

Project 3 — Modular Programming

David Arnold

February 19, 2002

1 Introduction

The best way to attack a problem is to break it down into manageable pieces. This type of programming is known by several names: top-down programming, modular programming, etc. All of these names refer to a common idea. You take a large, intractable problem, and you solve the puzzle by breaking the larger problem up into smaller pieces that are more easily solved.

In this activity, you will be introduced to the modular programming style. The goal is to input three integers, sort them in ascending order, then print them to the screen in their newly sorted order. We begin by writing one large program to accomplish this task.

1.1 Swapping Integers

At the heart of this project lies the problem of comparing two integers, then rearranging their order. Suppose, example, that we have integers, m and n . We test to see if m is smaller than n . If $m < n$, then we leave the numbers alone. However, if $m > n$, then we must switch the contents of m and n . Here is a first attempt at accomplishing our goal.

```
program swap2
integer m, n
read(*,*) m, n
write(*,*) 'Input m: ', m
write(*,*) 'Input n: ', n
if (m .gt. n) then
    m=n
    n=m
endif
write(*,*) 'The value of m is: ', m
write(*,*) 'The value of n is: ', n
end
```

However, when the program `swap2` is compiled and executed, we get the following output.

```
swap2
5 4
Input m: 5
Input n: 4
The value of m is: 4
The value of n is: 4
```

At first glance, the output of our routine is entirely unexpected. What happened to the integer 5? However, close inspection that the output is exactly what our code instructed the computer to perform.

We began with integers

$$m = 5$$
$$n = 4.$$

We then instructed the computer to set the value of m to the contents of the variable n with the instruction:

```
m=n
```

Thus, the contents of n are stored in the variable m and our variables now have the values

$$\begin{aligned}m &= 4 \\ n &= 4.\end{aligned}$$

Our next instruction asked that the contents of the variable m be placed in the variable n .

```
n=m
```

Because m now contains the integer 4, this instruction stores the integer 4 in the variable n . This means that our variables now contain the following integers.

$$\begin{aligned}m &= 4 \\ n &= 4\end{aligned}$$

What is needed is a temporary variable to hold the contents of m , so we adjust our code as follows.

```
program swap2
integer m, n, temp
read(*,*) m, n
write(*,*) 'Input m: ', m
write(*,*) 'Input n: ', n
if (m .gt. n) then
    temp=m
    m=n
    n=temp
endif
write(*,*) 'The value of m is: ', m
write(*,*) 'The value of n is: ', n
end
```

The idea is simple enough. We first protect the contents of the variable m by storing it in a temporary variable with:

```
temp=m
```

Then we store the contents of the variable n in the variable m with:

```
m=n
```

However, this time the original contents of the variable m are protected in our temporary variable. We can now easily slip the contents of the temporary variable into the variable n .

```
n=temp
```

As you can see in the following output, this strategy provides a successful swap of the integers.

```
swap2
5 4
Input m:  5
Input n:  4
The value of m is:  4
The value of n is:  5
```

1.2 Ordering Three Integers

Next, consider the task of arranging three integers in ascending order. Suppose for instance that we are given three integers i , j , and k . Just how would we go about arranging these numbers in ascending order?

Let's consider a brute-force technique.

1. Compare the first two integers. If they are not in ascending order, swap them, else leave them alone.
2. We are now assured that the integer in the first position is smaller than the integer in the second position. But, is the integer in the first position less than the integer in the third position? We need to compare the integers in the first and third positions. If the integer in the first position is smaller than that in the third, leave the order alone, else swap the integers in the first and third positions.
3. We are now guaranteed that the smallest integer is in the first position, so we move over a position to see if the integer in the second position is smaller than that in the third position. If it is, leave the order alone, else swap the integers in the second and third positions.

Steps 1, 2, and 3 above guarantee that our three integers are now arranged in ascending order.

1.3 One Large Program

We now write one large program to accomplish the task of arranging three integers in order. If we think about the task at hand, we need to do three things.

1. Get three integers from the user.
2. Sort the integers.
3. Print the sorted list of three integers.

We begin our task by declaring integer variables to hold the three integers and an additional temporary variable to handle the swaps.

```
program swap3
integer n1, n2, n3, temp
```

Next, we prompt and receive input from the user. We also echo the input back to the user.

```
read(*,*) n1, n2, n3
write(*,*) 'Input n1: ', n1
write(*,*) 'Input n2: ', n2
write(*,*) 'Input n3: ', n3
```

We now apply our brute force method described in subsection 1.2. We compare the first and second integer and swap, if necessary. Then we compare the first and third and swap, if necessary. Finally, we compare the second and third, swapping if necessary.¹

```
if (n1 .gt. n2) then
    temp=n1
    n1=n2
    n2=temp
endif
if(n1 .gt. n3) then
    temp=n1
    n1=n3
    n3=temp
endif
```

¹Start thinking about how you would order a list of 100 integers. This will certainly be a programming assignment in the future.

```

if (n2 .gt. n3) then
  temp=n2
  n2=n3
  n3=temp
endif

```

Now that we have a sorted list, we need to show the user the results.

```

write(*,*)
write(*,*) 'The sorted integers are:'
write(*,*)
write(*,*) 'n1: ', n1
write(*,*) 'n2: ', n2
write(*,*) 'n3: ', n3
end

```

Of course, a sample run is required. We must test that our program actually sorts the integers.

```

swap3
5 4 3
Input n1: 5
Input n2: 4
Input n3: 3

```

The sorted integers are:

```

n1: 3
n2: 4
n3: 5

```

All seems well.

1.4 A Subroutine that Swaps

You might have noted that the construct

```

temp=n1
n1=n2
n2=temp

```

or something similar, was used on three separate occasions. Imagine if your were sorting a list of 100 numbers. Can you count how many times you would have to write this construct in your code?² Prohibitive, to say the least. It would be much more efficient if we could reuse this code in some manner. This would make our code more “modular” and greatly reduce the amount of typing required.³ In Fortran, the subroutine is used for this task.

This is the basic idea behind “modular” or top-down programming. You take a small part of your program and you place it in a module, in this case a module called a subroutine. This is problem solving at its finest, taking a larger programming problem and breaking it into smaller problems. In this example, we concentrate on one piece of the puzzle, the process of exchanging two integers.

The subroutine is a subprogram that is called from the main program. It can be included in the same file as the main program, or, as we shall see, can be compiled separately and even stored in a “library” of subroutines for later use. The subroutine in this instance is assigned the task of swapping two integers.

The subroutine is called from the main program with the following syntax

```

call subroutinename(actualargument, ... , actualargument)

```

²The answer is 4,950 times. Can you explain why?

³Particularly if the construct could be embedded in some sort of DO loop.

The subroutine itself has the following form.

```
subroutine subroutinename(dummyvariable, ..., dummyvariable)
declarations
fortran statements
end
```

Here is the subroutine for swapping two integers.

```
subroutine swap(a,b)
integer a, b, temp
if (a .gt. b) then
temp=a
a=b
b=temp
endif
end
```

Note that the subroutine begins with the keyword `subroutine`. Next comes the subroutine name `swap`. Then, included in parentheses, comes the dummy variables `a` and `b`. We then make the usual variable declarations. After that, the Fortran code for swapping `a` and `b` is written. Note that this code is practically identical to the code used in `swap3` in subsection 1.3. Finally, the keyword `end` signals the compiler where the subroutine ends.

One comment is in order regarding the variable `temp` in the subroutine `swap`. This variable is *local* to the subroutine `swap`. This means that even if the variable `temp` is defined and assigned in the main program or another subroutine or function, that assignment has no affect on variable `temp` that is local to the subroutine `swap`. This means that you need not fear that changing the contents of the variable `temp` in subroutine `swap` will harm the contents of another variable named `temp` that might reside in other parts of your program. Indeed, the variable `temp` is created when the procedure `swap` is entered, but it is destroyed when the subroutined finishes its task and returns control to the calling program.

1.5 Calling the Subroutine Swap

It is now a simple matter to call the subroutine `swap` defined in subsection 1.4. Simply replace each block of code in the main program `swap3` written in subsection 1.3 with a call to the subroutine `swap`. For example, replace the lines

```
if (n1 .gt. n2) then
temp=n1
n1=n2
n2=temp
endif
```

with a call to the subroutine `swap` as follows.

```
if (n1 .gt. n2) then
call swap(n1,n2)
endif
```

There are a number of important points to make at this juncture.

1. Note that the variable names in the calling instruction

```
call swap(n1,n2)
```

do not have to be the same as the dummy variable names used in the subroutine definition

```
subroutine swap(a,b)
```

However, they can be the same if you wish them to be. The point is, they don't have to be the same.

2. However, They must agree in type. If you try to pass an actual argument that has integer type to a dummy variable having real type, then your program is going to experience difficulty. Maybe not at compile time, but certainly at run time. Modern Fortran 90 compilers will catch this type mismatch, but g77 will not.
3. The actual variables in this case are passed by reference. This means that an address containing the variables' contents is what is passed, not the contents of the memory location itself. This means that when the subroutine changes the contents of *a* and *b*, these changes also occur in the variables *n1* and *n2*.
4. It is an entirely different matter to call the subroutine with actual values instead of variables, as in:

```
call swap(15,10)
```

We will not need to discuss this here, but it is an important point which we will later address.

Although the subroutine can be compiled separately, then linked to the main routine, in our first examples we will include the subroutine in the same code containing the main program. This is shown in program *swap4*.

```
program swap4
integer n1, n2, n3
read(*,*) n1, n2, n3
write(*,*) 'Input n1: ', n1
write(*,*) 'Input n2: ', n2
write(*,*) 'Input n3: ', n3
if (n1 .gt. n2) then
    call swap(n1,n2)
endif
if(n1 .gt. n3) then
    call swap(n1,n3)
endif
if (n2 .gt. n3) then
    call swap(n2,n3)
endif
write(*,*) 'The sorted integers are:'
write(*,*)
write(*,*) 'n1: ', n1
write(*,*) 'n2: ', n2
write(*,*) 'n3: ', n3
end

subroutine swap(a,b)
integer a, b, temp
if (a .gt. b) then
    temp=a
    a=b
    b=temp
endif
end
```

Of course, only a test run will show if we've correctly solved the problem.

```
swap4
5 4 3
Input n1: 5
Input n2: 4
Input n3: 3
```

The sorted integers are:

```
n1: 3
n2: 4
n3: 5
```

It looks as if all is well.

1.6 Subroutines Without Arguments

It is entirely possible to design a subroutine that needs no arguments passed from the main subprogram. This is an excellent way to modularize your code, breaking a larger problem down into a sequence of smaller tasks. Then each individual piece can be written, tested, and included in the final program. This is the most efficient way of writing Fortran program.

For example, consider the following silly piece of code.

```
program swap5
  call hello
end

subroutine hello
  write(*,*) 'Hello'
end
```

Obviously, the same result is easily accomplished with the following code.

```
program swap5
  write(*,*) 'Hello'
end
```

However, an extremely important point is being made here, and that is the concept of program modules. Again, to reiterate, modular programming requires that you break a large program down into smaller program units, test them, then include them in the main program. This is precisely the technique you will use on this week's programming assignment.

2 Project 3

Your programming task is to completely modularize the program written in section 1. The problem of swapping three integers is easily broken down into a sequence of smaller tasks.

1. Get three integers from the user.
2. Sort the three given integers in ascending order.
3. Print the sorted list.

If you are thinking modularly, then your main program will look like this:

```
program swap3
  integer n1, n2, n3
  call enterintegers(n1,n2,n3)
  call sort(n1,n2,n3)
  call printresults(n1,n2,n3)
end
```

You need to write four subroutines, `enterintegers`, `sort`, `printresults`, and the `swap` subroutine written in section 1.4.

Save and compile your program, then, once it's running properly, obtain a hardcopy printout for submission. In addition, place all files associated with this project in your Fortran folder in a folder called Project3 on your H drive on the school's server. I will test each of your programs for accuracy from my office, but I won't be able to locate your files unless you follow this convention.

3 The Grading Rubric

The following rules will apply for this program, after which we will discuss and adjust the rubric during class.

1. (30 points) Will be awarded for adequate comments. Comments should include:
 - (a) A description of the program's purpose.
 - (b) Name, date, version or revision number.
 - (c) A complete dictionary of all variables and parameters used in the program.
 - (d) Interprogram comments should precede any code snippets explained by the comments. These should be adequately sprinkled throughout your code.
2. (50 points) Will be awarded if the program works and does what it was asked to do.
3. (10 points) Will be awarded for good program style. This includes good indentation practices, etc.
4. (10 points) Will be awarded for creativity and extra effort. Did you just do the bare minimum? Or did you stretch and reach a little higher? Did you put something cute or clever into your program that nobody else seemed to think of?

4 Penalties

Each program that is assigned during the term will have a due date. On that date, the program must be on the instructor's desk before the start of class. Penalties will be assessed as follows.

1. (10 points) There will be a 10 point deduction for any program that is handed in after the class has begun.
2. (20 points) There will be a 20 point deduction per class period. That is, if you hand the program in one class period late, there is an automatic 20 point deduction. Two class periods warrants a 40 point deduction, etc. To be clear, if the program is in the instructor's hands before the beginning of the next class, that is a 20 point deduction. If the program is in the instructor's hands before the start of the second class period past the due date, that is a 40 point deduction, etc.

5 Managing Files and Folders

Each of you has been given personal space on the sci-math server to store your work. Typically, this space is mapped to the drive letter H. If you open the Windows Explorer (the file manager, not the internet browser), you can see that the drive letter has been mapped to your login name.

In this folder, create a new folder call FortranPrograms. Note that you must **never** use spaces in filenames. In the Windows operating system, filenames are not case-sensitive, which is exactly opposite what happens in Unix and Linux, where filenames are case-sensitive.

In your FortranPrograms folder, create another folder called Program3. It is in this folder that you are to place the source code and executables for this current project. When you receive your next project, create a new folder called Program4 to hold that project, etc.

If you work at home, I still want you to place copies of your work in the space reserved for you on our system. Simply copy your home files onto a floppy disk and bring them with you to school. Use the Windows Explorer to copy the files on your disk into the proper folder (H:/FortranPrograms/Program3).

If everyone follows these simple rules, I can easily access your work from my office machine for purposes of assigning a grade.

6 Caveat

On this project, if you stop by my office with hardcopy of your program before the due date of this assignment, I will give a quick glance and critique of your source code. Somewhat like receiving a grade on a draft before submitting your final draft for assessment.