PART 2 INTRODUCTION TO STRUCTURED PROGRAMMING

CHAPTER 5 FUNDAMENTALS OF MODULARIZATION

IN THIS CHAPTER, YOU WILL LEARN HOW TO:

- Define procedural programming techniques and its implications on logic design
- Define modularization and describe its benefits
- Describe identifier and module scope and the implications of global, local and block scope designations
- Describe the process of identifying code to modularize
- Recognize the importance of module abstraction and reuse
- Demonstrate how called modules can accept parameters and return a value
- Recognize how different programming languages implement modules

PROCEDURAL PROGRAMMING TECHNIQUES

We started in our first three chapters taking baby steps with logic models and simple programs. In our last chapter, the pace quickened as we learned about control structures and how to add complex logic to our programs that in previous chapters we had to avoid. In this chapter, we add the organization of procedural programming techniques to design more complex logic models and programs with the modular approach characterized by procedural programming.

We will begin to incorporate procedural programming techniques into our logic design. Procedural programming is characterized as a problem solving technique that follows a “divide and conquer” approach. What is meant by “divide and conquer” is a stepwise approach that takes a complex problem and break it down into smaller subtasks. The pieces can communicate with each other so the computer program becomes a series of subtasks that make up the whole. There is quite a bit of evidence that suggests that this approach of problem solving can be very effective in designing complex solutions.
Procedural programming gets its name from its dependence on procedures (or blocks of logic) that create statements to perform one subtask. A program is a collection of many sub tasks or procedures that together fulfill the requirements behind the program. Just like control structures, procedures are found in most programming languages. Whereas control structures are typically implemented with if and while statements, procedures come by many names (like function, method, subroutine and procedure). Even if the name is different, all of these terms refer to a procedure that represents a subtask of the program.

A procedure is very similar to the code block used by control structures. The difference is that a procedure typically contains many structures and statements and is called out in program logic to be run as a sub task.

**Key Concept:** The procedure or module is executed when it is called in the program. When called, all of the statements in the module are executed and then control is return back to the call point where execution is now focused on the next statement. If the module returns a value, then the statement that called the module must receive that value and use that value in the calling statement.

Let’s choose a non programming example to illustrate procedural designs. Do you suppose that the design team who constructed the space shuttle sat down in front of the drawing board and designed all of the systems from start to finish? Would they have done this without ever trying to break down shuttle functions or operations into smaller pieces? I would suggest that the answer is no and that the space shuttle, although it had a design that joined all the systems, most of the space shuttle was designed in pieces. One piece or sub task might have been the launch system. Other pieces might have been the communications, reentry, cooling and heating and crew quarters. One other piece or subsystem might have been the arm used in the cargo bay. It is likely that each of the components on the space shuttle was given to a different group of engineers to design with one group focusing on merging all of the components. Now, as I am sure you can imagine, I had no part in the design of the space shuttle but I have been involved with development of several very complex systems. What characterized my involvement in the design of these systems was the technique of modularization and the process of taking the larger problem and breaking in down into smaller pieces.
**Procedural Programming:** Procedural programming is built around the concept of developing programs based on sub tasks (also known as modules of code). The modules are called and connected by a single parent module that coordinates all of the modules into a working solution. Procedural programming gets its name from procedures. A procedure is a subtask that is implemented in programming languages under a variety of language specific names (i.e. functions, procedures, methods, subroutines, etc.)

---

**Modularization Defined**

Modularization is the process programmers use to break up complex programs into smaller more manageable procedures. These procedures are also referred to as modules. To borrow an earlier example from chapter four, I spoke of a procedure to calculate sales tax as an example of statements in a program that could be bundled together and represent a sub task of the entire program. If the program I am writing needs to calculate the cost of a customer order, if I break off into a procedure the sales tax processing and possibly the logic for quantity discounts, I can concentrate on the design and testing of each module or subtask and help to make the program logic more effective. The “divide and conquer” approach will allow me to work on each piece individually and later connect those pieces for the finished product. Aside from the obvious benefits of reducing program complexity, modularization helps in many other ways.

---

**Modularization:** Modularization defines the process of reviewing program logic to identify program subtasks. When we divide the program up into subtasks we make the design of logic simpler and more accurate. The logic contained within the subtasks is stored as modules. These modules are combined with statements to make up the program.

**Modules:** A module is a block of code that represents a program subtask. This subtask is a standalone piece of code that performs part of the program’s solution but also could stand by itself and potentially be used in other programs. For example, I could create a sub-task that calculates the sales tax of a purchase. It is only part of the logic but is required for the final solution. Because it can stand alone, it could be reused in other programs that have a similar requirement (i.e. other programs also would need to calculate sales tax).
BENEFITS OF MODULARIZATION

Simplifying programs is just one of the benefits of modularization. It may be the most obvious and also one of the more important benefits but it is valuable in other ways. Modularization helps us as programmers by providing improved organization to our logic and programs.

BENEFITS CODE REUSE

In regards to modules, there are two situations where a module could be a used in the development of programs. The most common application of code reuse is the use of modules across different programs. If the module is designed to stand alone and be independent of the rest of the program, there is no reason why a module that’s developed for one program might not work just as well in another. To use the example of sales tax, for company that does a retail business there will be many opportunities for the sales tax module to be used in programs. The program might be used at the register or might be used to print a hard copy invoice to a customer. In its simplest form, module reuse can be simply copying source code from one program to another. This can be dangerous since a programmer unaware that the procedure is reused could change the code so that it is different than the original copy. This could create a main its problem on future updates. Many languages allow for modules to be compiled to the source code is hidden but still the logic is available via programmer built interfaces to the module.

**Logic Tip:** Procedures, subtasks and modules all equate to the same thing. In each case, they are terms used to indicate a collection of statements and structures that represent some subtask of the entire program. This wiki will use modules most of the time but subtask and procedure will also appear in the text from time to time.

A reused code module, if developed by an experienced programmer who has experience designing modules that are of high quality and performance, can be a tremendous asset to an organization. Their code can be used by everyone in the organization to save time. For the new programmers who are still learning design, having a library of tested and well designed modules make there work higher quality and teaches them the characteristics of well written code modules.
Background Information: An economic case for code reuse - Programming code reuse has become a very fashionable subject in program design. As more and more languages support object orientated programming, reuse is becoming more and more a part of the programmers design alternatives. Aside from design benefits, there are also economic reasons for reusing code. Most programming departments have one thing in common. That one thing is the inability to keep up with a number of programming requests that come in every day. Providing ways of making programmers more productive is critical to most organizations. Using design techniques which provide for code reuse is one way of reducing the amount of time needed to develop software. After all, time is money and saving time is saving money. We will see code reuse as an advantage in modularization and also we will see this highlighted again when we discuss object orientated programming.

A SIMPLE PSEUDO CODE EXAMPLE OF MODULARIZATION

The following code snippet demonstrates how a program can have its logic organized into sub-tasks or module. First we have a pseudo code program with out modularization.

```
start
  input "enter amount"
  if countryCode = "Maricopa" then
    taxRate equals .086
  else
    taxRate equals .075
  print "Sales Tax = " taxRate times amount
end
```

After modularization the same program would now look like the following.

```
start
  input "enter amount"
  call salesTaxModuleNameProc()
  print "Sales Tax = " taxRate times amount
salesTaxModuleNameProc()
  if countryCode = "Maricopa" then
    taxRate equals .086
  else
```
taxRate equals .075
return
dend

- The call statement tells the program to look for the salesTaxModuleProc.
- Here the salesTaxModuleProc is written in our program and the statements for sales tax have been placed together in this module.
- The return keyword signals the program that the module is done and to return to the next line in the program (i.e. the print statement).

A second form of reusability that can benefit the program includes recalling (a slightly different kind of reuse) the module from within the program itself. In the case of sales tax, it is highly possible that sales tax pay will need to be calculated several times in the same program. You have some options. You can either create the statements necessary for sales tax and place them in a module or duplicate the statements every time there needed within the program. The more efficient alternative would be to create a module and call the module whenever the sales tax calculation was needed. An immediate advantage here is program maintenance. If the code and module need to be changed for any reason, the change only needs to occur in one location (the module) and any place that module was called will automatically pickup the code change.

**Programming Tip:** Many programming languages compile modules so that the source code (and therefore logic) is not visible to the programmer. This means you are using a module that you are trusting works correctly but do not know how it accomplishes its task. This hidden logic can be understood if you understand the power of abstraction in program design.

**ABSTRACTION**

Modularization can occur in both logic models and in programming. For logic models, the modules are built in either pseudo code or flow charts. All of the logic is “English” like and easy to read. In programming this is not always the case. Many reusable code modules are compiled into machine readable formats with only helps files to instruct the programmer how to implement the module. The code behind the module is not visible to the programmer as source code. A couple examples of this “hidden source” code occurs in the Java application programming interface and the Microsoft
.Net Framework. Each contains thousands of reusable code modules but neither actually shows the programming statements behind them.

The concept of compiled reusable modules is defined as abstraction. Abstraction is one of the foundation points for modularization and object oriented programming. Abstraction is discussed frequently in programming but it also has a larger meaning. Abstraction also is used to describe how humans interact with “things” (devices, machines, equipment, etc.) in our environment. Abstraction says that to use something we only need to know how to communicate with it. Communication requires that we are able to send inputs and that we have some idea of what we will receive as outputs. As humans, we use abstraction everyday as we interact with things in our world. It is from this application in humans that the concept was carried over to computer programming.

**Abstraction:** Abstraction is a process where logic or program details are not available to the programmer. Rather than seeing things in detail you are shown an abstraction or only what is necessary to communicate and use the module. This information includes the modules inputs and what is returned as outputs.

My favorite example to illustrate abstraction is the data projector mounted to the ceiling of my programming classes. The data projector allows me to take Power Point slides and project them on a screen in the front the room. I use the following analogy to explain how abstraction eliminates the need to know all of the details. I asked the students if they know how to use the data projector? Many of them shake their head yes. I ask them do they know what the wiring schematic looks like for the electronics or what plastic polymer was used to make the data projector case? Do they know what kind of glass used in the lens? I get some very strange looks but then I explain to them that what we really need to know is how to set the data projector messages and understand what outputs it will in turn deliver back to us. We used the data viewer with knowing the details. As humans, we are very comfortable using abstraction.

I think most of us would instinctively use the data projector as an abstract thing. Using abstraction, we would intuitively know that we need to plug the graphics cable into the computer and the power cord into the wall socket. We would need to push the on button and adjust the volume and focus so that the picture is displayed clearly. We do not need to know any of the other details that went into the making of the data viewer or what is happening inside the data viewer to make it work.
Correctly. Most of the things we use daily, we interface with by only knowing the inputs and the outputs. The process in the middle is something which we ignore. This is abstraction.

**Related Subject:** Black boxing it! - There’s an expression which is used in the world of engineering and also the world of computer programming called “black box.” When we refer to something as a black box, we are saying that we acknowledge that a device or program is instrumental to our solution but we have chosen to ignore the details (like the data projector). We will assume that the outcome of the black box step is predictable and consistent even if we do not know what it looks like. The term is really talking about abstraction and modularization where we identify our problem as a series of subtasks and acknowledge that some of these subtasks are things we do not need to know about or understand to still come up with a solution.

**BENEFITS: SPLITTING DEVELOPMENT OF PROGRAMS**

Dividing programs into subtasks via modularization can be done at various levels of detail. In other words, modularization can take place at the program level and modularization can also take place at the system level. For example, you are one programmer of many involved with the development of a general ledger accounting system. Since a system can be broken down into smaller pieces, your responsibility might be the program using in creating general ledger reports. Another programmer on your team may be responsible for creating the programs to back up the general ledger data files. Modularization allows more than one programmer to work on a project at one time. We are focusing on modularization at the program level in this class but breaking down problems into smaller pieces can take place at many levels.

In addition to allowing more than one programmer to work on a system at one time, modularization also allows development of new programs to be split in such a way that the different pieces of the program can be developed independent of one another. For example, within your program you may have identified twenty modules. Since the modules stand independent of one another and of the program, it is possible to work on that one module, complete it and then pick any other module and the sequence and development. Each module can independently be designed and tested. Later when all the modules are complete, they can be reassembled with the rest of the programs statements for the final solution.
**Status Check**

What is meant when we say to “divide and conquer” when solving a problem?

How is using computer hardware an example of abstraction?

How does modularization promote logic/code reuse?

**A PROGRAM BEFORE AND AFTER MODULARIZATION**

The process of modularization requires that we include an extra step in the logic design process to isolate sections of logic which can standalone as subtasks of the entire program. I will choose an example that I’ve already referenced in the book and one which would be familiar to all of us. We are writing logic for a program which will calculate the sale of an item at a retail establishment like Cactus Books and Tapes. For purposes of our instruction, I will simplify the program from what you might find implemented at your local bookstore. We’re not trying to avoid the hard parts but want to concentrate on just modularization in this example.

**Key Concept:** When we are modularizing a program we are in effect providing improved organization of our logic or programming source code. To illustrate, think of your city phone book sorted in alphabetical order with a cardboard tab placed at the start of each new letter and the alphabet. Now think of a telephone book that is only sorted in alphabetical order. Which of the two phone books will it be easier to find information in? Declaring modules to better organize your code in your program is a lot like adding cardboard tabs to the letters of the alphabet in a phone book. The end result is a program which is easier to maintain and understand because the modularization has made it a better organized solution.

When you purchase a book at the bookstore, the book is taken by the cashier and keyed or scanned into the register to calculate the cost of the sale. For most book stores, there are typically discount cards which give the customer a discount after purchasing so many books. If the discount applies, the discount percentage is taken off the cost of the books. The last part of the transaction involves calculating the sales tax. The sales calculation is completed and you’ll now know how much to pay the cashier for your purchase. In this example, we will take the discount off of the
price before we apply the sales tax. I will show both the flowchart and pseudo code for this application so that you can see the flow of the logic.

Figure 1: Flowchart of Sales Purchase

```
start
input items cost into cost
input discount card number into discountNumber
if discountNumber exists then
    call calcDiscount()
else
    totCost equals cost
    call calcSalesTax()
print totCost
end
```

```
calcDiscount()
    bookDisc equals cost times .10
    totCost equals cost - bookDisc
return
calcSalesTax()
    input State Code into stateCode
    if stateCode = "AZ" then
        totCost equals totCost times 1.08
    return
```
start
input items cost into cost
input discount card number into discountNumber
if discountNumber exists then
    bookDisc equals cost times .10
    totCost equals cost - bookDisc
else
    totCost equals cost
input State Code into stateCode
if stateCode = “AZ” then
    totCost equals totCost times 1.08
print totCost
end

This example is a short one but there’s enough code here to perform modularization. Let’s look at the preceding pseudo code. After looking closely at the logic, we can see three subtasks developing that when added together perform the sales purchased logic. Inside the logic, we can identify logic steps related to the application and calculation of a sales discount. We can also see inside the pseudo code, the logic statements that pertain to the calculation of sales tax for Arizona residents. Beyond that both of these sub-tasks work together as part of the larger application involved with processing the sales purchase. Sales purchase is the program that controls the logic and calls the two sub-process sees for discount and sales tax at the appropriate time and the program.

Once the subtasks are identified they can be named in segregated inside their own code block or drawing in a flowchart. The module names should reflect the function of the module and use camel casing if the module has more than one word in the module name. In this example, I have used calcDiscount() and calcSalesTax() respectively to identify my two modules. Both of these names have trailing left and right parenthesis included as part of the name. These parentheses will be explained later in the chapter but for now include these as part of the module header.

Modules in a flowchart

We can now explore how the flowchart is changed after modularization has been performed. We now see the flowchart that rather than having one chart including all the logic steps in the program, we now have one main program which calls to modules. The two modules are also drawn with flow chart symbols and are shown on
the same page. The symbols and structures used for the modules are the same as
the symbols and structures used in earlier programs. If you look carefully at the
flowchart diagram, after modularization you’ll see some changes in some of the
labeling. For the two modules, you will see that instead of a start and stop a label on
the terminator symbol, you will instead see the start symbol indicating the name of
the module (again you will see the parenthesis next to the name) and the ending
terminator symbol labeled with the word return.

Figure 2: Sample start and stop symbols for modules

After you have identified symbols as candidates for modules, you would remove
those symbols from the main flowchart and use them as the starting point for your
module flowchart. Where the statements are removed you would insert a new
module symbol that consists of a rectangle that has two lines drawn across the top
and bottom or down the sides of the rectangle thus making them different than our
sequence symbols. Inside the module symbol would be the module name.
Modules are typically called from the main program or called from other modules (yes, modules can be nested just like structures). When you call a module, execution moves to the module and the statements are executed just as any flowchart except at the conclusion of the module diagram the return label on the terminator indicates that execution control returns to the calling program (or module).

**Tip Logic:** Calling modules is a lot like clicking a hyperlink on a web page. When you click on the hyperlink you temporarily change focus to a new page and then if you click on the return button focus goes back to the original page. Modules work in much the same fashion. They are called from the main program and when they finish executing their logic, they return to the call point.
start

Enter Item Cost Store in Cost

Enter Discount Card No. Store in discountNumber

discountNumber exists

false

totalCost = Cost

true

bookDisc = Cost * .10

totCost = Cost - bookDisc

Enter State Code store in stateCode

stateCode = "AZ"

false

Print totCost

End

totCost = totCost * 1.08
Figure 4: Before Modularization
Figure 5: Flow chart after modularization was performed
MODULES IN PSEUDO CODE

The process of changing a pseudo code program into one that contains modules is similar to that of flowcharts. The process of isolating subtasks still needs to be performed with pseudo code. The rules and considerations for breaking the program up into pieces still apply with pseudo code. The only difference from flowcharts is rather than using symbols, we now use logic statements. When the module is called for execution you’ll need a verb like call or perform to signal the pseudo code to find the code module and begin processing the logic steps within that module. At the end of the module, you’ll see a return statement to redirect execution back to the statement below the calling statement. Below, I have an example of pseudo code which calls modules as part of the logic. This is the same program we just put into a flowchart.

Pseudo code sales purchase logic with modularization

```
start
input items cost into cost
input discount card number into discountNumber
if discountNumber exists then
call calcDiscount()
else
totCost equals cost
call calcSalesTax()
print totCost
end

calcDiscount()
  bookDisc equals cost times .10
  totCost equals cost - bookDisc
return

calcSalesTax()
  input State Code into stateCode
  if stateCode = “AZ” then
  totCost equals totCost times 1.08
return

- The calcDiscount() represents the sub task that calculates the discount. This standalone module could easily be cut and pasted into another pseudo code program.
```
This block of code passes the test qualifying it as a module.

- `calcSalesTax()` also qualifies as a module. Notice that modules can have control structures and these structures can be nested. Notice how this code was taken almost statement by statement from the original pseudo code.

```
start

input items cost into cost

input discount card number into discountNumber

if discountNumber exists then
    call calcDiscount()
else
    totCost equals cost

    call calcSalesTax()

print totCost

end
```

- `calcDiscount()` and `calcSalesTax()` are modules. Notice how these functions are modular and can be called within the main code to perform specific tasks.
The process of modularization concerns how to best logically break up the program into subtasks. In my design, I made a decision to include all the logic for processing State Code inside one module called calcSalesTax(). I also chose to keep part of the discount logic in the main program and implement actual calculation within the module. An argument could be made to include the entire discount logic within the calcDiscount() module like was done with calcSalesTax(). This design would have been perfectly acceptable. We will look at some criteria you may want to use in determining what to modularize but as with many things in programming design, it is largely reflective of how you want the logic statements to interact and what is resuable. When the computer compiles or interprets the statements, it will not care if there are modules or not. It will run equally the same with no benefit such as increased speed or smaller executable files.

**Logic Tip:** The example above has the pseudo code start and end keyword added to the logic. I have omitted these in many examples because it is only seen in complete programs and not the code snippets found in many of the wiki examples. This is significant because you will see that the modules are outside the main logic block do not have start and end keywords. The main logic block (also called the main module), contains the logic which controls the sequence of statements and modules necessary to implement the solution. The modules by themselves contain only the logic of the subtask they are associated with. The main module that controls the program will always have a start and end label.

**DESCRIBE THE PROCESS OF IDENTIFYING CODE TO MODULARIZE**

There are no hard and fast rules or techniques that can be used for isolating statements in your program that make good candidates for modularization. Selecting modules requires practice and as you become more experienced in designing logic, the modules will stand out from other statements in your program. That is not to say that there are not some considerations that can be made to determine if the program should have modules and if parts of the program make better candidates for modularization than others. Some considerations can take in determining if code requires modularization are as follows:
• Does creating a module out of a block of logic contained within the program going to simplify the understanding of the program and reduce the possibility of design errors or maintenance errors? If the answer is yes, then this code is a good candidate for modularization.

• Does the code identified for modularization a good candidate for reuse across multiple programs? If the answer is yes, then it makes sense to modularize that block of code.

• Does the code identified for modularization represent a subtask and an independent piece of logic? If it is a legitimate subtask, then it is an excellent candidate for a module.

• Will breaking up the code into modules facilitate multiple programmers being able to work on a program in parallel? If the answer is yes, then modularization would make sense.

• If the code was modularized, would it allow the development process to be done in stages and allow development to be done on the program at different times? If yes, then modularization makes sense.

**REPRESENTING MODULE DESIGN IN A STRUCTURE CHART**

We can now introduce a new chart to our programming repertoire. As we begin to modularize our programs we will need to nest modules to make the code easier to understand, we will need to somehow document this nesting so that we don’t add complexity to our solution. We use structure charts to document the calling and nesting of modules. A structure chart acts like an organizational chart with the main program sitting at the top of the chart and the called modules listed below. If the module calls another module, a line is drawn from that parent module to the child modules it calls. Here we have a structure chart example using the sales purchase case study but with more complexity and a greater number of modules.
In this structure chart, I’ve changed the original logic slightly to accommodate the nesting of modules. The first difference you’ll see is that at a new module called printSale(). We’ll printSale() calls the modules printDisplay() and screenDisplay(). But the structure diagram shows as is that printSale() is the parent and therefore the call point of the two sub-modules called printDisplay() and screenDisplay() respectively. For the rest of the diagram, both calcDiscount() and calcSalesTax() now have new sub-modules which they call.

**Status Check**

What are some of the benefits of modularization in logic design?

How does a structure help identify nested modules?

**IDENTIFIER AND MODULE SCOPE**

With the introduction of modules, we now need to return to a discussion of identifier and module scope (scope will also appear later in the wiki as we introduce class objects). Scope determines when an identifier or module is visible to the statements of your program. Most programming languages support global and local scope with the newer programming languages also supporting block scope. Scope can vary by programming language so it is difficult to speak of scope in universal terms. Some
programming languages support extensive options for scope implemented with a modifier or keyword. The scope a programming element can affect the length of time it is available in memory, the amount of memory your program reserves and the programs ability to easily reuse blocks of code. All of this will be covered in this chapter.

### TYPES OF SCOPE

Most programming languages typically support to levels of scope and optionally a third. The most common and sometimes the most commonly abused scope is global. An identifier or module with global scope is available to any statement within the program throughout the life of the program. Local scope is typically associated with modules. Typically, when a variable is declared within a module is only available for use with that module and is referred to as local. Finally, many of the newer languages are supporting block scope which says that variables that are created within a structure, such as an if block or a while block, are only available while the program is within the block. A more comprehensive and specific explanation of each is as follows:

- **Global** - a global variable or module is one which is available to the entire program for the entire life of the program (unless specifically destroyed within the program). Identifiers with global scope are very useful when variables need to be available across to several modules. For example, let’s say that during the program you’re designing you have several places where you need to add to a totalCost variable. By making that totalCost variable global, the variable is available to be updated or to be read by any instruction in the program.

- **Local** - A local variable or module is a element that has been defined to exist only within a program module. When control is transferred to the module the variable is declared. The variable or module is visible and available to be read or written to only while execution stays within the module. When control has transferred back to the module call point local variables disappear along with their contents. If the module’s called again, the local variables are created from scratch and at the conclusion of the module are again deleted. It is possible with module parameters (discussed later in the chapter) to pass values from outside the module into the module’s local variables. This technique is very important in the creation of modules that need to be reusable.

- **Block** - Block variables and modules exist only within the structures were they are defined. It is very possible that one building a decision structure using an if statement or
building a repetition structure using a while or for statement did variables are declared to be used with other structures. The statements maybe a simple as a sequence structure or the variable could be used in something more complex like a nested if or nested while block. As the block terminates, the block variable is released by the operating system and the contents are emptied. If control is returned to the structure, just like local variables the variables are redefined an error raced at the conclusion of the block.

Logic Tip: Given the variability in implementing a global, local and block variables, we have assumed and continue to assume that all variables defined in a flow chart or pseudo code are global. As we implement pseudo code into PYTHON solutions we will see scope modifiers and the existence of local and global variables now become part of program logic.

| In Practice: | Why not make everything global? - One of my more predictable questions following a discussion of global and local variables is a student asking why you would not want to declare all variables as global. Isn’t it much easier to declare the variable once and then not have to worry about whether using it inside a module, inside a block or anywhere in the program? If it is global the variable is always available to be read or to be written to. |

The problem occurs when a module with a global variable needs to be reused. If you use local variables and pass information into the module via parameters, the variables within the module will always remain local. If you use global variable inside a module, this means any time you reuse that module you’ll have a requirement to also create a global variable. Local variables and modules do not require any code outside of what is contained within the module.

Another less important reason against using all global variables is that the global variable stays in memory until the conclusion of the program. Not letting the variable release itself after being used (i.e. a local variable that is destroyed when the module ends) is inefficient use of memory and if the application was being developed for a smaller device (i.e. a PDA or smartcard), that inefficient use of memory may cause problems running a program on these memory constrained devices.

| Logic Tip: | We have conveniently omitted the declaration of variables in our pseudo code and flow chart logic diagrams. There are provisions in both flow charting and pseudo-code to declare variables used within the logic. Typically, since seudo code |
and flowcharts don’t support extensive data typing, you would typically declare variables to be either char (character) or num (number).

PASSING DATA TO MODULES - RETURNING DATA FROM MODULES

One of the more powerful aspects of modules is the ability to accept data from other modules or statements outside of the module. If the module could only work with statements and data defined within the module, the module itself would be very limiting. Modules can not exist as islands in the program but must exist as resources to be called/shared by the program. There have been extensive libraries of modules created for programming languages which are designed to be ready for use by programmers. Almost all of these library modules and probably most of the modules you’ll design will support the passing of data values to the module’s logic. Also, modules have the capability of returning a data value. Please note that only a single value or object is returned. As you’ll find in the chapter on arrays and class objects, and object can contain a more than one value so it would be incorrect to say that modules return a single value exclusively.

**Key Concept:** Modules can contain any number of parameters as input but typically can only optionally single value as output.

PASSING PARAMETERS TO A MODULE

When a module accepts parameters as input it is not without rules and structure. As the designer of the module you are going to set how many parameters can be accepted as input, the data type of those parameters and the order in which those parameters must be passed. If the data type is different between the values sent from the calling statement to the module, the compiler will return a syntax error. If you send to many or too few parameters from the module call statement to the module, you’ll also get a syntax error. If the order of the parameters sent from the calling statement is different than the order contained within the parameter list, the compiler will return a syntax error.
**Key Concept:** The number of module input parameters, the data type of the input parameters and the order of the module input parameters is critical when passing data to a module. If any of these three rules are violated the parameters identified in the module header, the calling statement will return a syntax error.

Below I have created examples of modules in pseudo-code and in PYTHON. The pseudo code is extremely valuable and planning out the logic. PYTHON will allow you to test that logic to ensure that it’s working correctly.

```pseudo
start
  totCost equals 0
  input State Code into stateCode
  call calcSalesTax(stateCode)
end
```
```
calcSalesTax(inStateCode)
  if inStateCode = “AZ” then
    totCost equals totCost times 1.08
  return
```

- The module is called (invoked) with the call keyword. The module has one argument which in turn is passed to the module and accepted into the module parameter list.
- The `calcSalesTax()` module accepts the parameter as its own local variable saved as `inStateCode`. 
In this pseudo code example, the module was called after collecting stateCode input from the keyboard. The call statement is passing a single parameter to the module. This parameter is a variable which contains a character value. Once the module was called, the value stored within stateCode is accepted into the module and stored within a new variable called inStateCode. The variable inStateCode is local to the sales tax module and cannot be seen by the main program. Many programmers will choose a name for the input parameter variable that matches the variable being sent in the call statement. This is potentially dangerous because in this example stateCode is a global variable and the stateCode variable identified in the module parameter list would be local to the module. For most languages, this scenario would yield two different variables and depending on the logic in the module, this situation could lead to unpredictable results if the programmer is not careful keeping track of each similarly named variable.
**In Practice** - Terminology Parameter versus Argument - It is common in many programming languages to see the terms parameter in arguments used when talking about modules. It’s important to understand, that even those these terms may sound the same, they’re in fact very different. When we call a module, we insert arguments in the calling statement to identify the values we want to transfer to the module. In modules, we create parameters to accept the values sent to us as arguments. Arguments are used to identify values that are passed when we call a module and parameters are the variables we store those values into as we bring them into the modules logic.

**RETURNING A VALUE FROM A MODULE**

Aside from a module just running a block of code, a module can also return a value. Let’s take the same sales tax example and change the logic so that rather than updating the global totCost variable, it is passing back the value stored in the local variable newTotCost. As mentioned earlier, this is the preferred approach since the solution is implemented with local variables. Our previous solution used a global variable which would complicate reusing this module and other programs, since any program that used this module would also have to implement the global variable totCost. Please note the return statement has identified the value that will be passed back to the call point. In this case, the value is stored in the variable newTotCost.

Let’s look At the pseudo code first:

```
start
  totCost equals 0
  input State Code into stateCode
  totCost = calcSalesTax(stateCode)
end

calcSalesTax(inStateCode)
  newTotCost = 0
  if inStateCode = “AZ” then
    newTotCost equals totCost times 1.08
  return newTotCost
```

- The PYTHON statement is an assignment which calls the calcSalesTax() module with a argument (statecode). This is accepted as a local variable within the calcSalesTax module.
This module is different from the modules covered so far. This module includes a return statement which will sent

```
start
  totCost equals 0
  input State Code into stateCode
  totCost = calcSalesTax(stateCode)
end

calcSalesTax(inStateCode)
  newTotCost = 0
  if inStateCode = "AZ" then
    newTotCost equals totCost times 1.08
  return newTotCost
```

Figure 9: Graphic showing return statement returning module value back to call point.

**Key Concept:** The logic model in many cases is a perfect world solution. It is language neutral and should never be designed with any one language in mind. When implementing logic into a specific programming language, you may find that you will need to create program logic that is different then the logic model. This is an acceptable and normal response to implementing logic. You should never feel that the logic model is the end all solution that can not be modified.

**Programming Tip:** A module can optional have input parameters and return values but it is also acceptable to have neither.
When working with modules, it is very important that you understand how to read the module header. The module header leaves important clues as to how to correctly use the module within your program. As I mentioned earlier, many modules used in some programming languages are available only as machine code dynamic link library files (i.e. .DLL file extension). This makes looking at the source code within the module unavailable. If you have the header however, you have all the information necessary to use the module and understand any value it may return. For starters, the header contains the module name. This is important because you need the module name to call the module. The module header also includes the number, order and data type of the parameters used, if the module accepts input parameters. Finally, most languages in their programming reference will also identify the data type of the value returned by the module. In some cases, as we have seen earlier, the module may not return any value at all.

**PYTHON Tip:** PYTHON has the ability to store compiled modules that can be called from other PYTHON programs with the import keyword. These modules are pre-compiled as byte code files. The source code files are regular PYTHON source code files and may or may not be available for the programmer to view (abstraction). This is how PYTHON supports code reuse and abstraction.

In Practice: What’s a void anyway? - Many languages expect their modules to always contain the data type of return value even if the module returns no value at all. Enter the void keyword. Void is used by many languages to indicate that the module will return no information. Void is largely a placeholder and a message to the programmer and the compiler/interpreter that this module does not return a value.

**CASE STUDY - CACTUS BOOKS AND TAPES**

Given your hopefully newfound appreciation for organizing code and modules, you have decided to revisit some of the work done in the previous chapter for the Chavez sister’s book business. Your thinking is that better organized code will make the programs easier to maintain and perhaps improve the quality of your solution. You also suspect that there are modules in this code that might come in handy in other programs that will need to be developed as part of the entire order entry system.
In the last chapter, our case study had two parts. In this chapter, we will return our focus to the Part A problem and reevaluate the initial pseudo code and PYTHON code solution. We need to identify those code statements which represent subtasks and would be good candidates for modularization.

**Pseudo Code logic solution before modularization**

```
Start
  Input WSCost
  markup equals 1.25
  handling equals 5
  loyaltyBonus equals 0
  quantityDiscount equals 0
  Input custLoyalStatus
  If custLoyalStatus = true then
    loyaltyBonus equals 1.5 per cent
  Input Qty
  if Qty >=3 then
    quantityDiscount equals 10
  SubTotal equals WSCost times markup plus handling
  custDiscount equals subTotal times loyaltyBonus less quantityDiscount
  custCost = subTotal less custDiscount
  Print custCost
End
```

**Pseudo Code logic solution after modularization**

```
Start
  call declareVariables()
  Input WSCost
  call customerLoyalty()
  call customerQty()
  Print salesCalcs(WSCost)
End

declareVariables()
  markup equals 1.25
  handling equals 5
```
loyaltyBonus equals 0
quantityDiscount equals 0
custCost equals 0
return
customerLoyalty()
    Input custLoyalStatus
    If custLoyalStatus = true then
        loyaltyBonus equals 1.5 per cent
    return
customerQTY()
    Input Qty
    if Qty >= 3 then
        quantityDiscount equals 10
    return
salesCalcs(newWSCost)
    SubTotal equals newWSCost times markup plus handling
    custDiscount equals subTotal times loyaltyBonus less quantityDiscount
    custCost = subTotal less custDiscount
return custCost

- Start of the pseudo code with its four modules are defined. Notice, that each of these modules is a sub-task of the entire program.

In our new solution, we have identified four sub-tasks the qualify as pseudo-code modules. We have identified a module for declaring our logic variables as declareVariables() and a second module called customerLoyalty() that will contain the code necessary to calculate the customer loyalty discount. A third module called customerQTY() which collects how much of the product that customers ordered and finally salesCalcs() which takes care of all of the calculation logic.
In Practice: Final Refinement (the missing PDP phase?) - We have certainly spent a lot of time in this manuscript explaining the importance of following the process. There is a step that most programmers always perform that doesn’t always show up in the programming development cycle. This step is called refinement. When a programmer does refinement he takes his finished program and looks at it one last time for code modifications that may make the program easier to maintain or possibly run faster. These changes are made with the confidence of knowing that the fundamental logic is sound and has been tested. These last changes are partly cosmetic and partly refinement. It is a step similar to something done by any craftsmen, who during his final inspection, finds a slight defect that, he would like to repair. Many times this type of change is subtle and has only marginal benefits but dedicated programmers want their programs to be as perfect as time allows and reflect positively on their ability.

PUTTING IT TO USE

The intent of this section is to introduce the new programmer to standards and techniques frequently used by professional programmers. Topics in this section relate to concepts introduced in the chapter but with a more vocational or occupational focus for students considering a career in programming.

ADDITIONAL TOPICS: THE MAIN PROGRAM... THE MAIN METHOD... THE MAIN FUNCTION...

Inside this chapter, I have referred to the main program on several occasions to identify a place in the program that controls the flow of instructions and calls to modules. Main is a term used frequently in programming. In the C family of programming languages, which includes Java, A main function/method found within a program represents the programming starting point. This may module acts like a conductor or an orchestra. It determines the sequence of where the program must
visit to implement the solution. It directs the control of the program and where it needs to execute next.

What you get for free ... a program language application programming reference (API). Java and Microsoft .NET Framework both provide extensive libraries of built-in classes all of which contain methods (again another name for a module). For Java, the list of methods that are available to call is literally in the thousands. All of these methods could provide very important functions and features to your programs. If you buy into code reuse and work not to reinvent the wheel by creating your own version of programs, when a version has already been created, you need something that will tell you what’s available as a built-in module. Both Java and.NET Framework provide extensive help and documentation through their products API (Application Programming Interface) documentation. This documentation contains indices and search tools to locate modules by their function. Once a candidate for a particular task is located, you can look at the module header to determine how to correctly implement it. The header we’ll give you the number of inputs, the sequence of the inputs and the data type of the inputs along with information on any returned value. Most of us will reuse modules that have been successfully implemented and other programs. You’ll tend to gravitate towards and remember names of built-in modules that have worked well for you in the past. It’s important not to become too comfortable and complacent since new built-in modules are created with each new release of a programming language.

CHAPTER REVIEW

Chapter summary, highlights, key terms, short answer questions, quizzes and case studies to reinforce topics covered in this chapter.

CHAPTER SUMMARY

After finishing this chapter you should understand and in some cases, demonstrate the following topics:

• Define procedural programming techniques and its implications on logic design
  o Procedural programming techniques add an element of organization to programs that make the program easier to maintain.
Procedural programming allows us to take complex programs and break them into smaller easier to solve pieces.

- Define modularization and describe its benefits
  - Modularization identifies blocks of code in our program logic that stand alone as sub-tasks so that we can organize them as modules. This organization makes the code design more efficient.
  - Modules make the program easier to design, easier to maintain, and easier to reuse logic and programming code.

- Describe identifier and module scope and the implications of global, local and block scope designations.
  - Global, local and block scope allow programmers to more efficiently manage memory and better design reusable programming code by defining the lifetime and accessibility of identifiers and modules.
  - Global scope has a lifetime of the entire program and can be accessed by any statement or module within the program. Local scope is reserved for module variables and makes them available only for the lifetime of module members exclusively. Block scope is reserved for structure variables and have a lifetime equal to the structure block and are accessible only by statements within the structure.

- Describe the process of identifying code to modularize
  - Modularizing program logic is a stepwise process in which the programmer first identifies standalone sub-tasks that is called by the main module. The main module controls the program flow by directing the calling of statements and modules necessary to come up with a program solution.
  - Once modularization is complete, it is useful to diagram the modules and how they’re called via a structure chart. The structure chart is like an organizational chart except it diagrams the calling sequence and relationship between logic modules. At the top of the structure chart, is the main module.

- Recognize the importance of module abstraction and reuse
  - Abstraction and reuse are critical factors in saving time and therefore money. By reusing code that has been developed previously, we avoid creating that same code again in what could be a time wasting process. Many reusable modules have their details hidden (abstraction) but programmers are informed of module parameters and the outputs returned by the module so that these modules can be incorporated into their programs.

- Demonstrate how called modules can accept parameters and return a value
  - Modules use parameters to provide a mechanism for inputting values into local module variables. The calling statement must match the input parameters in number, data type and sequence.
- Modules can optionally return a single value back to the call point if the module uses a return statement to initiate the passing back of a value.
- The module header contains all the information necessary to use a method. Because of the abstraction, understanding the details behind the module is not required. Understanding the module header is crucial to using the module.

- Recognize how different programming languages implement modules.
- Different languages use different keywords to implement modularization. These languages use parameters and arguments in much the same way we have used pseudo code and PYTHON. Other programming languages define modules with different names like function, method and procedure.

CHAPTER KEY TERMS

Abstraction
Application Programming Interface
Block scope
Global scope
Local scope
Main
Modifier
Modularization
Module header
Module
Passing of data values
Procedural programming
Returning a data value
Scope
Structure charts
Syntax coloring
Void