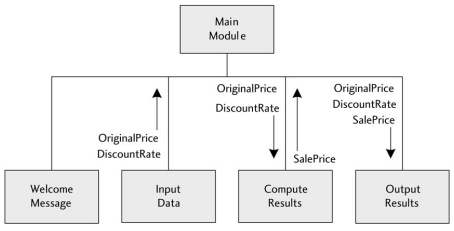
Chapter 9
Program Modules, Subprograms, and Functions

9.1 Data Flow Diagrams, Arguments, and Parameters

- How is data transmitted between program submodules (or subprograms)?
  - **parameters** allow the program to transmit information between modules
  - **data flow diagrams** keep track of the data transmitted to and from each subprogram
- If a data item in the main program is needed in a subprogram, its value must be **passed to**, or **imported** by that subprogram.
- If a data item processed by a subprogram is needed in the main program, it must be **returned to**, or **exported** to that module.
- We say that we **pass a value to a subprogram** and that subprogram may or may not **return a value to the calling program**.

Data Flow Diagram [using the Sale Price Computation Problem from the text]



Parameters and Arguments

Suppose we have designed a module whose purpose is to output the results of some calculations.

- We need to pass the results of the calculations (the data) to that module so that it can display the output.
- The syntax for defining a module that accepts data (**has parameters**) is shown below:

```
Call Subprogram Sub_Name (Var1, Var2, Var3)
```

- The syntax for calling such a module and how to pass in the data (**arguments**) is shown below:

```
Sub_Name (VOne, VTwo, VThree)
```

- Notice that the names of the variables that are used in the calling module do not need to be the same (and probably should not be the same) as the names in the called module.

Parameters and Arguments (continued)

When a subprogram is called, the values of the arguments are assigned to corresponding parameters, based on the order of appearance in the two lists.

For example, when the subprogram with the header:

```
Subprogram Output_Results (OldPrice,Rate,NewPrice)
```

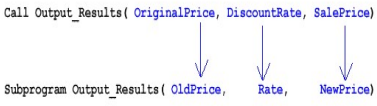
is called by the statement

```
Call Output_Results (OriginalPrice,DiscountRate,SalePrice)
```

the following occurs:

- The value of the 1st argument (`OriginalPrice`) is assigned to the 1st parameter (`OldPrice`).
- The value of the 2nd argument (`DiscountRate`) is assigned to the 2nd parameter (`Rate`).
- The value of the last argument (`SalePrice`) is assigned to the last parameter (`NewPrice`).

Sale Price Program Example (continued)



Parameters and Arguments (continued)

The subprogram `Exam` displays the time and place of your History Final. The Final is at 9:00 o'clock in Room Number 3. See what could happen if the arguments are not passed in correct order!

```
Call Exam(9, 3)
Subprogram Exam (time, room)
    Write "Your History Final"
    Write "will be at"
    Write time + "o'clock in"
    Write "Room Number " + room
End Subprogram
```

Display:

```
Your History Final
will be at
9 o'clock in
Room Number 3
```

```
Call Exam(3, 9)
Subprogram Exam (time, room)
    Write "Your History Final"
    Write "will be at"
    Write time + "o'clock in"
    Write "Room Number " + room
End Subprogram
```

Display:

```
Your History Final
will be at
3 o'clock in
Room Number 9
```

Why Use Arguments and Parameters?

- **To ehance the usefulness of subprograms**
  - They can be designed and coded independently of the main program and used in several different programs, if desired.
  - Only the *structure* of the subprogram is important; not the naming of its variables.
- **To make it easier for different programmers to design and code different subprograms**
  - The programmer of a particular subprogram only needs to know what kinds of variables are transmitted to or from that module.
  - The programmer does not need to be concerned about how variables are named or used in the main program or in another subprogram.
- **To make testing and debugging easier**
  - Subprograms or submodules are independent of the main program.

Assigning Data Types to Parameters

- The data type of a parameter must be defined in the subprogram header
- The data type of the argument sent in to a variable must be the same as the data type of that variable in the subprogram header.
- Pseudocode to assign data types to parameters (used in this book):

  The following syntax declares a subprogram with 3 variables – a `String` variable, an `Integer` variable, and a `Float` variable:

```
Subprogram Sub_Name(String Var1,Integer Var2, Float Var3)
```

Example: Using a Subprogram to Format Output

This program displays a sequence of asterisks (*****) before and after a message.

```
Main
      Declare Message As String
      Write "Enter a short message: "
      Input Message
      Call Surround_And_Display(Message)
End Program
Subprogram Surround_And_Display(String Words)
      Write "***** " + Words + " *****"
End Subprogram
```


9.2 More About Subprograms

- Parameters can be passed **by value** and **by reference**.
- Data may be sent from the call (main program) to the subprogram. This data is **passed** or **imported** to a subprogram.
- Data may also be passed from the subprogram back to the call. This data is **returned** or **exported** to the main program.
- The correspondence between arguments and parameters does not indicate which way the data is flowing.

Value and Reference Parameters

- **Value parameters** have the property that changes to their values in the subprogram ***do not affect*** the value of the corresponding (argument) variables in the calling module. These parameters can only be used to import data into a subprogram.
- **Reference parameters** have the property that changes in their values ***do affect*** the corresponding arguments in the calling module. They can be used to both import data into and export data from a subprogram.

To Pass by Value or Pass by Reference?

[That is the question!]

- **Pass by value**: When a variable is passed by value to a submodule, the submodule only receives a **copy** of that variable.
  - A separate storage location is created and the value of the variable is stored in that location.
  - Therefore, any changes made to that variable in the subprogram do not affect the original variable.
- **Pass by reference**: When a variable is passed by reference, the submodule receives the **actual storage location** of that variable.
  - Therefore, changes made to the variable in the subprogram are also made to the original variable.

Example: Passing by `Val` and Passing by `Ref`

```
1   Main
2           Declare MyNumber As Integer
3           Declare YourNumber As Integer
4           Set MyNumber = 156
5           Set YourNumber = 293
6           Call Switch_It(MyNumber,YourNumber)
7           Write MyNumber + " " + YourNumber
8   End Program
9   Subprogram Switch_It(Integer Number1, Integer Number2 As Ref)
10          Set Number1 = 293
11          Set Number2 = 156
12  End Subprogram
```

`Number1` is passed by value, and it *is not* changed by the subprogram.
`Number2` is passed by reference and it *is* changed by the subprogram.

This program prints: `156 156`

The `ToUpper()` and `ToLower()` Functions

- When a `String` value or variable is placed inside the parentheses of the `ToUpper()` function, all characters in that string are converted to uppercase.
- Similarly, when a `String` value or variable is placed inside the parentheses of the `ToLower()` function, all characters in the string are converted to lowercase.
- These functions are helpful:
  - Allow users to type upper or lower case responses without getting errors
  - Create usernames or other identification techniques
  - And more…

Example: Using `ToUpper()` and `ToLower()`

```
1  Declare Response As Character
2  Declare Word, Box As String
3  Declare Count, X As Integer
4  Write "Do you want to draw a word-box? Enter 'Y' or 'N"'
5  Input Response
6  While ToUpper(Response) == "Y"
7      Write "Enter any word: "
8      Input Word
9      Set X = Length_Of(Word)
10     Set Box = ToLower(Word)
11     Set Count = 1
12     While Count <= X
13           Write Box
14           Set Count = Count + 1
15     End While(Count)
16     Write "Create a new box? Enter 'Y' or 'N'"
17     Input Response
18 End While(Response)
```


Be Careful When You Pass Variables Around!

Natalie and Nicholas are co-presidents of the Gamers at College club (GAC). They have created a web site and they want the site to be secure. Nick suggests that each member should have a secret login name and Natalie offers to write a program to achieve this. Unfortunately, Natalie did not study this chapter carefully and does not understand the difference between value parameters and reference parameters. She writes the pseudocode shown in the next two slides.

(continued on the next slide →)

Passing Variables Carefully, (continued): `Main` program

```
1  Main
2      Declare Response, First, Last As String
3      Write "Do you want to start? Enter 'yes' or 'no' :"
4      Input Response
5      Set Response = ToLower(Response)
6      While Response == "yes"
7            Write "Enter this member's first name:"
8            Input First
9            Write "Enter this member's last name:"
10           Input Last
11           Call Secret_Login(First, Last)
12           Write "Member name: " + First + " " + Last
13           Write "Enter another member?"
14           Input Response
15           Set Response = ToLower(Response)
16     End While
17 End Program
```

(subprogram is continued on the next slide →)

Passing Variables Carefully, (continued): `Secret_Login` subprogram and results

```
18 Subprogram Secret_Login(Part1 As Ref, Part2 As Ref)
19     Declare Login, Temp As String
20     Set Temp = Part1
21     Set Part1 = ToLower(Part2) + "**"
22     Set Part2 = ToLower(Temp)
23     Set Login = Part1 + Part2
24     Write "Your secret login is: " + Login
25 End Subprogram
```

Natalie runs the program and enters the name `Mary Lamb`. When she sees the display she realizes what happened:

```
Your secret login is: lamb**mary
Member name: lamb** mary
```

How can this be fixed? Change line 18 to:

```
Subprogram Secret_Login(Part1, Part2)
```


The Scope of a Variable

- **scope of a variable**:
  - The part of the program in which a given variable can be referenced is called the scope of that variable.
- **global variables**:
  - Global variables are those variables declared in the main program.
  - They are available to all subprograms and submodules.
- **local variables**:
  - Local variables are declared in a particular subprogram.
  - They are said to be local to that subprogram or module.
  - They can only be used inside that subprogram or module.

Properties of Local Variables

- **Local variables** have the following properties:
  - When the value of a local variable changes in a subprogram, the value of a variable with the same name outside that subprogram remains unchanged.
  - When the value of a variable changes elsewhere in a program, a local variable with the same name remains unchanged in its subprogram.

Using Local and Global Variables to Keep Track of `MyNumber`

```
1   Main
2     Declare MyNumber As Integer
3     Set MyNumber = 7654
4     Call Any_Sub()
5     Write MyNumber
6   End Program
7   Subprogram Any_Sub()
8     Declare MyNumber, YourNumber As Integer
9     Set MyNumber = 2
10    Set YourNumber = MyNumber * 3
11    Write YourNumber
12  End Subprogram
```

Output:
```
6
7654
```


Using Counters Locally:

Note that **`Count`** is used in both `Main` and `Pay_Employee` but, because it is declared locally in the subprogram, its value in the subprogram does not affect its value in `Main`.

```
1   Main
2       Declare Name As String
3       Declare NumEmployees, Count As Integer
4       Write "How many employees do you have?"
5       Input NumEmployees
6       For(Count = 1; Count <= NumEmployees; Count++)
7               Write "Enter this employee's name: "
8               Input Name
9               Call Pay_Employee(Name)
10      End For
11  End Program
12  Subprogram Pay_Employee(String EmpName)
13      Declare Rate, Hours, Sum, Pay As Float
14      Declare Count As Integer
15      Set Sum = 0
16      Write "Enter the pay rate for " + Name
17      Input Rate
18      For(Count = 1; Count <= 7; Count++)
19              Write "Enter hours worked for day " + Count
20              Input Hours
21              Set Sum = Sum + Hours
22      End For
23      Set Pay = Sum * Rate
24      Write "Gross pay this week for " + EmpName + "is $ " + Pay
25  End Subprogram
```


9.3 Functions

A **function** is a subprogram whose name can be assigned a value. This allows the calling program to use the name of the function in an expression. Some examples were already introduced in earlier chapters. These functions, provided by the programming language, are called built-in functions.

- `Sqrt(X)` computes the square root of the positive number **`X`**.
- `Int(X)` computes the integer obtained by discarding the fractional part of the number **`X`**.
- `Ceiling(X)` computes the integer obtained by rounding the number **`X`** up to the next integer.
- `Floor(X)` computes the integer obtained by discarding the fractional part of the number **`X`**.
- `Random` generates a random integer (whole number) from 0.0 to 1.0, including 0.0 but not 1.0.
- `Length_Of(S)` computes the length of the string **`S`**.
- `ToUpper(S)` changes the value of all characters in a string, **`S`**, to uppercase.
- `ToLower(S)` changes the value of all characters in a string, **`S`**, to lowercase.

Built-in Functions

Most programming languages provide **built-in functions**. The code for these functions are supplied in separate modules (often called a **library**) and can be used by any programmer.

Some examples of built-in functions that we have not yet seen are:

- `Abs(X)` computes and returns the absolute value of **X**. It is of type `Float`.
- `Round(X)` rounds the real number, **X**, to the nearest whole number. It is of type `Integer`.
- `Str(X)` converts the number **X** to a corresponding string. It is of type `String`.

Examples Using Built-in Functions

- `Abs(10)` returns `10`
- `Abs(-10)` returns `10`
- `Round(10.6)` returns `11`
- `Round(100 * 10.443)/100` returns `10.44`
- `Str(31.5)` returns `"31.5"`
- `Str(-100)` returns `"-100"`
- `Val("31.5",N)` returns the number `31.5`, **N** = `1`
- `Val("abc",N)` returns the number `0`, **N** = `0`

User-Defined Functions

User-defined functions are created by the programmer.

The differences between a function and a subprogram are:

1. A function's name may be assigned a value in the code that defines it.
1. A function is called by placing its name with its arguments anywhere in the program where a constant of the function's type is allowed.

Example: The Cube Function

Since function calls evaluate to some value, the definition of the `Function` must include which **data type** the function evaluates to.

The following pseudocode shows the syntax for defining a `Function` that finds the cube of a number:

```
Main
      Declare LittleBox As Float
      Set LittleBox = Cube(10)
      Write LittleBox
End Program
Function Cube(Side) As Float
      Set Cube = Side^3
End Function
```


When to Use a Function

- If the programming language you are using contains both functions and non-function subprograms, then either one may be used to implement a given program submodule.
- If the submodule computes and returns a single value to the calling module, then implement it with a function.

Example: Getting Good Mileage Out of a Function

The following program segment will compare the cost of several road trips by calculating miles per gallon used on each trip. By identifying the type of trip, it will compare highway miles and city miles.

The output required is a table that identifies the specific type of road trip, the total miles of each trip, and the miles per gallon used for each trip. Parallel arrays, as follows, are used to store this information:

- A `String` array: **`TripName[10]`**
- Two `Float` arrays: **`TripMiles[10]`** and **`TripMPG[10]`**

After the information for each trip is entered, the program will calculate the miles per gallon. A user-defined function named `Answer()` will be created to do this. It will receive the number of miles traveled and the number of gallons of gas used. The result of the `Answer()` function will be stored in the appropriate element in the **`TripsMPG[]`** array.

Assume the following variables have been declared in the beginning of `Main`, as well as the three arrays previously mentioned:

- **`Count`** and **`K`** as `Integer` variables
- **`Name`** as a `String` variable
- **`Miles`** and **`Gallons`** As `Float` variables

Continued on next slide ⇩

Getting Good Mileage Out of a Function (continued)

```
1  Main
2       Set Count = 0
3       While Count < 10
4               Write "Enter a description of this trip: "
5               Input Name
6               Set TripName[Count] = Name
7               Write "How many miles did you drive? "
8               Input Miles
9               Set TripMiles[Count] = Miles
10              Write "How many gallons of gas did you use on this trip?"
11              Input Gallons
12              Set TripMPG[Count] = Answer(Miles, Gallons)
13              Set Count = Count + 1
14      End While(Count)
15      Set K = 0
16      Write "Trip Name \t Miles Traveled \t MPG"
17      While K < 10
18              Write TripName[K] + "\t" + TripMiles[K] + "\t" + TripMPG[K]
19              Set K = K + 1
20      End While(K)
21  End Program
22  Function Answer(Float Num1, Float Num2) As Float
23      Set Answer = Num1/Num2
24  End Function
```


9.4 Recursion

A **recursive** subprogram is one that calls itself.

Some languages do not permit **recursion**.

Recursive subprograms may be an effective way to provide a solution for a specific problem.

Problems that are solved with recursion are those that can be easily described in terms of themselves.

Example: the sum of the first **N** integers can be written as:

```
Sum(N) = Sum(N - 1) + N
```


Example: Recursive Code to Sum N Positive Integers

```
Function Sum(N) As Integer
     If N = 1 Then
          Set Sum = 1
     Else
          Set Sum = Sum(N - 1) + N
     End If
End Function
```

Recursive call to Sum

Tracing recursive code is difficult at first. The recursive call must be conditional, or there will be an infinite recursion. This is similar to a looping construct, but no repetition is used; there are only function calls to the function we are defining.

Recursive Code to Sum N Positive Integers (continued)

Suppose that this function is called by the statement

```
Set Total = Sum(4)
```

(**Total** is an `Integer` variable)

- First call to the function: **N** = 4: Control goes to `Else` clause and line 5 is executed to get:

```
Set Sum = Sum(N – 1) + N → Set Sum = Sum(3) + 4
```

- Second call to the function: The `Else` clause is executed now with **N** = 3. (**N** – 1) = 2, so we get

```
Sum(2) + 3
```

which causes the function to be called again with **N** = 2.

- Third call to the function: The `Else` clause is executed with **N** = 2. The right side of the assignment statement is evaluated, giving:

```
Sum(1) + 2
```

- Fourth (and last) call to the function: Since **N** = 1, this time the `Then` clause is executed and `Sum` is set equal to `1`. Now the function does not call itself and execution of this call to the function is completed.

Continued on next slide

Recursive Code to Sum N Positive Integers (continued)

- Control now returns to the assignment statement on line 5 as:

```
Set Sum = Sum(1) + 2
```

in the third call to the function (where the last call to the function was made).

- Here `Sum(1)` is replaced by `1` and `Sum` (on the left side) takes on the value `3`. Execution of the third call is now complete.
- Control now returns to the assignment statement as:

```
Set Sum = Sum(2) + 3
```

in the second call to the function. Here `Sum(2)` is replaced by `3` and `Sum` (on the left side) takes on the value `6`. Execution of the second call is now complete.

- Finally control returns to the assignment statement as:

```
Set Sum = Sum(3) + 4
```

in the first call to the function. Here `Sum(3)` is replaced by `6` and `Sum` (on the left side) takes on the value `10`. Execution of the first call is now complete and **Total** (in the initial calling statement) is set equal to `10`.