Modules, Flags and Priming Reads

MODULARITY

We have already seen the use of modules in our designs. Several of our designs have called a loop module as the only statement in our Mainline WHILE loop. And several have called a module to do a series of calculations within a loop. Those examples demonstrate several of the reasons for breaking our designs up into smaller modules.

In the "good old days" a program would very li9kely be designed in one long (hundreds, possibly thousands of lines) module. As you can imagine, there were some problems with that style of design. The module quickly became unwieldy to work with. Whenever a change was made to any part of it, the designer had to consider the impact on the rest of the design since there was no way to isolate any part of it from the rest. If several people were working together on the program, it was difficult to merge their contributions into the single module. And modifying the design/program later seemed next to impossible!

The solution that was derived was to split programs up into modules where each module handled a given function or task. We've already looked at beginning our designs by listing the main tasks. We just need to continue that approach by breaking our design into task-related modules. Frequently (but not always) one main task equals one module.

What are the advantages?

1. The design is much more "readable" and "maintainable."
2. Teamwork is much easier because team members can divide the design by modules.
3. Designs can be "built" by including completed modules from other designs.
4. A segment of code that is repeated at different points in a program can be separated into a module and simply "called" each time it is needed.

One key is in deciding how the modules should be defined. The easy answer is that each major task becomes a separate module (or module system including a module with sub-modules), but usually life isn't quite that simple. In fact, there is no easy answer, and there isn't a single answer. Our goal should be to define our modules so that all the statements within one module are very closely connected or related and so that modules are loosely connected. Usually we think of connecting the statements within a module by function. All the statements within a module are related to the same function or task, and, from a different perspective, all the statements related to that function are in the same module. That is, we don't scatter statements related to a single task over several modules.

If we were designing a program to create a report, we might have very specific instructions about how the report should look, for example, a one-inch margin at the top of the page, multi-line headings on each page, a page number printed in the top right corner, and so on. All of the statements that implement these page formatting instructions would be closely related to the same function, and we would probably put them together in the same module. If someone later needed to modify the design and program to change the page format, he or she could to directly tot eh single module that held all the related statements.

Sometimes, we connect the statements because they all occur at the same time within a design. It is common to have a module called prior to the loop to do all the one-time-only tasks needed to get the program going. Sometimes called HouseKeepting or StartUp, this module would include beginning tasks such as opening files and initializing counters. These tasks are not related to a single function, but are all performed once prior to the loop.

There are very few hard and fast rules for you to go by. Some suggest that a module shouldn't be more than 50 lines, but that number just comes from the number of lines that will fit neatly on a page. It is true that it is easier if the entire module is on one page, but it isn't a rule without exceptions.

There could reasonably be modules with only two or three lines in them. Generally, I would not recommend modules with just a few lines unless (1) the module performs a task that is so well-defined and unique that it makes sense to separate it (like a READ module as we will see below) or (2) you expect to have additional lines in future modifications that would go into the module.

The "connection" between modules will usually be the variables or values that are passed from one module to another. If we have module that calculates needed variables, we are probably assuming that those calculated values will be available to other modules. Different programming languages will control what values are passed from one module to another and how they are passed in different ways. We certainly don't want our design to get caught in the syntax intricacies of a particular language. But we do want to be aware that there is a flow of data among modules.

A module may *receive* a value from another module. A module may also *return* a value that has been input or modified. In some cases, a value will be both received and returned by a module. Some variables will be neither received nor returned; they are used only within the one module. Look at the process and Calculation modules that follow.

<u>Process</u>

```
READ name, hours, rate
DO Calculation
WRITE name, pay, deduction, netPay
ADD pay to pay Total
ADD deduction to deduction Total
```

<u>Calculation</u>

```
    pay = hours * rate
    IF pay > 800
        deductionRate = .2
    ELSE
    deductionRate = .15
    ENDIF
    Deduction = pay * deductionRate
    netPay = pay – deduction
```

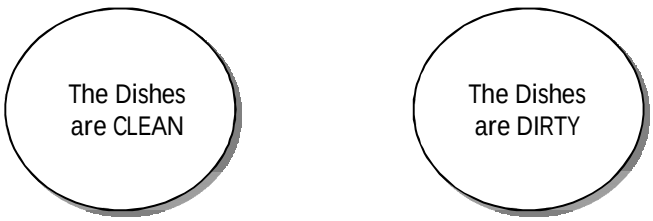
The Calculation module uses six variables: hours, rate, pay, deductionRate, deduction, and netPay. Where do the values come from? Hours and rate are input variables that are read in the Process module. Since the Calculation module uses these values that are read in the other module, we could say that the Calculation module *receives* hours and rate from the Process module. Using these two received values, Calculation calculates the pay variable, and based on the value of pay, it also assigns a value to deductionRate. And finally it calculates the deduction and netPay variables. Most of these variables are calculated so they can be written out in the output report, but they don't get written in the Calculation module. Calculation *returns* pay, deduction, and netPay to the Process module so they can be written out there.

In programming languages, this communication between modules is an important concern. Many languages require the programmer to specify which variables are passed to the called module whenever a module is called. We won't indicate the passing of variables in our pseudocode, but you should be aware that when you begin programming, you will probably need to identify which variables are received by a module and which variables are returned to the calling module.

FLAGS AND SWITCHES

A flag is a variable that we "control" so that we can remember that a certain condition has occurred. Frequently in designing and implementing a program, we need to know at one point in the program if a specified condition has been met or an action has occurred earlier in the program. We use a flag to "remember" that our condition has occurred. We simply establish a variable (a location in memory), and we establish possible values that will have meaning for us. Let's look at a non-programming example first.

In my household, and probably in yours too, it sometimes happens that the dishwasher is run through but the dishes are not put away immediately. And on occasion, someone, not realizing that the dishes are clean, adds dirty dishes to the load. Once I even put away the dirty dishes! My solution is a magnetic circle that is stuck on the outside of the dishwasher and can be flipped so the appropriate side is showing.



The condition that needs to be "remembered" here is that the dishwasher has been run through but the dishes have not been put away. So when the dishwasher is started, the "flag" is turned to "The Dishes are CLEAN," and when the clean dishes are later put away, the "flag" is switched to "The Dishes are DIRTY." And when somebody wants to put some dishes in the dishwasher, he or she can check the value of the flag to know if the dishes are

clean or dirty. Of course, there's a catch to it. The flag is only as good as the faithfulness of the flag-setter. The dishwasher will not automatically reach out and change my flag. I have to remember to change it. We have the same potential problem with flags in our designs.

Let's look now at a program design. Assume that we are designing a program to read and write a list of names from an input file. As we read through the list, we are to check for the name "Sylvester." If we find a Sylvester in the input file, we are to write the message "Sylvester is included in the list" *at the end* of the complete listing. Now how can we do that? The first part is pretty easy. We would have a design that looks like this:

```
Mainline
WHILE there are data
        READ name
        WRITE name
ENDWHILE
```

But, of course, that design ignores the need to check for the name Sylvester. We could add and IF test inside the loop like this:

```
Mainline
WHILE there are data
        READ name
        WRITE name
        IF name = 'Sylvester'
                WRITE 'Sylvester is included in the list.'
        ENDIF
ENDWHILE
```

But we are to write our extra message at the *end* of the list, not as soon as we see Sylvester's name. So how about this solution?

```
Mainline
WHILE there are data
        READ name
        WRITE name
ENDWHILE
IF name = 'Sylvester'
        WRITE 'Sylvester is included in the list.'
ENDIF
```

This may look pretty good, but it won't work. The problem is that we have only one variable (location in memory) called name. so every time we read another name, we lose (forget?) the previous one. The computer can only store one value at a time in a given memory location. When we check the variable name after the loop, we are really checking to see if the *last* name read was Sylvester. That's not what was intended.

One possible solution uses a flag. We initialize the flag to one value and change it to a different value when the test condition occurs. Here we'll initialize the flag to 'no.' If we see a name of Sylvester, we will change the flag to 'yes.' At the end of the loop, we can check the value of our flag to know if we should write out the message. If the flag still has a value of 'no,' we know the name Sylvester was never read.

```
Mainline
nameFound = 'no'
WHILE there are data
        READ name
        WRITE name
        IF name = 'Sylvester'
                nameFound = 'yes'
        ENDIF
ENDWHILE
IF nameFound  = 'yes'
        WRITE 'Sylvester is included in the list'
ENDIF
```

Walk through the design to make sure you understand what is happening. Notice the importance of initializing nameFound to 'no' before the loop. It has to have a value so that we can check it after the loop. Actually we could have initialized it to anything other than 'yes'-'PeanutButter' would have worked since the important part is that it is changed to a value of 'yes' when the name is found and we are checking only for yes. If we had not found a Sylvester, the flag would still be 'PeanutButter,' and when we made our test IF nameFound = 'yes' the answer would be false. So why use 'no'? For the same reason we called the flag nameFound-because it gives more meaning to the design. We want someone else to be able to read our design and be able to understand what's happening.

Would our program work if we forgot to include the IF test inside the loop to change the flag to 'yes'? Of course not! The computer won't change the flag for us any more than my dishwasher will flip over my magnetic circle. We have to remember to initialize the flags and change their values when appropriate.

Flags help us to help make our designs clearer, more readable, more maintainable, and (in some cases) possible. Using meaningful names for our flags and meaningful values helps to achieve those goals. We could also have used d a variable called test instead of nameFound, and we could have used the value 0 as the initial value and 1 as the value if a Sylvester was read. But that wouldn't have made our design very meaningful. An IF test after the loop of

```
IF test = 1
        WRITE 'Sylvester is included in the list.'
ENDIF
```

does not give much help in understanding why we're writing the message. Try to always use meaningful names and values in your designs.

Try another example. Suppose we are reading an input file of inventory data. Each record has the partNum, partName, and quantity, and we know that the partNums should be consecutive beginning with 101. That is, the number for any record should be one more than the number for the previous record. We are creating a report of the inventory, but at will use a flag to remember if the skipped condition has occurred, and we will test the value of the flag after the loop. This design also uses the variable lastNum to save the previous partNum for comparison.

```
Mainline
lastNum = 100
numSkipped = 'no'
WHILE there are data
        DO Process
ENDWHILE
IF numSkipped = 'yes'
        WRITE 'WARNING: A part number was skipped in input file.'
ENDIF

Process
READ partNum, partName, quantity
WRITE partNum, partName, quantity
IF partNum < > lastNum + 1
        numSkipped = 'yes'
ENDIF
lastNum = partNum
```

Try desk checking this design. The table below shows a sample input file, the values that each variable in memory would have at the designated point in the processing, and the output at each point. Step through the entire design to be sure you understand how the flag is working.

INPUT Data

106	Washers	97
107	Nuts	98
109	Bolts	66
110	Screws	43

end of file

Memory Variables

Point in processing	part-Num	part-Name	quantity	last-Num	num-Skipped	Output
before loop				100	'no'	
after loop (1)	106	'Washers'	97	106	'no'	106 Washers 97
after loop (2)	107	'Nuts'	98	107	'no'	106 Washers 97 107 Nuts 98
after loop (3)	109	'Bolts'	66	109	'yes'	106 Washers 97 107 Nuts 98 109 Bolts 66
after loop (4)	110	'Screws'	43	110	'yes'	106 Washers 97 107 Nuts 98 109 Bolts 66 110 Screws 43

Point in processing	part-Num	part-Name	quantity	last-Num	num-Skipped	Output
end of design				110	'yes'	106 Washers 97 107 Nuts 98 109 Bolts 66 110 Screws 43 WARNING:……

PRIMING READS AND LOOP CONTROL

So far we have looked at two different kinds of loops. We have had count-controlled loops in which we knew how many times we were to do a loop and we used a counter in the design. Our loop control test was then based on the counter, for example, WHILE count < 10. We also have looked at loops in which we were to read all of a file, and our loop control was something like WHILE there are data.

Sometimes we want to control our loop based on some other condition. Assume that we are to read and write from a file of all the students enrolled at the school and that the file has an entry of "**LAST**" included to mark the end of the data to be processed. There may or may not be data following the "**LAST**" indicator. All we know is that we should not process beyond that point. Will the following design work?

```
Mainline
OPEN files
WHILE name < > '**LAST**'
        READ name
        WRITE name
ENDWHILE
CLOSE files
```

Try desk checking the design with the following sample input:

```
Bob
Frank
Carol
Susan
**LAST**
2316 2481
4498 8622
End of file
```

You probably found the first problem very quickly. The first time the loop control test is made we don't have a value for name! That's actually the easy problem. The more important problem is that we end up including '**LAST**' in our output list of names. We read '**LAST**' and write it, before we come back to the loop control test and exit the loop. We don't want to write the indicator, just test it to find when to exit the loop. We could fix both of these problems as follows:

```
        Mainline
OPEN files
Name = 'Continue'
WHILE name < > '**LAST**'
        READ name
        IF name < > '**LAST**'
                WRITE name
        ENDIF
ENDWHILE
CLOSE files
```

This works, but it isn't a very desirable solution. We are forced to initialize name to some value other than '**LAST**' just to get into the loop the first time. (It doesn't make any difference what value we choose as long as it isn't '**LAST**' since well will assign a new value to name when we do the READ). Second, we are testing exactly the same condition in two different places in a very short program. The conditions for the SHILE test and the IF test are identical. Redundancy is not always bad; we sometimes choose to be redundant for the purpose of clarity. But in this case, the double test is really just sloppy design. There's a better solution.

Our problem lies in the need to have our loop control test after the READ and before the WRITE. In all our loops so far, we have tested the condition for the loop, and when we entered the loop we knew we wanted to READ and WRITE the next record. This time, we

don't know if we want to WRITE until after we READ and test the value. Instead of assigning a temporary value to name before the loop, we will actually READ the first value. Then assuming we enter the loop, we can WRITE the previously read value and READ the next value in preparation for the next loop control test. Our design ends up looking like this:

```
Mainline
OPEN files
READ name
WHILE name < > '**LAST**'
        WRITE name
        READ name
ENDWHILE
CLOSE files
```

The Read before the loop is called a *priming read*; it is done once, *before* the loop, to "get things going" much like an old hand pump needed to be primed to get the water flowing. The value (like '**LAST**') that is used to signal the end of the data to be processed is called a *sentinel value*. A sentinel value must always be an otherwise impossible value since if it occurred naturally in the date file, we would stop processing early. Suppose that our sentinel value above was 'LAST' instead of '**LAST**' and that a student named 'LAST' was included in the list (unlikely, 1 admit, but still possible). The program would exit the loop as soon as our student 'LAST' was read. By including the asterisks here as part of the sentinel, we make sure it is an otherwise impossible value. Any time you have a sentinel value that will control the loop, you should expect to use a priming read because you need to be able to put the loop control test immediately after the read. Sentinel values aren't used very often when input comes from a data file, but they are very common when the input comes from a user at the keyboard: an interactive program. How many times have you seen some variation of the prompt "Enter your next selection or enter 'QUIT' to exit the program"? QUIT is being used as a sentinel value to exit a loop.

Try another, slightly different example. Suppose you have a file of all the students enrolled in the school along with their GPAs and that the file is ordered by GPA values from the highest GPA to the lowest. Your design is to create a list of all the students eligible for the High Achievers Club, which requires a GPA of 3.8 or better. Since the records are in order by GPA, we can just read and write from the file UNTIL we see a GPA that is less than 3.8. In this case the GPA is not a true sentinel value since it isn't included in the file only to signal the end of the data, but we can treat it like a sentinel value. So we'll have a priming read, and our loop control test will be on the just read GPA. Notice that we do not write the record that gets us out of the loop.

```
Mainline
OPEN files
READ name, GPA
WHILE GPA > = 3.8
        WRITE name, GPA
        READ name, GPA
ENDWHILE
CLOSE files
```

We can make one more modification to our priming read/sentinel value testing. We can set up a flag that will be initialized to one value and changed when the sentinel value has been read, then use the flag in the WHILE test. What follows is the same design, converted now to use a flag. The READ has been moved to a separate module since the two statements together form a unique, and separate, function. It also allows us to avoid writing the same lines in two different places in the design.

```
Mainline
OPEN files
moreData = 'yes'
DO Read
WHILE moreData = 'yes'
        WRITE name, GPA
        DO Read
ENDWHILE
CLOSE files

READ
READ name, GPA
IF GPA < 3.8
        moreData = 'no'
ENDIF
```