FORTRAN

and writing the program

Preparation for writing a program

1. Look at the output required.
2. Look at the input.
3. Assign variables and determine calculations to be made.
4. Do checksum, determining output field sizes.
5. Make a printer spacing chart.
6. Make a flowchart.
7. Write the code.

Logon

If **CTC** appears, press `CLEAR`, type `VMCTC`, press `ENTER`. Logon: `CMS568`.

Start **XEDIT** by typing `XEDIT` *`FILENAME`* `ANY A` and create some blank lines by typing `55A` in the prefix area at the end of the top line.

Next goes the initial **job control language**.

```
//FOR2TP?? JOB CECSAA55,'TOM PENICK'
//STEP1 EXEC WFCTC
//FORT.SYSIN DD *
$JOB WATFOR
```

Replace the ?? with the program number.

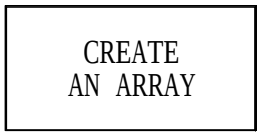
Next goes the **fortran source program** beginning with the following heading. The dashes run from space 10 to space 31. The C's are in column 1 and define the line as a comment, not to be read by the compiler.

```
C     ----------------------
C     -                    -
C     -   PROGRAM # ?      -
C     -   COSC 1403.CON    -
C     -   TOM PENICK       -
C     -                    -
C     ----------------------
```

Following this is:

Description of what the program does
Input record description
Dictionary of variable names.

ARRAYS



If the program has an array(s) it must be created next. The following statement creates a single-dimensional and a two-dimensional array.

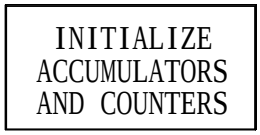
```
DIMENSION NRAY(8), IRAY(10,12)
```

For the array "IRAY", 10 is the number of rows, and 12 is the number of columns. (The space in front of IRAY is unnecessary.) These numbers are called the **subscript**.

Subscripts may be written in the following integer forms:

```
constant    variable + constant     constant * variable + constant
variable    variable - constant     constant * variable - constant
            constant * variable
```

Set **Accumulators and Counters** to zero or one. Variables must have a value before they can be used.



The WRITE statement


```
12 WRITE(6,111)VAR1,VAR2,IVAR3
```

The word *WRITE* begins in column 7. The number 12 in this case is an optional **heading** and ends in column 5. The number 6 refers to the printer. The number 111 refers to the heading of the corresponding FORMAT statement. If there are variables to be written, they follow the parenthesis. Our printer can print 132 spaces on the line.

The FORMAT statement

```
111 FORMAT(1H1,100X,'TOM PENICK'/1H0,8X,2(F5.2,8X),I4)
```

The 111 ends in column 5 and is a **heading** which identifies the format statement with an appropriate write or read statement. Columns 1-5 may be used for the heading. FORMAT begins in column 7. The FORMAT statement must not extend past space number 72, but may be **carried over** to the next line (up to 5 lines total) by placing any character (except zero) in space 6 of the line of continuation. Columns 73-80 are reserved for sequence numbers (from the days of punched cards) and we do not use them.

If the format statement is associated with a WRITE statement, the first statement after the open parenthesis must be carriage control. If the format statement is associated with a READ statement, there is no carriage control. Statements are separated by commas. The statement 100X means to skip 100 spaces.

Characters and spaces enclosed in **single quotes** are printed as is. If it is necessary to print a single quote, then an alternate method is used. For example to instruct the printer to print `DAN'S GRILL` the format statement would include `,11HDAN'S GRILL,` where `11H` means to print the next 11 characters as is. This is called a **holorith**.

A forward slash preceeds each **carriage control statement** except the initial one and takes the place of a comma. The statement `,2(F5.2,8X),` means to perform the statements within the parenthesis 2 times.

`F5.2` is a **field descriptor** that means to print the **real number variable** given in the WRITE statement with two decimal places allowing for a field of 5 spaces counting the decimal. There must be at least one digit on each side of the decimal, so an `F2.2` would not be valid. `I4` is another **field descriptor** that means to print the **integer variable** given in the WRITE statement in a field of 4 spaces. If there is an insufficient number of field descriptors in the FORMAT statement to handle the number of variables given in the WRITE or READ statement, the computer will return to the beginning of the FORMAT statement and start using them over.

Carriage control

`1H+` - 0 lines (type over)
`1H` - 1 line (single space)
`1H0` - 2 lines (double space)
`1H-` - 3 lines
`1H1` - top of next page

The READ statement


```
11 READ(5,112,END=900)VAR1,VAR2
```

The number 11 ends in column 5 and is a **heading** which serves as a locator for this statement. Columns 1-5 may be used for the heading. READ begins in column 7. The READ statement must not extend past space number 72, but may be **carried over** to the next line (up to 5 lines total) by placing any character (except zero) in space 6 of the line of continuation. Columns 73-80 are reserved for sequence numbers (from the days of punched cards) and we do not use them.

The number 5 refers to the source of the data to be read (the program is the source in this case). The number 112 identifies the format statement to which this statement relates. `END=900` instructs the program to go to heading 900 in the event all records have been read. `VAR1,VAR2` are real number variables to be given the values read. The format statement that goes with the read statement will not have carriage control. It will identify the location and field sizes of the data to be read into the variable names. Real number data may be **written without decimals**. For example, if the data consists of 4-digit numbers without decimals to be read in as real numbers with the last two digits following a decimal, the format statement would describe them as F4.2. If the same numbers were to be printed using a write statement, its corresponding format statement would describe them as F5.2. If there is already a decimal in the data that is read, then that decimal will be used rather than the one specified in the field descriptor.

The Preread statement is used to read a record before reading the rest of the data. This example reads in the date and then writes it:

```
    READ(5,50)MO,IDA,IYR
 50 FORMAT(3I2)
    WRITE(6,100)MO,IDA,IYR
100 FORMAT(1H1,10X,I2,'/',I2,'/',I2)
```

The date would have to be the first line of data following the `$ENTRY` , with the rest of the data beginning on the second line.

Reading text. A read statemtnt can be used to load text into memory. Each 32-bit memory location can store up to 4 letters in EBCDIC code.

```
    READ(5,100)N
100 FORMAT(1A4)  . . . . . .  Read one 4-letter word.

    DIMENSION N(6) . . . . . .  Create an array "N" of 6 memory locations.
    READ(5,200)(N(I),I=1,6)
200 FORMAT(6A4)  . . . . . .  Read six 4-letter words.
```

The DATA statement

is used to load multiple values into multiple variables.

```
DATA A,B,C,N/0.0,2.0,3.3,4/
```

In this statement, the four variables preceeding the first / are loaded with the four values following. In the statement below, the array, `IRAY(5,5)` is filled with zeros:

```
DATA IRAY/25*0/
```

The variables and arrays ahead of the / are filled with whatever values and multiple values are listed to the right of the first / in the order that they appear. The / at the end is mandatory.

A data statement can be used to load **text** into memory (stored in EBCDIC code), with a maximum of 4 letters per 32-bit memory location.

```
DATA N/'FITZ'/
```

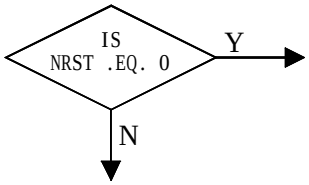
The GO TO statement

The **Computed GO TO** is typically used for menus or when reading records of different types that require processing in different ways.

```
NUM=3
GO TO(50,60,80,20,20,20),NUM
```

This statement results in a "`GO TO 80`" because 80 is the third value in the list and NUM=3. This statement could fall through if NUM was equal to 0 or 7, for example.

The Logical IF statement




```
IF(NRST .EQ. 0)GO TO 14
```

NRST is a variable. .EQ. is a **relational operator**. There must be a space before and after it. `GO TO 14` is the instruction to be used if the the preceeding statement is true, otherwise the program proceeds to the next statement. You cannot compare real and integer values in an IF statement; they will never be equal. You can have only one "then" statement within the IF statement. If more than one command is to be executed following a true IF statement, then either use multiple IF statements or a GO TO statement.

Relational Operators

- `.EQ.` - equals
- `.NE.` - not equal to
- `.GT.` - greater than
- `.GE.` - greater than or equal to
- `.LT.` - less than
- `.LE.` - less than or equal to

The relational operators may be used in the same IF statement along with the following logical operators.

Logical Operators

- `.AND.` - and
- `.OR.` - or
- `.NOT.` - not (we are not to use "not")

The Arithmetic IF - Whatever is in the parenthesis is solved for sign. This also could simply be a variable. The sign of the value in parenthesis is used to determine which label to GO TO. If it's negative it goes to the first; if it's zero it goes to the second; and if it's positive it goes to the third. Two of the labels could be the same.

```
IF(B-A)10,20,30
IF(NRST)20,30,30
```

The statement cannot fall through, it must go to one of the three labels.

The CONTINUE statement

```
13 CONTINUE
```

The continue statement does nothing. It's just a place to put a heading to be used with a GO TO statement.

Variables

Variable names may contain up to 6 letters and numbers and must begin with a letter. Integer variables begin with letters I, J, K, L, M, or N. A variable must have an assigned value before it can be used in a calculation, hence counters and accumulators are set to 0 (or some other number) before they are used.

Arithmetic statements

```
VAR1 = VAR1 + VAR2 - 2.57
```

= is a **replacement operator**. The value obtained to the right of the = is assigned to the variable on the left. There may be only one variable on the left.

Math Hierarchy:

( )	parenthesis
**3	exponent (cubed in this case)
*,/	multiplication, division
+,-	addition, subtraction

The float. If an integer variable is used in a calculation with a real number variable, it may be treated as a real number by **float**.

```
VAR1 = VAR1 * FLOAT(IVAR4)
```

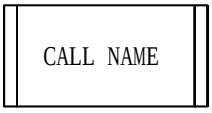
Mixed mode arithmetic. If integer and real number variables are used in an arithmetic statement without float, the math will be done in real mode. When a real value is loaded into an integer variable, any value to the right of the decimal is discarded, not rounded.

Subroutines

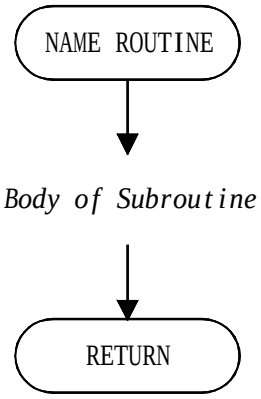
A subroutine is a **separate** program that does not share variables or headings. The subroutine is named using the same conventions as for variable names, except that the first letter does not denote real/integer. Variable **values** are passed from program to subroutine and back to program by placing them in the **call argument**. The order of the variables is significant; it does not matter whether or not the names match. If an **array** is passed, it must be dimensioned at the beginning of the subroutine, just as it was at the beginning of the program.

```
      CALL HEAD(MO,IDA,IYR)

           [other program code may go here]

      STOP
      END
      SUBROUTINE HEAD(MO,IDAY,IYR)
      WRITE(6,100)
  100 FORMAT(1H1,100X,I2,'/',I2,'/',I2)
      RETURN
      END
```

A subroutine can be called from multiple places within the program and will return to the line following the line from which it was called. A subroutine can call other subroutines but cannot call itself. The subroutine uses a separate flowchart. The flowchart symbol for call subroutine looks like this:



The subroutine flowchart looks like this:



where "NAME" is the name of the subroutine

The DO LOOP

Do everything beginning with the next line to label 5 (use a continue statement) a total of 30 times:

```
DO 5 NR=1,30
```

5 is a **statement label** specifying the **range** of the Do Loop. NR is an **index variable.** NR assumes the **starting value** of 1 and increments by the assumed **increment value** of 1 each time the statements following are executed, ending with the **end value** of 30. At this point NR increments to 31 and the statement carries through. The value of the index variable is unpredictable following the execution of the Do Loop. If this value is to be used later in the program, it should be stored to another variable before leaving the loop. The starting value, end value, and increment value may be replaced by variables but must be positive integers. To increment by some number other than 1, place the increment value following the end value:

```
DO 5 NR=1,30,2
```

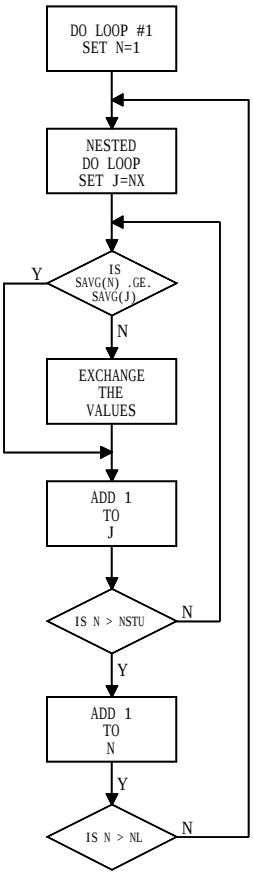
An **Implied Do Loop** may be used with a READ or WRITE statement.

```
    READ(5,100)(ARAY(I),I=1,5)
100 FORMAT(5F3.1)
```

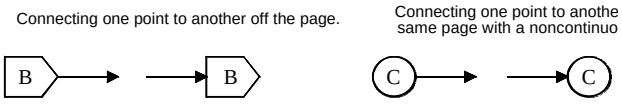
A Do Loop occuring within another Do Loop is a **Nested Do Loop**. The following example of a Nested Do Loop is called an **Exchange Sort** and sorts the averages of students in descending order. The averages are in an array called SAVG.

The process begins by the Nested Do Loop comparing the first record with the second. If the second is larger, then the position of the two records is switched. Then the first record is compared with the third record, then with the fourth, and so on until the last record is reached. The largest record is now in the first position. The first Do Loop switches us to the second record and the Nested Do Loop executes the process of comparing the second record with the third, then the fourth, and so on until the last record has been reached. Now the second largest record is in the second position. The first Do Loop switches us to the third record and the Nested Do Loop compares it with the remaining records, etc., etc.

```
      NL = NSTU - 1    . . . . . . . . . . . NL stands for next-to-last
      DO 91 N=1,NL     . . . . . . . . . . . First Do Loop begins
      NX = N + 1       . . . . . . . . . . . NX stands for next
      DO 92 J=NX,NSTU  . . . . . . . . . . . Nested Do Loop begins
      IF(SAVG(N) .GE. SAVG(J))GO TO 92   . Two averages are compared
      HOLD    = SAVG(J)  . . . . . . . . . . SAVG(J) is placed in temp. mem. location
      SAVG(J) = SAVG(N)  . . . . . . . . . . SAVG(J) is replaced by SAVG(N)
      SAVG(N) = HOLD     . . . . . . . . . . The swap is complete.
   92 CONTINUE     . . . . . . . . . . . . . End of the Nested Do Loop
   91 CONTINUE     . . . . . . . . . . . . . End of the first Do Loop
```



Other **Flowchart Symbols**:



Errors

Syntax error example:

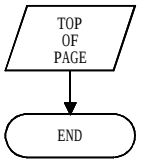
```
100 FOMAT
```

Runtime error example:

```
IVAR = 3.1
```

The program will stop when it encounters a runtime error. If you get UUUUU in your output, then the variable was empty. If you get ******, then the field was too small. The number of U's or *'s will be equal to the number of spaces allocated by the field descriptor.

Ending the program



A program may end with the instruction for the printer to go to the top of the next page:

```
    WRITE(6,199)
199 FORMAT(1H1)
    STOP            [stops the execution of the program]
    END             [physical end of the program; stops the compiler]
```

Following the end of the program is the JCL statement:

```
$ENTRY
```

Next comes the input records (data). To get the **data**, type on the command line:

```
GET SP95FOR? DATA C
```

where ? is the program number. After the data comes:

```
/*
//
```

That's it.

Proofreading

Check for appropriate use of headings; no two format statements have the same heading and write/read statements have a matching format statement.

Is the read statement set to read the correct spaces?

Do the format statements have the proper commas and slashes?

Do the variable names match the fields in type and quantity?

Printing

```
PRTMBR FILENAME ANY A
```

Get next assignment

```
PRTMBR FOR?SP95 LISTING C
```

(Replace ? with the assignment number.)

Run the program

```
JOB FILENAME ANY A
```

Label the program

TOM PENICK
COSC 1403
day & time
PROGRAM # __
RUNS USED: __

MISCELLANEOUS INFORMATION

Machine Language is the lowest level language. It is long, in binary, and machine specific; built in to the computer.

Assembly Language is easier to handle. It is a symblic representation of machine language.

Core Memory is magnetic, polarized one way or the other. Holds information with power off.

Volatile Memory Information is lost when power is removed.

ROM - Read Only Memory is permanent memory. It instructs the computer to load the operating system.

PROM - Programmable Read Only Memory
EPROM - Erasable Programmable Read Only Memory

The **Central Processing Unit (CPU)** of the computer is composed of three sections:

1. Arithmetic Logic Unit
2. Control Unit
   a. Program Counter (Pointer)
   b. Instruction Register
3. Main Memory Unit

External to the CPU are storage, and input/output devices.

High Level Languages such as Fortran, Cobol, C, Basic require a compiler.

Source Code is the language in which a program is written.

Object Code is the machine language.

Translators convert high level languages to machine language. Programs that translate are called translators, not operating systems.

An **Assembler** converts a source program into an object program (ones & zeros).

A **compiler** converts high level language to a machine language program. It allows a program to be used on various machines provided you have the compiler for that machine.

An **interpreter** translates to machine code as it executes. It does not use a compiler. It is less efficient because translating takes more time than executing.

Operating System - a program that allows a computer to run a variety of other programs without manually resetting switches.

Fortran means Formula Translation. It was one of the first high level languages. It is good at handling numbers and weak at handling strings.