Advanced COBOL

"the use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."

Edsger Dijkstra

This unit will cover the following topics

1. Arrays
2. Strings
3. Cobol versus C
4. Two worlds collide: Fortran and Cobol
5. Cobol-74 to Cobol-85
6. Re-engineering Cobol
7. Flowcharts

This course uses OpenCobol, for which an extremely good guide is available online.

1. Arrays

1.1 Making simple arrays

Cobol sometimes uses the term *subscripting* to describe arrays to store **tables** of data. This is achieved using the clause **occurs**. For example a one-dimensional "table"is of the form:

```
01 word.
    05 letter pic X(1) occurs 15 times.
```

This creates a tables of 15 entries, each of which is a 1-character string. Another example involving a numeric table:

```
01 alphabet-freq.
    05 freq pic 9999 occurs 26 times.
```

This creates a table with 26 numeric entries used to store the frequencies of the letters of the alphabet. So **freq(1)** refers to the frequency of 'a', **freq(2)** the frequency of 'b' etc. To increment the value of **freq(1)** can be achieved in the following manner:

```
add 1 to freq(1).
```

In Cobol, array elements may be atomic or composite. An array with atomic element values consists of an elementary data item value in each element. It must have a **pic** clause associated with its name, before or after the **occurs** clause. So the first example could be:

```
01 word.
    05 letter occurs 15 times pic X(1).
```

An array with composite elements must have an **occurs** clause together with the array name, followed by a substructure specification. For example:

```
01 name-list.
    05 st-name occurs 100 times.
        10 last-name pic x(10).
        10 first-name pic x(10).
```

1.2 2D arrays

Now a two-dimensional "table" would is of the form:

```
01 matrix.
    03 row occurs 8 times.
        05 element pic 9999 occurs 12 times.
```

Then **element(7,9)** refers to the 9th element of the 7th row. It is also possible to refer to **row(7)**, which refers to the set of numbers stored in row 7.

⚠ Cobol 74 was limited to 3 array dimensions. Cobol85 has extended this to 7.

Some rules:

- Arrays should be defined within a superordinate structure at levels other than 01.
- Array size should be specified using the **occurs** clause.

1.3 Assigning values to array elements

There are four ways to assign values to array elements:

1. Run-time individual element assignment using **move** statements.
2. Run-time global value assignment using **move** statements.
3. Compile-time value assignment using **value** clauses.
4. Compile-time value assignment using **redefines** clauses.

For example, individual elements of an array can be assigned values using the **move** statement:

```
move 0 to element(7,9).
```

All elements can be initialized using the *table name*:

```
move 0 to matrix.
```

Initialization can also be accomplished at compile-time using the **value** clause:

```
01 matrix.
    03 row          occurs 8 times.
        05 element occurs 12 times pic 9999 value zero.
```

If elements are to be initialized to different values at compile-time, a template structure and **redefines** clause must be used:

```
01 days-of-week.
    05 filler value "montuewedthufrisatsun" pic x(21).

01 days-table redefines days-of-week.
    05 week-day occurs 7 times pic xxx.
```

1.4 Reading and writing arrays

A value can be read into an array from standard input using the **accept** statement. For example:

```
    accept element(7,9) from console.
```

The display statement can be used to output to standard output:

```
    display element(7,9).
```

If an array is defined within a record structure of a file it can be input or output using file-oriented I/O.

Imagine a structure of the form:

```
01 student-info.
   05  student-name occurs 4 times.
       10  stdnt-name  pic x(15).
       10  stdnt-idno  pic x(7).
```

Below is a program to read 4 such records from a file named **student.dat**, store the records in an array-structure, and print out the student names from the array. This is a sample of the output:

```
Student name is Skywalker
Student name is Ackbar
Student name is Chewbacca
Student name is Palpatine
```

from this input in the file **student.dat**:

```
Skywalker      6543287Ackbar         1189283Chewbacca      9882870Palpatine
0000001
```

The program is below (each section is colour-coded):

```
identification division.
program-id. fileio_arrays.

environment division.
input-output section.
file-control.
select ifile assign to "student.dat".

data division.
```

```
file section.
fd ifile
    record contains 88 characters.
* Specify the structure of the records in the file
* In this case there are 4 records, each containing two elements
01 student-info.
   05  student-name occurs 4 times.
       10  stdnt-name  pic x(15).
       10  stdnt-idno  pic x(7).

working-storage section.
* Define the loop index
01  i  pic  9.

procedure division.

* Open the file, read in the data into the
* structure and close the file
    open input ifile.
    read ifile
    end-read.
    close ifile.
    move 1 to i.
* Loop four times
    perform print-out
        until i is greater than 4.
stop run.

print-out.
    display "Student name is " stdnt-name(i).
    add 1 to i.
```

The latter part of this code could also be modified to remove the module **print-out**, and use a block construct instead.

```
    perform until i > 4
        display "Student name is " stdnt-name(i)
        add 1 to i
    end-perform.
```

2. Strings

Cobol is a powerful string processing language.

2.1 Making simple strings

Strings can be created as fixed-length words. For example:

```
01 string-1 pic x(30).
```

Strings can also be created using arrays of the form:

```
01 string-array.
    05 str1 pic X occurs 40 times.
```

This creates a tables of 40 entries, each of which is a 1-character string. This can have a value assigned to it in the following manner:

```
move “darth vader, sith lord” to string-array.
```

Which in the array would look like:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
d	a	r	t	h		v	a	d	e	r	,		s	i	t	h		l	o	r	d																		

2.2 String processing - string

The **string** statement extracts a substring from a string, concatenates two or more substrings, and computes the length of a concatenated string.

Consider the following example:

```
01 str1          pic x(12).
01 str2          pic x(6).
01 str3          pic x(7).
01 str4          pic x(10).
01 result-str    pic x(40).
01 ptr-x         pic 99.

move “1950s atomic” to str1.
move “atomic” to str2.
move “coffee “ to str3.
move “machine   “ to str4.
```

```
     move 1 to ptr-x.

     string str1 delimited by str2
            str3 delimited by size
            str4 delimited by space
       into result-str
       with pointer ptr-x
       on overflow display “overflow!”.
```

The concatenated value in **result-str** is:

```
     “1950s coffee machine^^^^^^^^^^^^^^^^^^^^”
```

The variable **ptr-x** contains the value 21, the length of the string is 20.

If a string is used as a delimiter, the source string is transferred until the delimiter string is encountered. If **size** is used, the entire string is transferred, if **space**, then the source string is transferred up until a space is encountered.

Here is a second example:

```
     01 str1          pic x(5).
     01 str2          pic x(5).
     01 str3          pic x(5).
     01 str4          pic x(5).
     01 result-str    pic x(15).
     01 ptr-x         pic 99.

     string str1
            str3 delimited by space
            str4 delimited by size
       into result-str
       with pointer ptr-x.
```

str1	A	B	C	^	^
str2					
str3	D	E	F	G	^
str4	H	I	^	^	^

A	B	C	D	E	F	G	H	I	^	^	^			

Here is a third example:

```
string str1 delimited by str2.
       str3 delimited by size
       str4 delimited by ","
  into result-str
  with pointer ptr-x.
```

str1	A	B	C	D	E
str2	B	^	^	^	^
str3	F	G	H	^	^
str4	I	J	K	,	L

A	B	C	D	E	F	G	H	^	^	I	J	K		

In the phrase "str1 delimited by str2", the string in str2, "B^^^^", is not a substring contained in str1, hence str2 is ignored as a delimiter.

2.3 String processing - unstring

The **unstring** statement extracts one or more substrings from a string, and computes the length of a extracted substrings string. It does the opposite of **string**.

Consider the following example:

```
01 str1         pic x(40).
01 str2         pic x(12).
01 str3         pic x(12).
01 str4         pic x(12).
01 str5         pic x(12).
01 del2         pic x(12).
01 del3         pic x(3).
01 del4         pic x(3).
01 del5         pic x(3).
01 count2       pic 99.
01 count3       pic 99.
01 count4       pic 99.
01 count5       pic 99.
01 ptr-x        pic 99.
01 tall-x       pic 99.
```

```
     unstring str1
            delimited by all spaces or ":" or "."
       into str2 delimiter in del2 count in count2
       into str3 delimiter in del3 count in count3
       into str4 delimiter in del4 count in count4
       into str5 delimiter in del5 count in count5
       with pointer ptr-x
       tallying in tall-x.
```

```
str1 : "PROGRAMMING LANGUAGES:A PERSPECTIVE^^^^^"

str2 : "PROGRAMMING^"  del2 : "^^^"  count2 : 11
str3 : "LANGUAGES^^^"  del3 : ":^^"  count3 : 09
str4 : "A^^^^^^^^^^^"  del4 : "^^^"  count4 : 01
str5 : "PERSPECTIVE^"  del5 : "^^^"  count5 : 11

NOTE: ^ represents a blank.
```

2.4 String processing - inspect

There are four inspect statements:

- **inspect tallying** verifies the existence of a substring in a string, and counts the number of occurrences of a substring in a string.
- **inspect replacing** statement replaces an existing substring in a string with another substring.
- **inspect tallying/replacing** is a combination of tallying and replacing.
- **inspect converting** statement replaces one or more substrings in a string with other substrings.

The examples below do not show all the possibilities, for that it is suggested to refer to a Cobol reference manual.

Consider the following examples for **inspect tallying**:

```
     01 str1          pic x(15) value spaces.
     01 tall-x        pic 99    value zero.
     inspect str1
            tallying
            tall-x for all “E” before initial “C”.
```

str1

M	O	N	T	R	E	A	L		Q	U	E	B	E	C

tall-x

0	3

```
     inspect str1
            tallying
            tall-x for all “NG” after initial “K”.
```

str1

K	I	N	G		K	O	N	G						

tall-x

0	2

```
     inspect str1
            tallying
            tall-x for characters before initial “ ”.
```

str1

M	O	N	T		R	O	Y	A	L					

tall-x

0	4

```
     inspect str1
            tallying
            tall-x for characters.
```

str1

M	O	N	T		R	O	Y	A	L					

tall-x

1	5

```
     inspect str1
            tallying
            tall-x for leading "0" before initial "2".
```

str1

0	0	0	0	0	1	1	2	4	5	2	8			

tall-x

0	5

Consider the following examples for **inspect replacing**:

```
     01 str1         pic x(15) value spaces.

     inspect str1
            replacing
            characters by "*" after initial " ".
```

str1-before

M	O	N	T		R	O	Y	A	L					

str1-after

M	O	N	T		*	*	*	*	*	*	*	*	*	*

```
     inspect str1
            replacing
            all "A" by "I" before initial "V".
```

str1-before

D	A	R	T	H		V	A	D	E	R				

str1-after

D	**I**	R	T	H		V	A	D	E	R				

```
     inspect str1
            replacing
            all "A" by "I".
```

str1-before

D	A	R	T	H		V	A	D	E	R				

str1-after

D	**I**	R	T	H		V	**I**	D	E	R				

Consider the following examples for **converting**:

```
     01 str1         pic x(15) value spaces.

     inspect str1
            converting "AEIOU" to "12345".
```

str1-before

M	O	N	T		R	O	Y	A	L					

str1-after

M	**4**	N	T		R	**4**	Y	**1**	L					

which is equivalent to the following **inspect replacing**:

```
     inspect str1
            replacing
            all "A" by "1"
            all "E" by "2"
            all "I" by "3"
            all "O" by "4"
            all "U" by "5".
```

3. Cobol versus C

The sum 1 + 1/2 + 1/3 + 1/4 + 1/5 +... can grow beyond any limit if a sufficient number of terms if taken. Write a paragraph **summation** which increases the value of the sum by an arbitrary term 1/K. Then use this paragraph-name in a **perform**-statement entering a computation for deciding how many terms are necessary for making the sum greater than 10. The number of terms should be written out by a **display** statement.

```
summation.
    compute SUM-1 = SUM-1 + 1 / K.
    move K to R.
term-computation.
    move ZERO to SUM-1.
    perform summation varying K from 1 by 1
    until SUM-1 is greater than 10.
    display R.
```

Compare this against the equivalent C statements:

```
sum = 0;
K = 1;
while (sum <= 10.0)
{
    R = summation(&sum,K);
    K = K + 1;
}
printf("%d", R);

int summation(double *sum, double K)
{
    *sum = *sum + 1.0/K;
    return K;
}
```

You will notice in the Cobol program the lack of pass-by-reference, or indeed passing values all-together.

4. Two Worlds Collide: Fortran and Cobol

Some legacy code may involve language interactions, for example Fortran subprograms called from Cobol, or vice-versa. This theoretically enabled the best features of each language to be used, minimizing their disadvantages. With re-engineering there is the added disadvantage of having *two* legacy languages to deal with. In a paper from 1965[1], the author cites this as being advantageous because Cobol has "strong output and data manipulation characteristics", and "the capability of performing complex logical operations that are difficult to duplicate in Fortran IV". Cobol's Achilles heel is inputting large arrays, which Fortran is better able to deal with.

It is unlikely that this would be a problem in a more modern version of Cobol, or that Fortran couldn't handle the entire computational task. Refer to the paper for the program in question, which computes the value of a sum invested at a known rate of interest compounded periodically for five years. There is also the case of calling Cobol from Fortran[2].

[1] Shavell, Z.A., "The use of Fortran in subroutines with Cobol main programs", *Communications of the ACM*, Vol.8(4), pp.221-223 (1965).

[2] Tajiri, K., "The use of Cobol in subroutines with Fortran main programs", *Communications of the ACM*, Vol.8(4), pp.233 (1965).

5. Cobol 74 to Cobol 85

Some of the greatest changes came with Cobol85. Although we have discussed them throughout the units on Cobol, they are summarized below.

5.1 if

Cobol 74	Cobol 85

if d **is greater than** 0
    divide n by d giving r
else
    display ‘error’.
	if d **is greater than** 0
	then
	    divide n by d giving r
	else
	    display ‘error’
	end-if.

5.2 Pre-test loops

Cobol 74	Cobol 85

perform avg-temp
    until day is > 365.
	perform avg-temp
	    with test before
	    until day is > 365.

5.3 Post-test loops

Cobol 74	Cobol 85

perform avg-temp.
perform avg-temp
    until day is > 365.
	perform avg-temp
	    with test after
	    until day is > 365.

This is somewhat equivalent to a *do-until* or *do-while* loop.

5.4 Three-way selections

Cobol 74	Cobol 85

```
if num is < 0
    perform neg-nums.

if num is > 0
    perform pos-nums.

if num is = 0
    perform zero-nums.
```

```
if num is < 0
    perform neg-nums
end-if.
if num is > 0
    perform pos-nums
end-if.
if num is = 0
    perform zero-nums
end-if.
```

3.5 Nested ifs

Cobol 74

```
if num is < 0
    perform neg-nums
else
    if num is > 0
        perform pos-nums
    else
        perform zero-nums.
```

Cobol 85

```
if num is < 0
then
    perform neg-nums
else
    if num is > 0
    then
        perform pos-nums
    else
        perform zero-nums
    end-if
end-if.
```

6. Re-engineering Cobol

The challenging part of re-engineering Cobol programs is to identify structures in need of being "updated", or removed in the case of redundant features. This section deals with some of these issues, although it by no means covers everything.

6.1 Add end-if

All conditional statements which still use the **next** clause, or implicit terminators like a period should be transformed so that they are terminated by the explicit terminator **end-if**. This makes the code more consistent and removes redundant instructions (e.g. **next**). For example:

```
para-1.
    if x > 0
        go para-2
    else
        next sentence.
    display 'x'.
para-2.
```

```
para-1.
    if x > 0
        go para-2
    end-if.
    display 'x'.
para-2.
```

6.2 Eliminate go to

Eliminate jump instructions to reduce the amount of unstructured spaghetti-code. For example:

```
para-1.
    if x > 0
        go para-2
    end-if.
    display 'x'.
para-2.
```

```
para-1.
    if x > 0
        continue
    else
        display 'x'
    end-if.
para-2.
```

Note that the role of the **continue** statement is to prevent empty portions of the if statement. This can be taken a step further in the next section, which removes code containing continue statements that are not necessary.

6.3 Eliminate unnecessary continue

Eliminate continue instructions that are deemed not necessary. For example:

```
para-1.
    if x > 0
        continue
    else
        display ‘x’
    end-if.
para-2.
```

```
para-1.
    if not x > 0
        display ‘x’
    end-if.
para-2.
```

Sometimes it is easier to add a **continue** statement during earlier processing, and remove the redundant ones later as a post-processing step.

6.4 Condition normalization

In 3.3, removing the **continue** caused the condition to be changed.

```
para-1.
    if x > 0
        continue
    else
        display ‘x’
    end-if.
para-2.
```

```
para-1.
    if not x > 0
        display ‘x’
    end-if.
para-2.
```

6.5 Restructuring while

In dialects of Cobol 74 there was no **while** construct available, so it was often simulated using a **go to**. The two examples below show a while loop type construct in Cobol 74 and Cobol 85.

```
Cobol 74

loop-one.
    if expr
        ...
        go to loop-one
    end-if.
```

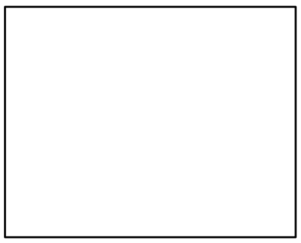
```
Cobol 85

loop-one.
    perform until not expr
        ...
    end-perform.
```

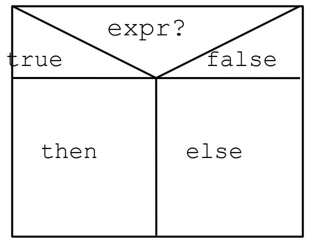
7. Flowcharts

It seems as though the concept of flowcharting isn't talked about much anymore. With the widespread adoption of ALGOL-like languages, pseudocode is now used more to represent algorithms. Flowcharts were an ever-present feature of Cobol programs. Apart from classic flowcharts, there are a number of variations: Nassi-Shneiderman diagrams, Ferstl diagrams and Hamilton-Zeldin diagrams. It is good to know a little about them, because they might help decipher a Cobol program.

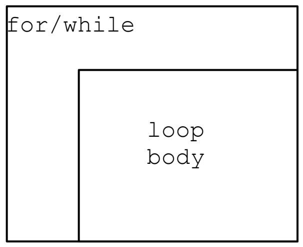
Nassi-Shneiderman diagrams (NSD), sometimes referred to as Chapin charts, were introduced in the 1973[3] as a way of visually describing an algorithm prior to coding. Their idea was to instigate a means of modeling computation in simply ordered structures. These diagrams employ rectangular symbols for each of the structured primitives, and their derivatives. The symbols are combined by placing them adjacent to each other. The absence of connecting lines protects the structure from being compromised by meandering branches. It consists of the following basic symbols:



A *process* symbol used for assignment, I/O, function calls and groups of statements

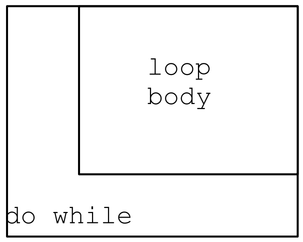


A *decision* symbol. The central triangle contains the expression to be evaluated. The left and right triangles contain possible outcomes.



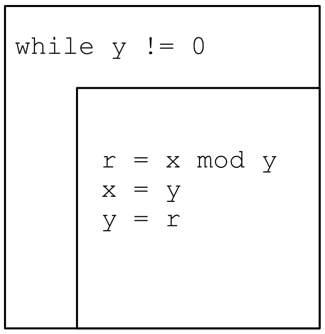
An *iteration* symbol. The inverted "L" shape allows for a path to follow once the iteration is complete. These loops may occur 0 or more times.

[3] Nassi, I., Shneiderman, B., "Flowchart Techniques for Structured Programming", *ACM SIGPLAN Notices*, Vol.8, No.8, pp.12-26 (1973).



An *iteration* symbol.These loops occur at least once.

By combining the basic structures, all of which are rectangular, we can design a structured program. If the NSD gets too complex any rectangular symbol can be replaced by a process symbol, and made into a separate diagram. The NSD promote modularization. For example, a NSD of Euclids iterative GCD algorithm:



```
while y != 0
    r = x mod y
    x = y
    y = r
```

NSD is a means of logic representation. The reference given in the unit outlines for Cobol:

Uckan, Y., *Application Programming in COBOL: Concepts, Techniques and Applications (Vol.1)*, D.C.Heath & Company, 1992.

gives a good introduction to a number of techniques including some for structure representation:

- Action diagrams - high-level structural overview.
- Decision tables and decision trees.
- Structure charts.
- Jackson diagrams.
- Warnier-Orr diagrams.