Introduction to

BASIC Programming for Foresters using Liberty BASIC

by

John R. Brooks
Division of Forestry
West Virginia University
(jrbrooks@wuv.edu)

and

Harry V. Wiant, Jr.
School of Forest Resources
Penn State University
(hvw3@psu.edu)

ROGRAMMING EXAMPLE

1. In several high-level languages, this statement is legal Z = X + Y
2. In assembly language it might be:

```
LOAD  X
ADD   Y
STORE Z
```

3. Machine language might be:

```
0010   0000   0000   0100
0100   0000   0000   0101
0011   0000   0000   0110
```

INTRODUCTION

Variations of BASIC.

This text was developed to introduce foresters to computer programming techniques using the BASIC programming language. Foresters have always had a need for repetitive calculations in their everyday operations. In forest inventory, early calculations were done using adding machines until the advent of the pocket calculator in the early 1970's. Many weary foresters have spent hours punching numbers on a hand calculator to compute volume totals and cruise statistics.

Since the release of BASIC by Kemeny and Kurtz of Dartmouth College in 1965, there have been many versions of the BASIC language in use throughout the United States. The authors published a book similar to this one in 1986 based on MBASIC. A windows version, Liberty BASIC, is used in this book. It can be downloaded from the internet and upgraded to facilitate creation of stand-alone programs for a very reasonable fee.

WHY LEARN BASIC?

There are many computer languages from which to choose, including FORTRAN, Pascal, C++, and BASIC, to name a few. Why then select BASIC as an introductory language? BASIC was designed as an educational tool, emphasizing ease of learning with a language that is very similar to English. These factors have played a key role in the popularity of this language. In addition, many programmable pocket calculators and early computers were sold with "built-in BASIC." For example, the early Texas Instruments and Commodore computers were sold with BASIC built into memory, available to the user when the machine was turned on. In addition, many computers were sold with "packaged software" which often included some version of BASIC. . This is not to imply that BASIC is the best language to use for all applications. Each language has its own inherent advantages and disadvantages under certain circumstances. However, it is a language that

is very popular and easy to learn. In the field of forestry today, many application programs were written in BASIC.

Why do I need to learn to program if I plan to use purchased software packages? This is a legitimate question that overlooks the fact that a person who has some background in programming makes a better program user. Some experience in the use of a programming language educates users as to why certain data inputs are illegal and prepares them for dealing with errors that will inevitably occur. In addition, those individuals that have some background in programming soon find themselves writing useful short programs that expedite everyday computations.

WORKING WITH BASIC

Following a RUN command, the program is interpreted one line at a time. For example, the following is a legitimate program:

```
PRINT 4
END
```

The BASIC statement PRINT instructs the computer to print the numeric value of 4 to the screen. The program then advances to the next statement, in this case END which terminates the program. It is also a good practice to terminate your program with an END statement.

Now that you are somewhat familiar with the system, it is time to write your first program. Type the following lines, pressing the return key after each line is entered:

```
A = 17.3
B = 16.1
PRINT A, B
END
```

Run the program and note the output.

NAMING CONSTANTS AND VARIABLES

Variable names are like street addresses.
Assigning a numeric or string value does not a constant make.

Programs foresters write usually contain constants and variables. Much of the following discussion will remind you of writing formulas in algebra. It is helpful to imagine that the computer stores data values at an address somewhere in its memory with the name assigned to the constant or variable. For example, the statement:

```
A = 10.53
```

says store in memory at an address called A the number 10.53.

CONSTANTS

Strictly speaking, the BASIC language does not facilitate the use of true constants. A constant, by definition, is a value that does not change throughout the program. Simply assigning a numeric or string value to a variable name does not, in the truest sense, create a constant, since we can reassign a different value to that same variable name at a different point in the same program. Let's use an example to clarify this point:

```
A = 10.53
A = 7.53
```

In the first statement we assign the variable A the value 10.53. In the next statement , we re-assign the variable A to 7.53, thus replacing its original value. Since the value of A changes within the program it is not really a constant. In practical application, however, we may assign a value to a variable name with no intent to alter its value, thus using it as if it were a constant. In this light, the following program statement might be used to define the constant pi:

```
PI = 3.1416
```

Later in the program when the value of pi is needed, the constant name PI may be used in place of its numeric value:

```
CIR = PI * R^2
```

Numeric constants are used more often than "string" constants, which we will discuss in more detail later. However, one might write a program where a landowner's name will be used several times, and the following constant might be useful:

```
NAM$ = "WILLIAM STEPHEN HARRISON"
```

To print the landowner's name later in the program would require only:

```
PRINT NAM$
```

The $-sign at the end of a constant indicates a string. The characters used in a constant (or variable) name must be connected; that is, there can be no spaces. TOP DIAM would not

do for a numerical constant or LAST NAMES for a string constant, but TOPDIAM and LASTNAME$ are acceptable.

VARIABLES

It is good practice when naming variables in a program to use names which give a clue to the identification of the variable. A variable identifying the circumference of a circle serves as a practical example. Consider the following options:

```
CIRCUMFERENCE = PI * R^2
CIR = PI * R^2
C = PI * R^2
```

The middle choice, CIR, is perhaps a more reasonable compromise between the ideal and the too brief. Words used to perform certain actions, such as PRINT, cannot be used as constant or variable names when programming in BASIC. These are referred to as reserved words and include BASIC commands, statements, and functions. You should become more familiar with these as you use the language. Variable (or constant) names must start with an alphabetic character. Note that in Liberty BASIC capital letters are different variables than lower-case letters (A is different than a, for example).

Variable names should be meaningful.
Caution - some names cannot be used.

REVIEW PROBLEMS

1. Which of the following variable names are incorrect?

(a) D B H (b) 25H (c) H5 (d) F$ (e) PRINT

2. What is wrong with the following statement: F = "John Henry"

ARITHMETIC OPERATIONS AND COMPARISONS

ARITHMETIC OPERATORS

Forestry programs usually involve mathematical operations, and the operators used in BASIC are generally those used in everyday calculations. They are:

```
+ addition, as Y = A + B
- subtraction, as Y = A - B
* multiplication, as Y = A * B
/ division, as Y = A / B
^ exponentiation, as Y = A + B^2
```

In any given equation, operations are processed from left to right in the following order; exponentiation, multiplication and division, and addition and subtraction. That order can be changed by the use of parentheses. For example, consider the following statement calculating the cordwood volume of some Appalachian hardwood trees:

```
VOL = (.172 * D^2 - 2.56)/90
```

In this case D will be squared, then multiplied by .172, followed by subtraction of 2.56, with the answer divided by 90. Without the parentheses, D would be squared, then multiplied by .172, followed by the answer being reduced by the quantity 2.56/90, obviously yielding a different answer than desired. Although not a perfect memory prompter because multiplication and division or addition and subtraction have equal precedence, the general order of operations can be remembered by the phrase "P"lease "E"xcuse "M"y "D"ear "A"unt "S"ally, which stands for parentheses, exponentiation, multiplication, division, addition, and subtraction, respectively.

RELATIONAL AND LOGICAL OPERATORS

To control the path of operations in a program, or the program "flow", we may want to compare two variables to determine if they are:

```
= equal, as X = Y
<> not equal, as X <> Y
< less than, as X < Y
> greater than, as X > Y
<= less than or equal to, as X <= Y
>= greater than or equal to, as X >= Y
```

REVIEW PROBLEMS

1. Calculate the answer of Y = 3+3^A2/3-4*2+l.

2. Calculate the answer of Y = (3+3^2)/3-(4*2+1) .

ASSIGNMENT STATEMENTS

As implied by its use in previous chapters, the equal sign *(=)* has a somewhat different meaning in BASIC that in ordinary arithmetic. It is used to assign the value derived from the expression on the right side of the equal sign to the address somewhere in the computer's memory which is identified by the variable name on the left side of the equal sign, as:

```
X = 45.7
```

The following form, unlike algebra, is not legal:

```
45.7    = X
```

PRINT STATEMENTS

A program statement may be written to calculate a value, such as:

```
 L = 40
 W = 100
AREA = L * W
```

After the last line has been executed, the product of L * W is stored in the computer's memory at an address we have called AREA. In a sense, the computer knows the answer but has not told us. We can have the answer printed to the screen using the statement:

```
PRINT AREA
```

When we run the program, 4000 will appear on the screen. It would be helpful to have the answer identified on the screen, as:

```
PRINT "Area = "; AREA
```

Upon execution of this statement, we will see on the screen:

```
Area =  4000
```

If you want the answer printed on paper rather than displayed on the screen, use LPRINT in place of PRINT (a printer must be attached to your computer, and the default printer is used).

FORMATTING OUTPUT

Semicolons and commas control spacing.
Statements for specifying output appearance.

Try: **PRINT 1/3**

Note the output is carried to several decimal places. To print to a given number of digits, one can us a template:

PRINT USING(templateString, numericExpression)

such as:

PRINT USING("#.###",1/3)

giving: 0.333

Note that **PRINT USING("##.###",23.2366)**

gives: 23.237 as output is rounded.

If we use semicolons instead of commas,

PRINT 30;40;-50

the number are printed with no spaces between them.

The TAB function permits the user to specify where the output occurs on the screen or printer. Up to this point, the only control we had over the output was to use default output fields and designate the form of the numbers with PRINT USING statements. What if we wanted to begin a list of figures in column 12? This is possible by using the TAB function. The statement TAB(12) instructs the screen (or printer) to begin printing at column 12. These statements are commonly used in conjunction with PRINT statements as the following example indicates:

A=5
B=6
C=7
PRINT A; TAB(1) ;B; TAB(9)

Note that A will be printed in column 1 , B in column 9.

REVIEW PROBLEMS

1. How will output from the following statements appear on the

screen?

```
PRINT 20/30;
PRINT 60
```

2. How will the output of **PRINT 78,80** appear on the screen?

DATA INPUT

INPUT STATEMENT

When a program is being run, the INPUT statement is used to accept data entered from the keyboard. The INPUT statement temporarily stops program execution, generates a question mark to the screen, and waits for the user to enter data from the keyboard. The following example will result in a question mark being printed to the screen and waits for an integer to be typed by the user:

```
INPUT G
```

After the value for G is entered and the return (or enter) key is pressed, the value is stored in the memory address for the variable G. The use of prompts within the INPUT statement is helpful to the user, as it labels the question mark, thus providing an idea of what the user is to enter. Consider:

```
INPUT "Name ";n$
print n$
```

The information included within the quotation marks is printed to the screen and the user's input is printed when the program is executed.

DATA STATEMENT

The DATA statement may be used to include data within the program. There is no limit to the number of DATA statements used or restrictions as to their location in the program. Some versions of BASIC require that they appear before the END statement. DATA statements are accessed by READ statements, and data are read one after the other in the order that the DATA statements appear. The following program calculates and prints volume (Int.-1/4") for trees with various diameters and number of logs:

```
DATA 12,1, 14, 1, 20,3
DATA 30,3, 16,2, 15,1.5
[10] READ D, H
V = .366 * D^2 * H
PRINT D, H, V
GOTO [10]
END
```

We get an error message as the program is trying to read past the last DATA statement. A better approach will be introduced later. Also, RESTORE will reset the reading of DATA statements so that the next READ will get information from the first DATA statement.

FILES

Data can be read into a program from a data file, but this is a more complicated procedure, and we will discuss it in a later chapter.

REVIEW PROBLEMS

1. Is the following statement legal (no error message)?

```
INPUT "X, Y, Z"; Z, Y, X
```

2. What would be output from the following program?

```
Z = 0
DATA 1,3,5,100,300
DATA 5
[10] READ X
Z = Z + X
IF X = 100 THEN GOTO [50]
GOTO [10]
[50] PRINT "Z = ";Z

END
```

3. Why is it good programming practice to initialize (set to zero) variables used in summing statements in a program (see statement 5 in question 2)?

2. What would be output from the following program?

```
Z = 0
DATA 1,3,5,100,300
DATA 5, 80
[10] READ X
```

```
Z = Z + X
IF X = 100 THEN GOTO [50]
GOTO [10]
[50] PRINT "Z = ";Z
END
```

3. Why is it good programming practice to initialize (set to zero) variables used in summing statements in a program?

LOOPING

Rapid repetition without mistakes.

Computers make it possible to do repetitive procedures rapidly and without the mistakes human beings would make if they had to do such boring work. Operations that will be executed several times in a program may be placed in a structure called a "loop." We will introduce several types: GOTO [ ], FOR-NEXT, and WHILE-WEND.
Beware of GOTO statement pitfalls.

GOTO [ ]

With the GOTO [ ]statement, the flow of a program from top to bottom is changed, allowing the flow to jump to any statement specified by [ ]. For example, the statement

```
GOTO [40]
```

causes the flow to jump back to the statement labeled [40]. This is a very handy statement, too handy it turns out, and programmers are advised to eliminate the use of GOTO []statements when possible. Their use often causes the flow of a program to jump around so much that a diagram of the flow has been described as resembling a plate of spaghetti. A few weeks after a medium-sized program is written which includes many GOTO [ ] statements, even the author of the program may find it difficult to understand just how the program works (the program logic).

FOR-NEXT LOOPS

Loops can be easy to locate and read.
Close inner loops before closing outer loops.

FOR-NEXT loops are useful when it is desirable to perform a sequence of operations a given number of times. Perhaps the basal area in square feet is needed for trees 10 to 40 inches dbh.

```
FOR D = 10 TO 40 STEP 1
BA = .005454 * D^2
PRINT D, BA
NEXT D
```

This will print the dbh and basal area for trees 10, 11, 12, etc. to 40 inches dbh. *D* in this case is the loop-index variable, 10 indicates the starting value and 40 the ending value, while STEP 1 indicates the interval at which D is to be incremented. Had the basal area of trees of only even-inch dbh's been needed, STEP 2 would have been appropriate. Actually, when the interval is 1, the use of STEP 1 in the statement is optional. The NEXT statement directs the flow of the program to the top of the loop until the terminal value is encountered (in this case 40). Notice that the body of the loop in the previous example is indented. The purpose is to make the loop easier to locate and read. It does not affect the execution of the program.

Loops can be easy to locate and read.
Close inner loops before closing outer loops.

FOR-NEXT loops may be placed within other FOR-NEXT loops. This is called "nesting." Each loop must have a different counting variable. To illustrate this, suppose we want to estimate the board foot volume (Int.-1/4") of trees 12 to 20 inches dbh in even-inch classes with 1 to 4 logs at half-log intervals. Volumes are calculated using the relation V = .366 * D^{A}2 * H, where H = number of 16-foot logs. The following statement could be used to solve this problem:

```
FOR D = 12 TO 20 STEP 2
FOR H = 1 TO 4 STEP .5
  V = .366 *D ^2 * H
  PRINT D, H, V
 NEXT H
NEXT D
```

It should be noted that D = 22 and H = 4.5 when these loops are completed because loop-index variables are incremented before ending values are checked. Loops cannot cross. That is, when using nested loops, the inner loop must be "closed" by its NEXT statement before the outer loop is "closed." Had we used NEXT D followed by NEXT H, an error

message would have been displayed. The numbers used in the FOR statement can be variables or expressions (such as FOR D = X TO Y STEP Z or FOR M = T TO W+2 STEP E/5). To make the output more meaningful, the first statement could be: PRINT "Dbh","Logs","Vol"

WHILE-WEND

WHILE-WEND statements can also be nested.

The WHILE-WEND loop continues until some specified condition is met. Assume we want to accumulate the sum and sum of squares of numbers we will input until a signal number, say -999, is entered.

```
INPUT "X ";X
WHILE X <> -999
   SUMX = SUMX + X

  SUMX2 = SUMX2 + X^2
 INPUT "X, -999 IF DONE ";X
WEND
PRINT "sum of X   = ";SUMX
PRINT "sum of X^2 = ";SUMX2
```

Note that the variable used in the WHILE statement, X, was defined before entering the WHILE-WEND loop, which is good programming practice although not required in BASIC. Once a -999 is input, the program flow will jump to the statement following the WEND statement. WHILE-WEND statements can be nested as was discussed for FOR-NEXT statements, but each will require its own WEND.

REVIEW PROBLEMS

1. Write a program using FOR-NEXT statements to sum all integers between and including 1 to 100.

2. Without a computer, calculate the answer that would be obtained using the following program:

```
SUM = 0
INPUT D
WHILE D < 50
  SUM = SUM + D
```

```
    INPUT D
  WEND
```

An attempt is made to input the following data:

10, *2*, 6, 1, 62, 5, 100

3. Write a FOR-NEXT loop to print the even numbers from 80 down to 60.

CONDITIONAL STATEMENTS

IF STATEMENT

IF statements provide for branching in a program depending on whether an expression is true or false. For instance, if dbh is less than 10 inches, we may wish to calculate volume as pulpwood while volume will be computed as sawtimber for dbh's of 10 inches or more. This could be done by the statement:

```
DBH=8
IF  DBH < 10 THEN
   GOTO [PulpVol]
  ELSE
  GOTO [SawVol]
END IF
[PulpVol] PRINT DBH
```

Rather than direct the program to certain program lines, actions can be specified, as:

```
DBH = 8
IF DBH < 10 THEN PRINT "PULPWOOD" ELSE PRINT "SAWTIMBER"
```

The ELSE statement provides for alternative actions, but is not required. A legal statement is:

```
IF N% = 1 THEN GOTO [calculate]
```

Complex IF statement could be useful.

REVIEW PROBLEMS

1. Write a statement that will send the program flow to statement labeled [400] if dbh is less than or equal to 12 and to statement [600] otherwise.

2. Write a statement that will terminate the program if N = 1.

STRING OPERATIONS

Alphabetic and numeric variables compared.

Alphabetic strings or variables can be compared and manipulated much like numeric variables. Actually, however, these operations are rarely used in most forestry programs, and we will provide only a brief introduction here.

Strings can be concatenated (joined), such as:

```
A$ = "BILL "
B$ = "JONES"
PRINT A$ + B$
```

This would print BILL JONES on the screen. Note the space left inside the quote after BILL. This causes the name to be printed with a blank space between the first and last names.

Strings can also be compared, such as:

```
X$ = "black"
Z$= "white"
IF X$<Z$ THEN PRINT "X$ is first alphabetically"
```

The message will be printed to the screen as X$ is first alphabetically (actually due to the ASCII values). Shorter strings are evaluated as less than longer strings, other things being equal (e.g., "BILL" is less than "BELLY"). There are other built-in functions in BASIC that allow string manipulation? We will introduce LEFTS, RIGHTS, VAL, and STR$ at this point, but refer the reader to their Liberty BASIC Help for more detail. Some simple examples of these functions include:

```
PRINT LEFT$("RED OAK",3)
PRINT RIGHT$(""RED OAK,3)
```

The first statement prints RED to the screen, the second statement prints OAK to the screen.

Numeric values can be expressed as strings or converted back to numeric, such as:

```
ZIP$ = "26506"
NUMBER = VAL(ZIP$)
CHANGE$ = STR$(NUMBER)
```

The first statement converts ZIP$ to a numeric value (NUMBER), the next statement converts NUMBER back to a string (CHANGES).

REVIEW PROBLEMS

1. Write a statement that will print the last 4 letters of "PROGRAMMING".
2. 2. See whether Liberty BASIC considers "a" or "A" as less. (Check the ASCII code and use a program to do this).

SUBROUTINES

Time-saving subroutines can be called with CALL SomeName.

Here we will introduce the concept of modular programming through the use of subroutines, and introduce the GOSUB and RETURN statements. A subroutine is a short program that is called by the main program to execute a specific task. Subroutines provide an excellent means for developing your program code into small modular sections. The main purpose of a subroutine is to provide efficiency when having to repeat a certain calculation throughout the body of the main program. It would be much easier to call a subroutine to do a specific calculation and return the value desired at several points within the program than to write the same lines of code over and over again whenever that specific calculation is necessary. For example, you are writing a program that will compute board foot volume based on a series of volume equations. It should be obvious that it would be more efficient to write the code once as a subroutine and call it when volume calculations are necessary than to write the same code several times throughout the body of the program. The statement that calls the subroutine is GOSUB and has the following format:

GOSUB[SomeName]

This instruction causes the computer to store the next line in a special storage register and then transfer control to a subroutine which begins at [SomeName]. The subroutine is processed line by line until a RETURN statement is reached. At that point, control returns to the statement following the GOSUB call as defined by the location previously stored in the special storage register. Unlike many other languages, BASIC does not require that you

pass variables between the main program and each subroutine. subroutine and vice versa. The variable names inside a subroutine are scoped locally, meaning that the value of any variable inside a subroutine is different from the value of a variable of the same name outside the subroutine. Variables can be made available throughout the program using the GLOBAL statement, as shown later. You may have as many subroutines as needed, call them as often as you like, and place them anywhere in the program. However, care must be taken to insure that general program flow does not inadvertently enter your subroutine. This can be accomplished by locating your subroutine section at the end of the main program following the END statement. BASIC does allow statements following the END statement. Run the following example to see how it works:

```
PRINT  "Here  I  am at  the first line of code"
GOSUB [60]
GOSUB[80]
PRINT "Back to the main program -done!"
END
[60] PRINT "Now I am at [60]"
RETURN
[80] PRINT "Now I am at [80]"
RETURN
```

Let's look at another, more practical example. This time we will use a subroutine to calculate the volume of a tree based on data entered by the user. Try the program with and without the GLOBAL statement!

```
GLOBAL DBH,LOGS,GVOL
 INPUT "ENTER DBH "; DBH
WHILE DBH > 0
   IF DBH <=0 THEN END
   INPUT "ENTER NUMBER OF 16-FOOT LOGS ";LOGS
   CALL vol
   PRINT TAB(10);"INT 1/4 VOLUME = ";GVOL
   PRINT "ENTER A NEGATIVE VALUE FOR DBH TO QUIT "
   INPUT "ENTER DBH ";DBH
WEND
SUB vol
 GVOL = 0.366 * DBH^2 * LOGS
END SUB
RETURN
```

Problems are easier to pinpoint in modular subroutines.

ADDITIONAL USES FOR MODULAR SUBROUTINES

It was mentioned earlier that a principal reason for using subroutines is to avoid repetitive code. There are other, quite valid reasons for writing modular subroutines as well. One reason is that it makes the overall program slightly easier to read, and if problems do come up (and they will), the bugs are easier to locate because you can normally pinpoint the subroutine that is giving you the problem. Second, modifications to your program are much easier if you only have to change a single subroutine rather than rewrite the whole program. One last reason for writing modular subroutines is that you can work on a single section at a time and check its operation and accuracy as you go. When one section is completed, you can start another section and then tie them together at the end. Sometimes this is less overwhelming than trying to write the complete program at once. In addition, once a subroutine works, you can keep a copy of it and use it in other programs you write -why reinvent the wheel!

REVIEW PROBLEMS

1. What statement calls a subroutine? What statement causes it to return?

2. Can you call a subroutine (GOSUB) while currently in a subroutine?

FUNCTIONS

Take short-cuts in calculations with function statements.
Functions are called by name and return a single value.

Up to this point, we have covered several BASIC commands and statements such as PRINT, IF, and FOR-NEXT. Now we are going to review a new type of statement available in most computer languages and in BASIC as well. This new type of statement is a function statement. The major purpose of functions is to provide the programmer with a tool to make shortcuts in calculations that may be used several times. There are two types of functions: those that are built into the software, commonly known as library functions, and those that are defined by the programmer.

LIBRARY FUNCTIONS

BASIC has many library functions available to you, one of which you have already used. Remember the TAB(12) statement you used when working with the PRINT statement? This is actually a function call. Note two important points about functions: they are called by name, and they return a single value. The format of a function is:

SQR(X)

where the function name comes first and the value acted upon (argument) is placed in parentheses. Do not leave a space between the function name and the first parentheses because this will lead to an error message. Functions can be used anywhere in the program and are usually associated with assignment statements. The following function call assigns the variable R the value equal to the square root of N:

```
N=10
R = SQR(N)
PRINT N,R
```

Liberty BASIC has many built-in functions:

ABS( ), ACS( ), ASC( ), ASN( ), ATN( ), CHR$( ), COS( ), DATE$( ), DECHEX$( ), EOF( ), EVAL( ), EVAL$( ), EXP( ), HBMP( ), HEXDEC( ), HWND( ), INP( ), INPUT$( ), INPUTTO$( ), INSTR( ), INT( ), LEFT$( ), LEN( ), LOF( ), LOG( ), LOWER$( ), MAX( ), MIDIPOS( ), MID$( ), MIN( ), MKDIR( ), NOT( ), RIGHT$( ), RMDIR( ), RND( ), SIN( ), SPACE$( ), SQR( ), STR$( ), TAB( ), TAN( ), TIME$( ), TRIM$( ), TXCOUNT( ), UPPER$( ), USING( ), VAL( ), WINSTRING( ), WORD$( )

Use Help to determine their use.

USER-DEFINED FUNCTIONS

Users can define their own functions.

Built-in functions are only one type of function available to the user. BASIC also allows programmers to define their own functions using the define function statement. This statement has the following format:

```
function SomeName(zero or more parameter variable names)
  'code for the function goes in here
  SomeName = returnValueExpression
```

end function

Description:
This statement defines a function. The function can return a string value, or a numeric value. A function that returns a string value must have a name that ends with the "$" character. A function that returns a numeric value must not include a "$" in its name. Zero or more parameters may be passed into the function. A function cannot contain another function definition, nor a subroutine definition. Note that the opening parenthesis is actually part of the function name. Do not include a space between the name and the opening parenthesis, or the code generates an error. Here is a function for calculating basal area for a tree of a given diameter, D:

```
INPUT "Diameter of a tree ";D
PRINT D, BA(D)
FUNCTION BA(D)
  BA = 0.005454 * D^2
END FUNCTION
```

TRIG FUNCTIONS IN DEGREES

Many times we would prefer to have the trigonometric functions return a value in degrees rather than in radians. This can be accomplished through the use of the following statements:

```
INPUT "Enter angle in degrees "; ANGLE1
COSANG = COS(ANGLE1* (4*ATN(1)/180))
SINANG = SIN(ANGLE1* (4*ATN(1)/180))
PRINT COSANG,SINANG
END
```

RANDOM NUMBER GENERATOR

There are several applications in the natural resources field where we are required to generate a list of random numbers. These numbers may be needed for selecting sample locations to visit in the field or for guessing the volume of a tree when using 3P sampling. To do this, we need to use the function RND(X) within a loop. Since the function returns a number between 0 and 1, we will need to multiply the function call by a factor (10 for 1 digit numbers, 100 for 2 digit numbers, etc.) and then assign the outcome to an integer value by rounding to the nearest whole number. The program that follows will produce a list of 25 random integers:

```
FOR x = 1 TO 25
A = RND(1)*1000
PRINT using("###",A)
NEXT x
END
```

REVIEW PROBLEMS

1. What are two major classifications of functions in BASIC?

2. What values are printed in the following statements?

   a. `PRINT INT(44.90)`

   b. `PRINT SQR(SQR(16))`

4. Write a short program to calculate the height of a tree if you are standing 97 feet from the base and the angle from the base of the tree to the top is 46 degrees.

SUBSCRIPTED VARIABLES

Array elements are stored in organized blocks of computer memory.
Dimension statements are declared at the beginning of the program.
One thousand data items can be entered in four lines of code.

We covered simple variable types earlier where you learned the BASIC rules for simple variable names. Now we are ready to deal with another type of variable: the subscripted variable, which is also commonly referred to as an array. The following four variables have very distinct differences, though to the first-time user they seem nearly identical:

```
VOLUME  VOLUME1  VOLUME(1)  VOLUME(Y)
```

The first two are simple variable names, but to the computer they are as different as X and Y. The third variable name is a subscripted variable and represents the first element in the single dimensional array (sometimes called a vector) named VOLUME. The fourth item is a little more complex. It too is a subscripted variable, but the subscript (the value within the parentheses) is also a variable. Some thought on this matter may help you to realize that Y has the potential to represent any of the array elements, depending upon the value it is assigned before the array is accessed. The form of a subscripted variable looks like this:

```
VOLUME (Y)
```

where Y represents the subscript and VOLUME represents the variable name. The variable is verbally referred to as VOLUME sub.

Arrays are unique in that the elements of the array are stored in organized blocks of computer memory. Each element of the array has its own storage location, as would any variable name, but array elements are stored as a group in one section of memory. You may wonder how the computer "knows" how many areas to set aside, because the variable may contain 10 or 110 elements. The answer is that it doesn't; you must declare the size of an array using the dimension statement which has the form

```
DIM VOLUME (100)
DIM HT(100), VOL(100)
```

The DIM statement sets aside the number of storage locations as specified by the integer in parentheses and initializes all the elements of the array to zero. REDIM redimensions an already dimensioned array and clears all elements to zero (or to an empty string in the case of string arrays). The second example indicates how multiple array declarations can be accomplished. In most programming work, the dimension statement is set at a level that is higher than the number of elements that you plan to use in the program, yet not so high as to require a great amount of memory that will never be used. If a subscript is used that is larger than the maximum value declared in the dimension statement, a "Subscript out of range" message appears. Dimension statements, since they reserve memory for program use, are declared at the beginning of the program. It is also possible to let each user define the size of the subscript needed because BASIC will allow the use of a variable in the dimension statement. Review the following lines of code:

```
INPUT "Enter the number of data items"; ITEMS
DIM DBH(ITEMS)
```

In this particular example, the size of the array is a variable entered by the user. In this way, the array is only as large as the application dictates. An array subscript can also be determined by a mathematical operation. Note that the following examples use different types of variable designations as a subscript:

```
VOLUME(Y) VOLUME(Y-1) VOLUME(Y+1)
VOLUME(2*A-C)
```

All the above are legitimate expressions, though in most cases you will find the first three examples most useful. The second and third examples are useful when you wish to refer to the previous or next array element.

ENTERING DATA INTO AN ARRAY

Data is entered into an array using either READ and DATA statements or an INPUT statement. At first it may seem somewhat laborious to use several INPUT statements to enter each array element (e.g. INPUT DBH(l), INPUT DBH(2), etc.) one after another. In fact, if you need 1000 data entries you would have to write 1000 lines of input statements! In actuality, we do not enter data in this way but use a FOR-NEXT loop to enter as many elements as we desire. Review the following statements that would enter 1000 data items in only four lines of code:

```
DIM DBH(1000)
FOR  T=1 TO 1000
  INPUT "Enter data item "; DBH
NEXT  T
```

This is a very powerful utility, so look it over carefully. As you progress through the loop, you enter a single array element equivalent to the index value of the loop (k). Now as *k* increments from 1 to 1000, you will enter DBH(1) through DBH(1000) in numerical order. DATA statements can also be used in this format. Just replace the INPUT statement with the READ statement and add sufficient DATA statements to fill the array.

SUMMING AN ARRAY

One of the nice features of arrays is the ease with which we can manipulate the data they store. The next example shows how to sum the elements of an array. The statement:

```
SUM = SUM + DBH(w)
```

instructs the computer to take the value of DBH(w) and add it to the current value of the variable SUM and take the result and store it in the memory location for the variable SUM. With this statement in a loop, every time you go through the loop the value of SUM increases by the value of DBH(w). One strong suggestion is to initialize all summing

variables to zero at the start of the program. If you do not, you are assuming that the value is zero when you start and you may be unpleasantly surprised!

```
DIM DBH(25)
SUM = 0
FOR w = 1 TO 10
  INPUT "Enter dbh"; DBH(w)
  SUM = SUM + DBH(w)
  NEXT w
FOR w = 1 TO 10
  PRINT DBH(w)
  NEXT w
PRINT:PRINT
PRINT "SUM = ";SUM
AVG=SUM/(w-1)
PRINT "Average = ";AVG
END
```

Two or more statements can share one line number.
PRINT:PRINT results in two blank lines between the last dbh and the SUM value, shows how two or more statements can share one line number with the use of a colon (overuse makes programs difficult to read). Notice that the mean was calculated using the value w-1. Remember our rules for looping? The value of the index value following the loop is always one larger than the ending value for the loop because it increments the index of the loop (in this case w) before it checks to see if it is greater than the ending value!

MAXIMUM AND MINIMUM VALUES

While on the topic of arrays, let us determine the code necessary to select the maximum and minimum value of an array. To accomplish this task, ask the question: How do I do this without a computer? The process may go something like this: I look at the first value and remember it, then scan through the list of values until I find one larger (or smaller, if you are looking for the minimum value). If I do find one larger, I remember this new value and continue scanning until I find another that is larger or I reach the end of the list. The following program includes a subroutine that will identify the largest and smallest value of an array:

```
DIM DBH(10)
FOR y = 1 TO 10
  INPUT "Enter a data item ";DBH(y)
NEXT y
GOSUB [MinMax]
PRINT "Minimum arrary value = ";MIN
PRINT "Maximum array value  = ";MAX
```

```
END
[MinMax]
  MIN = DBH(1)
  MAX = DBH(1)
FOR  b   =   2   TO   10

IF  MIN > DBH(b) THEN MIN=DBH(b)

IF  MAX <   DBH(b) THEN  MAX =DBH(b)
NEXT b
RETURN
```

In the subroutine, the first element of the array is assigned to both MIN and MAX which is analogous to the human example of remembering the first value in a list. In the last FOR-NEXT statement, we start comparing the present MIN and MAX values to the other elements of the array. If we find a new MIN or MAX, we "remember the new value" by reassigning the variables MIN and MAX. Once the FOR-NEXT loop is completed, control transfers to the main program and the minimum and maximum values are printed.

Each element in an array has a special meaning.

MULTIDIMENSIONAL ARRAYS

There are many applications for multidimensional arrays in forestry, but we are going to narrow our view of this topic to a two-dimensional array and the use of a stand table as our example. During our discussion of the two-dimensional array, think of it as a piece of graph paper with the rows and columns labeled and each block in the graph representing an element of the array. To declare the array, we use the same statement but with a slightly different form:

```
DIM  STAND(7, 5)
```

The values in parentheses represent 7 rows and 5 columns, respectively. We know that a stand table is a representation of the number of trees on a tract of timber by diameter class and species, often on a per-acre basis. In our example, we will assign the diameter classes of 20, 22, 24, 26, 28 and 30 to our rows and the columns will represent species (WO, RO, BO, and YP). Notice that even though our columns and rows represent diameter and species, the value actually stored in the array is number of trees. Each element in the array has a special meaning; for example, STAND(1,1) represents 20-inch class white oaks and STAND(6,4) represents 30-inch class yellow-poplars. Our dimension statement defines a 7

by 5 array, but up to this point we have only referenced a 6 (dbh classes) by 4 (number of species) array. Why the discrepancy? The main reason is that most stand tables include a listing of totals, so we have left room for a row of totals by species and a column of totals by dbh class. All we need to do now is find a way to enter our data and compute our information. To enter data (or to print) requires that we use the nested loop feature presented in earlier. The following program requests the number of sample trees for each species/dbh class, computes dbh and species totals, and prints the completed table to the screen. This is a lot of information, so take it slowly!

Data is entered one row at a time.

```
 DIM STAND(7, 5)
 FOR I = 1 TO 6
   FOR J = 1 TO 4
      PRINT "Enter the number of sample trees for species ";J;
      PRINT " and dbh-class ";I;
      INPUT STAND(I,J)
   NEXT J
 NEXT I
'-------------------------------------------------------------
 FOR I = 1 TO 6
   FOR J = 1 TO 4
     STAND(I, 5) = STAND(I,5) + STAND(I, J)
   NEXT J
 NEXT I
'-------------------------------------------------------------
 FOR J = 1 TO 4
   FOR I = 1 TO 6
     STAND(7,J) = STAND(7, J) +STAND(I, J)
   NEXT I
 NEXT J
'-------------------------------------------------------------
-
 FOR K = 1 TO 4
  STAND(7,5) = STAND(7,5) +STAND(7,K)
 NEXT K
'-------------------------------------------------------------
--
PRINT "WO","RO","BO","YP","TOTAL"
 FOR I = 1 TO 7
   FOR J = 1 TO 5
    PRINT STAND(I,J),
  NEXT J
   PRINT
 NEXT I
 END
```

Let's look at the program one section (indicated by dashed lines) at a time. The first section is the input section where we enter our data. Note the way to use PRINT statements with an INPUT. This is especially important if you are creating a menu where the user is to select some option, or, as here, where you are using variables in the PRINT statement other than the one you wish to input. Notice that the semicolon places the question mark on the screen following the last PRINT statement rather than on a new line by itself. This loop will continue to enter all data one row (dbh class) at a time. The next section computes the dbh class totals by summing all the elements in each row and storing them in the total column (column 5) for each row. The next section does practically the same thing, summing the columns rather than the rows (total by species). Think back to our original array of 7 rows and 5 columns. The lower right hand block (STAND(7,5)) represents the total by row or column, or in our application, the total number of trees per acre for the tract (considering all trees and species). We compute this value in the next section by summing the column totals. Now all the computations are complete and we are ready to print the results. The program lists each row of output on a single line (caused by the trailing comma in the fourth line in this section) and then skips a PRINT statement before incrementing the loop integer I. Type the program in your computer and go through it with your own data to see how it works.

REVIEW PROBLEMS

1. What is the purpose of the DIM statement? Where is it found in the program?

2. What are the values for the following subscripted variables after the program runs?

```
DIM Z (10)              Z(1) =
FOR 1% = 1 TO 5         Z(2) =
  Z (1%) = 2*I%"2       Z(3) =
NEXT 1%                 Z(4) =
END                     Z(5) =
```

3. Write a program that will calculate and print the mean, standard deviation, standard error of the mean, and the coefficient of variation as a percent using a vector (one - dimensional array) for any 25 values.

SORTING

Sorting is a process where we arrange elements of an array, or a certain field in a file, in a sequential order. This order may be from smallest to largest (ascending) or from largest to smallest (descending). Sorting can be applied to numeric as well as alphanumeric data. In the field of forestry, there are limited applications where sorting can be applied. Many of the applications are related to accounting type procedures and are more readily solved with some of the more sophisticated database management systems. Liberty BASIC has a command which makes sorting very easy:

SORT arrayName(), start, end, [column]

This command sorts both double and single dimensioned arrays. The start parameter specifies the element with which to begin the sort and the end parameter specifies the element where sorting should stop. Arrays can be sorted in part or in whole, and with double dimensioned arrays, the specific column to sort by can be declared. When this option is used, all the rows are sorted against each other according to the items in the specified column. We will use one simple column of data to illustrate (note: character data, such as names, can be sorted also):

```
dbh(1) = 23
dbh(2) = 10
dbh(3) = 45
SORT dbh(),1,3
for d=1 to 3
  PRINT dbh(d)
next d
```

FILES

Files store data or information that will be needed the next time you run the program.

There are many times in programming where there is a need to store the data or information that you have created for the next time you run the same program. Think of the futility of writing a checkbook program that lists your previous balance, the checks entered, and your final balance, if you had to enter all this information every time you ran the program. A more useful program would store your current balance and check descriptions for use next month when you balance with your bank statement. What use is a computerized Christmas card list if you cannot store your friends names from one

Christmas to the next? We need a way to store the data entered for use when running that particular program again or if accessing the data from a different program. This can be accomplished by designing the program to read and write the data to a file.

A file is made up of individual units called fields and records. Let's use an example to explain this point. A program has been designed that calculates the total volume for a tract of timber. The information regarding each tree (species code, dbh, and number of logs) is stored in a file on a disk (not necessarily the disk on which the program resides). Each of the three variables represents a field, while the collection of these fields for a single tree makes up a record. The data file then contains a list of records, one for each tree.

There are two major types of files in BASIC: sequential files and random access files. We will only consider sequential files here:

The following program will save data to a file.

```
input "Drive and filename, eg. a:\DbhLogs.txt ";FileName$
open FileName$ for output as #4
input "dbh, -9 if done ";dbh
while dbh<>-9
  print #4,dbh;
  print #4,",";              'here we are printing a comma separator
  input "logs            ";logs
  print #4,logs
  print
  input "dbh, -9 if done ";dbh
wend
close #4
end
```

Now, let us make a program to print out the data we just saved to a file.

```
input "Drive and filename, eg. a:\DbhLogs.txt ";FileName$
open FileName$ for input as #4
while eof(#4) = 0       'end of file, eof, is equal to 1 when out of data
   input #4, dbh, logs
wend
close #4
end
```

PROGRAM DESIGN AND DEVELOPMENT

You have now learned many aspects of the BASIC language and should have the capability to begin designing and writing programs for your own applications. This chapter deals with several considerations that are very important to program design. Simply because a program runs does not mean that you have created a good program! This may seem a little hard to swallow, but a poorly designed program can be inefficient, prone to have "bugs", and most importantly, almost impossible to modify or update because the source code is difficult for the author to follow. To this end, this chapter will concentrate on the following topics:

«Program design
• Documentation
• Debugging
• Testing

A program that runs is not necessarily a good program.
Work backwards when designing your program.

PROGRAM DESIGN

Designing a program does not start with entering information through the keyboard. The first step is to decide exactly what you want the program to do. This may seem to be a simple procedure, but if you can list the specific items that you want computed or printed, you have the problem well defined, and that is half the battle. It may be helpful when designing the program to work backwards. Visualize how you would like to see the output and what information should be included on your printout. Once you have an idea of how the final product will look, consider the computations and data that will be required. Look carefully at the data that will be needed by the program to insure against getting halfway through a program only to find that you do not have all the data necessary for your calculations.

Once your ideas regarding program design are clear, it is time to develop a blueprint of how the actual work is to be accomplished. This stage in itself can take longer than writing the actual source code, yet is one of the most important steps. How many major buildings or bridges are built without a blueprint? Sure, a small one-page program may not require such a structured development, just as throwing a log over a small stream for use as a bridge does not require a blueprint. However, the larger the job, the more important that blueprint becomes. The question may then be asked: what does a blueprint for a program look like? Historically, blueprints for computer programs were represented by a series of lines and special symbols which made up what was known as a flowchart. Today, there seems is less support for rigid flowchart development and an increase in the use of less structured approaches, such as the use of pseudocode. Pseudocode represents a step by step English description of each step in a program, possibly incorporating key words that represent programming statements.

The blueprint is one of the most important steps in program design.
while the number of data entries is less
than 100
print each data entry keep a running total compute the average after all data entries are entered

Initially, you may find it difficult to make each line of pseudocode represent only a few statements, but this refinement is a great advantage when writing the program.

. As you develop your plan of attack, keep in mind that your purpose is to provide a blueprint from which the source code can be written and to insure a smooth "top down flow" for your program. If you find yourself drawing many lines and arrows or your pseudocode seems to be going in circles...STOP! Now reconstruct your program so that it progresses, step by step, from the top to the bottom. This does not necessarily mean to eliminate all GOTO [ ] statements, but it does mean to limit their use.

While on the topic of program design, a few additional words regarding modularity are in order. We introduced the idea of modular programming in the chapter on subroutines. Most large programs can be separated into several smaller subroutines and functions. By developing your programs in this way, programs are easier to write and can be completed one section at a time. Modular programs are also easier to modify because one need not rewrite the whole program but only modify a section of a subroutine. This makes your subroutines somewhat portable, since once they are developed, you can include them in any program that needs that type of calculation (assuming that the variable names match). There is no real need to reinvent the wheel every time you write a new program. As an example, once you have written a subroutine to calculate board foot volume for a tree using Doyle, Scribner and International 1/4-inch log rules, you can enter this same subroutine in any program that needs such volume calculations. In addition, if you decide to change your volume equations, you need only change them in the one subroutine to modify the whole program.

The last item we will deal with under the broad category of program design is defensive programming. Defensive programming refers to the protection of your program from improper data entry. This may not seem like the responsibility of the programmer, yet for your program to be useful to you or your company, it is best to avoid rigid data entry requirements. The following items should be reviewed when developing a program blueprint:

1. Validate input data. Make sure at the time of data entry that the data is correct or that it is in a predictable range. If you create a menu from which the user can select item 1, 2 or 3, check the input to see that a 1, 2 or 3 was entered! If not, give an error message and send control to the data entry portion again. There is no sense in continuing with execution if input data are incorrect.

2. Echo input to the screen. After you have requested several items from the user, print them to the screen and ask if they are correct. Using a timber inventory example, think of the time wasted . when a BAF of 2 was mistakenly entered for BAF 20. The program will most likely run, but the information is garbage. A quick screen check could have avoided this problem.

3. Crash softly. If your program receives information (or the lack of it) that is not correct and prevents accurate calculations, print an error message and terminate.

Some aspects of program documentation are designed for the user, not the machine.

DOCUMENTATION

The aspect of program documentation can be divided into external and internal documentation. We will cover each of these separately.

External documentation: This type of documentation is more commonly referred to as a user's guide or manual, often provided in the program. It usually contains a synopsis of the program, any limitations in its use, and a sample program run with an appropriate printout. These may vary in their inherent complexity, but are generally designed to explain how the program operates.

Internal documentation: BASIC allows two forms of remark statements. A remark statement is a nonexecutable statement that allows written explanations to be embedded in the actual program. Anything that follows the word REM or a single quotation mark will be ignored during execution. The question then arises: if it is not going to be used, why include it? It is true that the program will run just as well without it; therefore, the inclusion is for us, not the machine. The major use is to explain the specific purpose of different sections of the program code. Many of you can remember the purpose of the code that you have written today and possibly that which you wrote in the past week, but consider the difficulty of reading a program you wrote a year ago or, even worse, one that someone else wrote! The more explicit your documentation, the easier the program is to modify. Internal documentation takes two BASIC forms; a header block and internal comments.

Remark statements can be included within an executable program line.

DEBUGGING

Debugging is the process by which programmers remove errors from a program. These errors can be syntax errors (improper statements as identified by the interpreter or compiler) or logic errors which are not identified by the system. Logic errors are the more difficult to correct, since you may have gone through three pages of code to print an

incorrect answer. Where is the mistake? ...somewhere in those three pages of code! There are several techniques one can employ when debugging a program. Liberty BASIC has a DEBUG RUN option which helps.

If you have decided to use the modular approach to program design, it is a good idea to tesj each module as it is written rather than waiting until the whole project is completed. This is not to say that if you test each section there will be no bugs in the completed program. You will have bugs, but they should be fewer and easier to locate.

Finally, the oldest and most used technique is to start at the beginning of the program and write down on a separate sheet of paper the exact value for each variable as you "step" through the program one line at a time. This is called tracing the program. You are really mimicking the same procedures as the BASIC interpreter, only manually. This is not a quick technique, but works very well if you are persistent.

One last note regarding debugging: if you have a syntax error and you have reviewed the entire statement and found no errors, check your variable names against the list of reserved words. This is one of the more difficult errors to locate.

For fewer program bugs, test each module as it is written.
Ask others to test your program.

TESTING

Program testing really begins when you start the debugging process. The purpose is to insure that the information provided by the program is correct for the data entered and that the program is free of bugs. The first step in testing a program is to complete a very simple problem by hand and check the values against those computed by your program. If the answers match, GREAT! If not, find out why! The second part of your testing would be to give the program to a co-worker for review. Many times what we see as an obvious prompt sometimes turns out to be ambiguous to others. Sure the program works well for your little test problem, but will it work for others? For most of you, this is where the testing ends (or revisions start); for others, consider sending it to someone practicing in the field for an actual "field production test." Remember, most every large program contains bugs you missed, and the revision process may never end!