end-perform.
```

For example:

Cobol 74	**Cobol 85**
move 0 to sum.	move 0 to sum.
move 1 to i.	move 1 to i.
perform loop-1	**perform** until i > 10
until i is greater than 10.	add i to sum
	add 1 to i
loop-1.	**end-perform.**
add i to sum.	
add 1 to i.	

7. Unconditional Control Transfer

The ubiquitous **go to** statement directs flow unconditionally to a paragraph or section. For example:

```
go to get-inputs.
```

⚠ The use of **go to** is not recommended - it leads to unreadable spaghetti code.

8. Terminating Programs and No-op Statements

A statement using the verb **stop** will cause the program to stop.

```
stop run       definitely stop the program
stop 25        stop the program temporarily
```

The following two statements are null statements that simply pass control to whatever physically or logically follows.

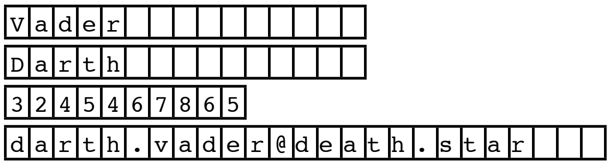
```
exit           Cobol74
continue       Cobol85
```

9. File I/O of Structures

Extending the file I/O example shown earlier to deal with structures is not that difficult. If we wish to read in a structure of the form:

```
     01  contact-data.
         05  last-name      pic x(15).
         05  first-name     pic x(15).
         05  phone-no       pic 9(10).
         05  email          pic x(25).
```

From a file called info.dat in which the data is arranged in this manner:



represented as sequential data in the file:

```
Vader          Darth          3245467865darth.vader@death.star
Skywalker      Luke           9238474775luke@rebellion.all
Jade           Mara           9983636545emperorshand@empire.net
Kenobi         Obi-Wan        7353636636obi-wan@jedimasters.net
```

Then the program which follows reads in each record, prints out the number of records, and the email address of each record. Here is a sample run of the program:

```
     Email is darth.vader@death.star
     Email is luke@rebellion.all
     Email is emperorshand@empire.net
     Email is obi-wan@jedimasters.net
     No. of Contacts = 004
```

Note that none of the records are stored internal to the program. They are merely processed.

```
identification division.
program-id. addnum.

environment division.
input-output section.
file-control.
select ifile assign to "info.dat"
    organization is line sequential.

data division.
file section.
fd ifile.
01 contact-data.
   05  last-name  pic x(15).
   05  first-name pic x(15).
   05  phone-no   pic 9(10).
   05  email      pic x(25).

working-storage section.
77 eof-switch    pic  9 value 1.
77 numcont       pic  999.
01 out-record.
   05  out1      pic x(8) value "Email is".
   05  filler    pic x.
   05  emailo    pic x(25).

procedure division.
    open input ifile.
    move 0 to numcont.
    perform sum-contacts
        until eof-switch = 0.
    close ifile.
    display "No. of Contacts = " numcont.
stop run.

sum-contacts.
    read ifile into contact-data
        at end move zero to eof-switch
    end-read.
    if eof-switch is not equal to zero
        add 1 to numcont
        move email to emailo
        display out-record
    end-if.
```