CHAPTER 1

BASIC PROGRAMMING RULES

- Introduction
- Screen Display Codes (BASIC Character Set)
- Programming Numbers and Variables
- Expressions and Operators
- Programming Techniques

INTRODUCTION

This chapter talks about how BASIC stores and manipulates data. The topics include:

1) A brief mention of the operating system components and functions as well as the character set used in the Commodore 64.
2) The formation of constants and variables. What types of variables there are. And how constants and variables are stored in memory.
3) The rules for arithmetic calculations, relationship tests, string handling, and logical operations. Also included are the rules for forming expressions, and the data conversions necessary when you're using BASIC with mixed data types.

SCREEN DISPLAY CODES (BASIC CHARACTER SET)

THE OPERATING SYSTEM (OS)

The Operating System is contained in the Read Only Memory (ROM) chips and is a combination of three separate, but interrelated, program modules.

1) The BASIC Interpreter
2) The KERNAL
3) The Screen Editor

1) **The BASIC Interpreter** is responsible for analyzing BASIC statement syntax and for performing the required calculations and/or data manipulation. The BASIC Interpreter has a vocabulary of 65 "keywords" which have special meanings. The upper and lower case alphabet and the digits 0–9 are used to make both keywords and variable names. Certain punctuation characters and special symbols also have meanings for the Interpreter. Table 1-1 lists the special characters and their uses.
2) **The KERNAL** handles most of the interrupt level processing in the system (for details on interrupt level processing, see Chapter 5). The KERNAL also does the actual input and output of data.
3) **The Screen Editor** controls the output to the video screen (television set) and the editing of BASIC program text. In addition, the Screen Editor intercepts keyboard input so that it can decide whether the

Table 1-1. CBM BASIC Character Set

CHARACTER	NAME and DESCRIPTION
	BLANK—separates keywords and variable names
;	SEMI-COLON—used in variable lists to format output
=	EQUAL SIGN—value assignment and relationship testing
+	PLUS SIGN—arithmetic addition or string concatenation (*concatenation*: linking together in a chain)
−	MINUS SIGN—arithmetic subtraction, unary minus (−1)
*	ASTERISK—arithmetic multiplication
/	SLASH—arithmetic division
↑	UP ARROW—arithmetic exponentiation
(	LEFT PARENTHESIS—expression evaluation and functions
)	RIGHT PARENTHESIS—expression evaluation and functions
%	PERCENT—declares variable name as an integer
#	NUMBER—comes before logical file number in input/output statements
$	DOLLAR SIGN—declares variable name as a string
,	COMMA—used in variable lists to format output; also separates command parameters
.	PERIOD—decimal point in floating point constants
"	QUOTATION MARK—encloses string constants
:	COLON—separates multiple BASIC statements in a line
?	QUESTION MARK—abbreviation for the keyword PRINT
<	LESS THAN—used in relationship tests
>	GREATER THAN—used in relationship tests
π	PI—the numeric constant 3.141592654

characters put in should be acted upon immediately, or passed on to the BASIC Interpreter.

The Operating System gives you two modes of BASIC operation:

1) DIRECT Mode
2) PROGRAM Mode

1) When you're using the DIRECT mode, BASIC statements don't have line numbers in front of the statement. They are executed whenever the RETURN key is pressed.
2) The PROGRAM mode is the one you use for running programs.

When using the PROGRAM mode, all of your BASIC statements must have line numbers in front of them. You can have more than one BASIC statement in a line of your program, but the number of statements is limited by the fact that you can only put 80 characters on a logical screen line. This means that if you are going to go over the 80 character limit you have to put the entire BASIC statement that doesn't fit on a new line with a new line number.

NOTE: Always type NEW and hit RETURN before starting a new program.

The Commodore 64 has two complete character sets that you can use either from the keyboard or in your programs.

In SET 1, the upper case alphabet and the numbers 0–9 are available without pressing the SHIFT key. If you hold down the SHIFT key while typing, the graphics characters on the RIGHT side of the front of the keys are used. If you hold down the C= key while typing, the graphics characters on the LEFT side of the front of the key are used. Holding down the SHIFT key while typing any character that doesn't have graphic symbols on the front of the key gives you the symbol on the top most part of the key.

In SET 2, the lower case alphabet and the numbers 0–9 are available without pressing the SHIFT key. The upper case alphabet is available when you hold down the SHIFT key while typing. Again, the graphic symbols on the LEFT side of the front of the keys are displayed by pressing the C= key, while the symbols on the top most part of any key without graphics characters are selected when you hold down the SHIFT key while typing.

To switch from one character set to the other press the C= and the SHIFT keys together.

PROGRAMMING NUMBERS AND VARIABLES

INTEGER, FLOATING-POINT AND STRING CONSTANTS

Constants are the data values that you put in your BASIC statements. BASIC uses these values to represent data during statement execution. CBM BASIC can recognize and manipulate three types of constants:

1) INTEGER NUMBERS
2) FLOATING-POINT NUMBERS
3) STRINGS

Integer constants are whole numbers (numbers without decimal points). Integer constants must be between −32768 and +32767. *Integer constants do not have decimal points or commas between digits.* If the plus (+) sign is left out, the constant is assumed to be a positive number. Zeros coming before a constant are ignored and shouldn't be used since they waste memory and slow down your program. However, they won't cause an error. Integers are stored in memory as two-byte binary numbers. Some examples of integer constants are:

```
   −12
  8765
−32768
   +44
     0
−32767
```

NOTE: Do NOT put commas inside any number. For example, always type 32,000 as 32000. If you put a comma in the middle of a number you will get the BASIC error message **?SYNTAX ERROR.**

Floating-point constants are positive or negative numbers and can contain fractions. Fractional parts of a number may be shown using a decimal point. Once again remember that commas are NOT used between numbers. If the plus sign (+) is left off the front of a number, the Commodore 64 assumes that the number is positive. If you leave off the decimal point the computer will assume that it follows the last digit of the number. And as with integers, zeros that come before a constant are ignored. Floating-point constants can be used in two ways:

1) SIMPLE NUMBER
2) SCIENTIFIC NOTATION

Floating-point constants will show you up to nine digits on your screen. These digits can represent values between −999999999. and +999999999. If you enter more than nine digits the number will be rounded based on the tenth digit. If the tenth digit is greater than or equal to 5 the number will be rounded upward. Less than 5 the number will be rounded downward. This could be important to the final totals of some numbers you may want to work with.

Floating-point numbers are stored (using five bytes of memory) and are manipulated in calculations with ten places of accuracy. However, the numbers are rounded to nine digits when results are printed. Some examples of simple floating-point numbers are:

```
  1.23
   .998877
 +3.1459
   .7777777
-333.
   .01
```

Numbers smaller than .01 or larger than 999999999. will be printed in *scientific notation*. In scientific notation a floating-point constant is made up of three parts:

1) THE MANTISSA
2) THE LETTER E
3) THE EXPONENT

The *mantissa* is a simple floating-point number. The letter *E* is used to tell you that you're seeing the number in exponential form. In other words *E* represents *10 (eg., 3E3=3*10↑3=3000). And the *exponent* is what multiplication power of 10 the number is raised to.

Both the mantissa and the exponent are signed (+ or −) numbers. The exponent's range is from −39 to +38 and it indicates the number of places that the actual decimal point in the mantissa would be moved to the left (−) or right (+) if the value of the constant were represented as a simple number.

There is a limit to the size of floating-point numbers that BASIC can handle, even in scientific notation: the largest number is +1.70141183E+38 and calculations which would result in a larger number will display the BASIC error message **?OVERFLOW ERROR**. The smallest floating-point number is +2.93873588E−39 and calculations which result in a smaller value give you zero as an answer and NO error message. Some examples of floating-point numbers in scientific notation (and their decimal values) are:

```
235.988E−3      (.235988)
2359E6          (2359000000.)
−7.09E−12       (−.00000000000709)
−3.14159E+5     (−314159.)
```

String constants are groups of alphanumeric information like letters, numbers and symbols. When you enter a string from the keyboard, it can have any length up to the space available in an 80-character line (that is, any character spaces NOT taken up by the line number and other required parts of the statement).

A string constant can contain blanks, letters, numbers, punctuation and color or cursor control characters in any combination. You can even put commas between numbers. *The only character which cannot be included in a string is the double quote mark (").* This is because the double quote mark is used to define the beginning and end of the string. A string can also have a null value—which means that it can contain no character data. You can leave the ending quote mark off of a string if it's the last item on a line or if it's followed by a colon (:). Some examples of string constants are:

```
""          ( a null string)
"HELLO"
"$25,000.00"
"NUMBER OF EMPLOYEES"
```

NOTE: Use CHR$(34) to include quotes (") in strings.

INTEGER, FLOATING-POINT AND STRING VARIABLES

Variables are names that represent data values used in your BASIC statements. The value represented by a variable can be assigned by setting it equal to a constant, or it can be the result of calculations in the program. Variable data, like constants, can be integers, floating-point numbers, or strings. If you refer to a variable name in a program before a value has been assigned, the BASIC Interpreter will automatically create the variable with a value of zero if it's an integer or floating-point number. Or it will create a variable with a null value if you're using strings.

Variable names can be any length but only the first two characters are considered significant in CBM BASIC. This means that all names used for variables must NOT have the same first two characters. *Variable names may NOT be the same as BASIC keywords and they may NOT contain keywords in the middle of variable names.* Keywords include all BASIC commands, statements, function names and logical operator names. If you accidentally use a keyword in the middle of a variable name, the BASIC error message **?SYNTAX ERROR** will show up on your screen.

The characters used to form variable names are the alphabet and the numbers 0–9. The first character of the name must be a letter. Data type declaration characters (%) and ($) can be used as the last character of the name. The percent sign (%) declares the variable to be an integer and the dollar sign ($) declares a string variable. If no type declaration character is used the Interpreter will assume that the variable is a floating-point. Some examples of variable names, value assignments and data types are:

```
A$="GROSS SALES"    (string variable)
MTH$="JAN"+A$       (string variable)
K%=5                (integer variable)
CNT%=CNT%+1         (integer variable)
FP=12.5             (floating-point variable)
SUM=FP*CNT%         (floating-point variable)
```

INTEGER, FLOATING-POINT AND STRING ARRAYS

An *array* is a table (or list) of associated data items referred to by a single variable name. In other words, an array is a sequence of related variables. A table of numbers can be seen as an array, for example. The individual numbers within the table become "elements" of the array.

Arrays are a useful shorthand way of describing a large number of related variables. Take a table of numbers for instance. Let's say that the table has 10 rows of numbers with 20 numbers in each row. That makes a total of 200 numbers in the table. Without a single array name to call on you would have to assign a unique name to each value in the table. But because you can use arrays you only need one name for the array and all the elements in the array are identified by their individual locations within the array.

Array names can be integers, floating-points or string data types and all elements in the array have the same data type as the array name. Arrays can have a single dimension (as in a simple list) or they can have multiple dimensions (imagine a grid marked in rows and columns or a Rubik's Cube®). Each element of an array is uniquely identified and referred to by a subscript (or index variable) following the array name, enclosed within parentheses ( ).

The maximum number of dimensions an array can have in theory is 255 and the number of elements in each dimension is limited to 32767. But for practical purposes array sizes are limited by the memory space available to hold their data and/or the 80 character logical screen line. If an array has only one dimension and its subscript value will never exceed 10 (11 items: 0 thru 10) then the array will be created by the Interpreter and filled with zeros (or nulls if string type) the first time any element of the array is referred to, otherwise the BASIC DIM statement must be used to define the shape and size of the array. The amount of memory required to store an array can be determined as follows:

```
         5 bytes for the array name
       + 2 bytes for each dimension of the array
       + 2 bytes per element for integers
    OR + 5 bytes per element for floating-point
    OR + 3 bytes per element for strings
   AND + 1 byte per character in each string element
```

Subscripts can be integer constants, variables, or an arithmetic expression which gives an integer result. Separate subscripts, with commas between them, are required for each dimension of an array. Subscripts can have values from zero up to the number of elements in the respective dimensions of the array. Values outside that range will cause the BASIC error message **?BAD SUBSCRIPT.** Some examples of array names, value assignments and data types are:

A$(0)="GROSS SALES"	(string array)
MTH$(K%)="JAN"	(string array)
G2%(X)=5	(integer array)
CNT%(G2%(X))=CNT%(1)−2	(integer array)
FP(12*K%)=24.8	(floating-point array)
SUM(CNT%(1))=FP↑K%	(floating-point array)

A(5)=0	(sets the 5th element in the 1 dimensional array called "A" equal to 0)
B(5,6)=0	(sets the element in row position 5 and column position 6 in the 2 dimensional array called "B" equal to 0)
C(1,2,3)=0	(sets the element in row position 1, column position 2, and depth position 3 in the 3 dimensional array called "C" equal to 0)

EXPRESSIONS AND OPERATORS

Expressions are formed using constants, variables and/or arrays. An expression can be a single constant, simple variable, or an array variable of any type. It can also be a combination of constants and variables with arithmetic, relational or logical operators designed to produce a single value. How operators work is explained below. Expressions can be separated into two classes:

1) ARITHMETIC
2) STRING

Expressions are normally thought of as having two or more data items called *operands*. Each operand is separated by a single *operator* to produce the desired result. This is usually done by assigning the value of the expression to a variable name. All of the examples of constants and variables that you've seen so far, were also examples of expressions.

An operator is a special symbol the BASIC Interpreter in your Commodore 64 recognizes as representing an operation to be performed on the variables or constant data. One or more operators, combined with one or more variables and/or constants form an expression. Arithmetic, relational and logical operators are recognized by Commodore 64 BASIC.

ARITHMETIC EXPRESSIONS

Arithmetic expressions, when solved, will give an integer or floating-point value. The arithmetic operators (+, −, *, /, ↑) are used to perform addition, subtraction, multiplication, division and exponentiation operations respectively.

ARITHMETIC OPERATIONS

An arithmetic operator defines an arithmetic operation which is performed on the two operands on either side of the operator. Arithmetic operations are performed using floating-point numbers. Integers are converted to floating-point numbers before an arithmetic operation is performed. The result is converted back to an integer if it is assigned to an integer variable name.

ADDITION (+): The plus sign (+) specifies that the operand on the right is added to the operand on the left.

EXAMPLES:

```
2+2
A+B+C
X%+1
BR+10E 2
```

SUBTRACTION (−): The minus sign (−) specifies that the operand on the right is subtracted from the operand on the left.

EXAMPLES:

```
4−1
100−64
A−B
55−142
```

The minus can also be used as a unary minus. That means that it is the minus sign in front of a negative number. This is equal to subtracting the number from zero (0).

EXAMPLES:

```
−5
−9E4
−B
4− (−2) same as 4+2
```

MULTIPLICATION (*): An asterisk (*) specifies that the operand on the left is multiplied by the operand on the right.

EXAMPLES:

```
100*2
50*0
A*X1
R%*14
```

DIVISION (/): The slash (/) specifies that the operand on the left is divided by the operand on the right.

EXAMPLES:

```
10/2
6400/4
A/B
4E2/XR
```

EXPONENTIATION (↑): The up arrow (↑) specifies that the operand on the left is raised to the power specified by the operand on the right (the exponent). If the operand on the right is a 2, the number on the left is squared; if the exponent is a 3, the number on the left is cubed, etc. The exponent can be any number so long as the result of the operation gives a valid floating-point number.

EXAMPLES:

2↑2	Equivalent to: 2*2
3↑3	Equivalent to: 3*3*3
4↑4	Equivalent to: 4*4*4*4
AB↑CD	
3↑−2	Equivalent to: ⅓*⅓

RELATIONAL OPERATORS

The relational operators (<, =, >, <=, >=, <>) are primarily used to compare the values of two operands, but they also produce an arithmetic result. The relational operators and the logical operators (AND, OR, and NOT), when used in comparisons, actually produce an arithmetic true/false evaluation of an expression. If the relationship stated in the expression is true the result is assigned an integer value of −1 and if it's false a value of 0 is assigned. These are the relational operators:

<	LESS THAN
=	EQUAL TO
>	GREATER THAN
<=	LESS THAN OR EQUAL TO
>=	GREATER THAN OR EQUAL TO
<>	NOT EQUAL TO

EXAMPLES:

1=5−4	result true (−1)
14>66	result false (0)
15>=15	result true (−1)

Relational operators can be used to compare strings. For comparison purposes, the letters of the alphabet have the order A<B<C<D, etc. Strings are compared by evaluating the relationship between corresponding characters from left to right (see String Operations).

EXAMPLES:

```
"A" < "B"      result true (−1)
"X" = "YY"     result false (0)
BB$ <> CC$
```

Numeric data items can only be compared (or assigned) to other numeric items. The same is true when comparing strings, otherwise the BASIC error message **?TYPE MISMATCH** will occur. Numeric operands are compared by first converting the values of either or both operands from integer to floating-point form, as necessary. Then the relationship of the floating-point values is evaluated to give a true/false result.

At the end of all comparisons, you get an integer no matter what data type the operand is (even if both are strings). Because of this, a comparison of two operands can be used as an operand in performing calculations. The result will be −1 or 0 and can be used as anything but a divisor, since division by zero is illegal.

LOGICAL OPERATORS

The logical operators (AND, OR, NOT) can be used to modify the meanings of the relational operators or to produce an arithmetic result. Logical operators can produce results other than −1 and 0, though any nonzero result is considered true when testing for a true/false condition.

The logical operators (sometimes called *Boolean* operators) can also be used to perform logic operations on individual binary digits (bits) in two operands. But when you're using the NOT operator, the operation is performed only on the single operand to the right. The operands must be in the integer range of values (−32768 to +32767) (floating-point numbers are converted to integers) and logical operations give an integer result.

Logical operations are performed bit-by-corresponding-bit on the two operands. The logical AND produces a bit result of 1 only if both operand bits are 1. The logical OR produces a bit result of 1 if either operand bit is 1. The logical NOT is the opposite value of each bit as a single operand. In other words, it's really saying, "If it's NOT 1 then it is 0. If it's NOT 0 then it is 1."

The exclusive OR (XOR) doesn't have a logical operator but it is performed as part of the WAIT statement. Exclusive OR means that if the bits of two operands are equal then the result is 0 otherwise the result is 1.

Logical operations are defined by groups of statements which, taken together, constitute a Boolean "truth table" as shown in Table 1-2.

Table 1-2. Boolean Truth Table

The AND operation results in a 1 only if both bits are 1:

```
1 AND 1 = 1
0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0
```

The OR operation results in a 1 if either bit is 1:

```
1 OR 1 = 1
0 OR 1 = 1
1 OR 0 = 1
0 OR 0 = 0
```

The NOT operation logically complements each bit:

```
NOT 1 = 0
NOT 0 = 1
```

The exclusive OR (XOR) is part of the WAIT statement:

```
1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 1 = 1
0 XOR 0 = 0
```

The logical operators AND, OR and NOT specify a Boolean arithmetic operation to be performed on the two operand expressions on either side of the operator. In the case of NOT, ONLY the operand on the RIGHT is considered. Logical operations (or Boolean arithmetic) aren't performed until all arithmetic and relational operations in an expression have been completed.

EXAMPLES:

IF A=100 AND B=100 THEN 10	(if both A and B have a value of 100 then the result is true)
A=96 AND 32: PRINT A	(A – 32)

IF A=100 OR B=100 THEN 20	(if A or B is 100 then the result is true)
A=64 OR 32: PRINT A	(A = 96)
IF NOT X<Y THEN 30	(if X>=Y the result is true)
X= NOT 96	(result is −97 (two's complement))

HIERARCHY OF OPERATIONS

All expressions perform the different types of operations according to a fixed hierarchy. In other words, certain operations are performed before other operations. The normal order of operations can be modified by enclosing two or more operands within parentheses ( ), creating a "subexpression." The parts of an expression enclosed in parentheses will be reduced to a single value before working on parts outside the parentheses.

When you use parentheses in expressions, they must be paired so that you always have an equal number of left and right parentheses. Otherwise, the BASIC error message **?SYNTAX ERROR** will appear.

Expressions which have operands inside parentheses may themselves be enclosed in parentheses, forming complex expressions of multiple levels. This is called *nesting*. Parentheses can be nested in expressions to a maximum depth of ten levels—ten matching sets of parentheses. The inner-most expression has its operations performed first. Some examples of expressions are:

```
A+B
C↑(D+E)/2
((X−C↑(D+E)/2)*10)+1
GG$>HH$
JJ$+"MORE"
K%=1 AND M<>X
K%=2 OR (A=B AND M<X)
NOT (D=E)
```

The BASIC Interpreter will normally perform operations on expressions by performing arithmetic operations first, then relational operations, and logical operations last. Both arithmetic and logical operators have an order of precedence (or hierarchy of operations) within themselves. On the other hand, relational operators do not have an order of precedence and will be performed as the expression is evaluated from left to right.

If all remaining operators in an expression have the same level of precedence then operations happen from left to right. When performing operations on expressions within parentheses, the normal order of precedence is maintained. The hierarchy of arithmetic and logical operations is shown in Table 1-3 from first to last in order of precedence.

Table 1-3. Hierarchy of Operations Performed on Expressions

OPERATOR	DESCRIPTION	EXAMPLE
↑	Exponentiation	BASE ↑ EXP
−	Negation (Unary Minus)	−A
* /	Multiplication Division	AB * CD EF / GH
+ −	Addition Subtraction	CNT + 2 JK − PQ
> = <	Relational Operations	A <= B
NOT	Logical NOT (Integer Two's Complement)	NOT K%
AND	Logical AND	JK AND 128
OR	Logical OR	PQ OR 15

STRING OPERATIONS

Strings are compared using the same relational operators (=, <>, <=, >=, <, >) that are used for comparing numbers. String comparisons are made by taking one character at a time (left-to-right) from each string and evaluating each character code position from the PET/CBM character set. If the character codes are the same, the characters are equal. If the character codes differ, the character with the lower code number is lower in the character set. The comparison stops when the end of either string is reached. All other things being equal, the shorter string is considered less than the longer string. *Leading or trailing blanks ARE significant.*

Regardless of the data types, at the end of all comparisons you get an integer result. This is true even if both operands are strings. Because of this a comparison of two string operands can be used as an operand in performing calculations. The result will be −1 or 0 (true or false) and can be used as anything but a divisor since division by zero is illegal.

STRING EXPRESSIONS

Expressions are treated as if an implied "<>0" follows them. This means that if an expression is true then the next BASIC statements on the same program line are executed. If the expression is false the rest of the line is ignored and the next line in the program is executed.

Just as with numbers, you can also perform operations on string variables. The only string arithmetic operator recognized by CBM BASIC is the plus sign (+) which is used to perform concatenation of strings. When strings are concatenated, the string on the right of the plus sign is appended to the string on the left, forming a third string as a result. The result can be printed immediately, used in a comparison, or assigned to a variable name. If a string data item is compared with (or set equal to) a numeric item, or vice-versa, the BASIC error message **?TYPE MISMATCH** will occur. Some examples of string expressions and concatenation are:

```
10 A$="FILE" : B$="NAME"
20 NAM$ = A$ + B$           (gives the string: FILENAME)
30 RES$ = "NEW " + A$ + B$ (gives the string: NEW FILENAME)
```

Note space here.

PROGRAMMING TECHNIQUES

DATA CONVERSIONS

When necessary, the CBM BASIC Interpreter will convert a numeric data item from an integer to floating-point, or vice-versa, according to the following rules:

- All arithmetic and relational operations are performed in floating-point format. Integers are converted to floating-point form for evaluation of the expression, and the result is converted back to integer. Logical operations convert their operands to integers and return an integer result.
- If a numeric variable name of one type is set equal to a numeric data item of a different type, the number will be converted and stored as the data type declared in the variable name.
- When a floating-point value is converted to an integer, the fractional portion is truncated (eliminated) and the integer result is less than or equal to the floating-point value. If the result is outside the range of +32767 thru −32768, the BASIC error message **?ILLEGAL QUANTITY** will occur.

USING THE INPUT STATEMENT

Now that you know what variables are, let's take that information and put it together with the INPUT statement for some practical programming applications.

In our first example, you can think of a variable as a "storage compartment" where the Commodore 64 stores the user's response to your prompt question. To write a program which asks the user to type in a name, you might assign the variable N$ to the name typed in. Now every time you PRINT N$ in your program, the Commodore 64 will automatically PRINT the name that the user typed in.

Type the word NEW on your Commodore 64. Hit the RETURN key, and try this example:

```
10 PRINT "YOUR NAME":INPUT N$
20 PRINT "HELLO," N$
```

In this example you used N to remind yourself that this variable stands for "NAME." The dollar sign ($) is used to tell the computer that you're using a string variable. It is important to differentiate between the two types of variables:

1) NUMERIC
2) STRING

You probably remember from the earlier sections that numeric variables are used to store number values such as 1, 100, 4000, etc. A numeric variable can be a single letter (A), any two letters (AB), a letter and a number (A1), or two letters and a number (AB1). You can save memory space by using shorter variables. Another helpful hint is to use letters and numbers for different categories in the same program (A1, A2, A3). Also, if you want whole numbers for an answer instead of numbers with decimal points, all you have to do is put a percent sign (%) at the end of your variable name (AB%, A1%, etc.)

Now let's look at a few examples that use different types of variables and expressions with the INPUT statement.

```
10 PRINT "ENTER A NUMBER":INPUT A
20 PRINT A

10 PRINT "ENTER A WORD":INPUT A$
20 PRINT A$

10 PRINT "ENTER A NUMBER":INPUT A
20 PRINT A "TIMES 5 EQUALS" A*5
```

NOTE: Example 3 shows that MESSAGES or PROMPTS are inside the quotation marks (" ") while the variables are outside. Notice, too, that in line 20 the variable A was printed first, then the message "TIMES 5 EQUALS", and then the calculation, multiply variable A by 5 (A*5).

Calculations are important in most programs. You have a choice of using "actual numbers" or variables when doing calculations, but if you're working with numbers supplied by a user you must use numeric variables. Begin by asking the user to type in two numbers like this:

```
10 PRINT "TYPE 2 NUMBERS":INPUT A:INPUT B
```

INCOME/EXPENSE BUDGET EXAMPLE

```
5 PRINT "♥"— SHIFT CLR/HOME
10 PRINT"MONTHLY INCOME":INPUT IN
20 PRINT
30 PRINT"EXPENSE CATEGORY 1":INPUT E1$
40 PRINT"EXPENSE AMOUNT":INPUT E1
50 PRINT
60 PRINT"EXPENSE CATEGORY 2":INPUT E2$
70 PRINT"EXPENSE AMOUNT":INPUT E2
80 PRINT
90 PRINT"EXPENSE CATEGORY 3":INPUT E3$
100 PRINT"EXPENSE AMOUNT":INPUT E3
110 PRINT "♥"— SHIFT CLR/HOME
120 E=E1+E2+E3
130 EP=E/IN
140 PRINT"MONTHLY INCOME: $"IN
150 PRINT"TOTAL EXPENSES: $"E
160 PRINT"BALANCE EQUALS: $"IN-E
170 PRINT
180 PRINT E1$"="(E1/E)*100"% OF TOTAL EXPENSES"
190 PRINT E2$"="(E2/E)*100"% OF TOTAL EXPENSES"
200 PRINTE3$"="(E3/E)*100"% OF TOTAL EXPENSES"
210 PRINT
220 PRINT"YOUR EXPENSES="EP*100"% OF YOUR TOTAL
INCOME"
230 FOR X=1TO5000:NEXT:PRINT
240 PRINT"REPEAT? (Y OR N)":INPUT Y$:IF Y$="Y"THEN5
250 PRINT "♥":END
           SHIFT CLR/HOME
```

NOTE: IN can NOT = 0, and E1, E2, E3 can NOT all be 0 at the same time.

LINE-BY-LINE EXPLANATION OF INCOME/EXPENSE BUDGET EXAMPLE

Line(s)	Description
5	Clears the screen.
10	PRINT/INPUT statement.
20	Inserts blank line.
30	Expense Category 1 = E1$.
40	Expense Amount = E1.
50	Inserts blank line.
60	Expense Category 2 = E2$.
70	Expense Amount 2 = E2.
80	Inserts blank line.
90	Expense Category 3 = E3$.
100	Expense Amount 3 = E3.
110	Clears the screen.
120	Add Expense Amounts = E.
130	Calculate Expense/Income%.
140	Display Income.
150	Display Total Expenses.
160	Display Income – Expenses.
170	Inserts blank line.
180–200	Lines 180–200 calculate % each expense amount is of total expenses.
210	Inserts blank line.
220	Display E/I %.
230	Time delay loop.

Now multiply those two numbers together to create a new variable C as shown in line 20 below:

```
20 C=A*B
```

To PRINT the result as a message type

```
30 PRINT A "TIMES" B "EQUALS" C
```

Enter these 3 lines and RUN the program. Notice that the messages are inside the quotes while the variables are not.

Now let's say that you wanted a dollar sign ($) in front of the number represented by variable C. The $ must be PRINTed inside quotes and in front of variable C. To add the $ to your program hit the RUN/STOP and RESTORE keys. Now type in line 40 as follows:

```
40 PRINT "$" C
```

Now hit RETURN, type RUN and hit RETURN again.

The dollar sign goes in quotes because the variable C only represents a number and can't contain a $. If the number represented by C was 100 then the Commodore 64 screen would display $ 100. But, if you tried to PRINT $C without using the quotes, you would get a **?SYNTAX ERROR** message.

One last tip about $$$: You can create a variable that represents a dollar sign which you can then substitute for the $ when you want to use it with numeric variables. For example:

```
10 Z$="$"
```

Now whenever you need a dollar sign you can use the string variable Z$. Try this:

```
10 Z$="$":INPUT A
20 PRINT Z$A
```

Line 10 defines the $ as a string variable called Z$, and then INPUTs a number called A. Line 20 PRINTs Z$ ($) next to A (number).

You'll probably find that it's easier to assign certain characters, like dollar signs, to a string variable than to type "$" every time you want to calculate dollars or other items which require " " like %.

USING THE GET STATEMENT

Most simple programs use the INPUT statement to get data from the person operating the computer. When you're dealing with more complex needs, like protection from typing errors, the GET statement gives you more flexibility and your program more "intelligence." This section shows you how to use the GET statement to add some special screen editing features to your programs.

The Commodore 64 has a keyboard buffer that holds up to 10 characters. This means that if the computer is busy doing some operation and it's not reading the keyboard, you can still type in up to 10 characters, which will be used as soon as the Commodore 64 finishes what it was doing. To demonstrate this, type in this program on your Commodore 64:

```
NEW
10 TI$="000000"
20 IF TI$ < "000015" THEN 20
```

Now type RUN, hit RETURN and while the program is RUNning type in the word HELLO.

Notice that nothing happened for about 15 seconds when the program started. Only then did the message HELLO appear on the screen.

Imagine standing in line for a movie. The first person in the line is the first to get a ticket and leave the line. The last person in line is last for a ticket. The GET statement acts like a ticket taker. First it looks to see if there are any characters "in line." In other words have any keys been typed. If the answer is yes then that character gets placed in the appropriate variable. If no key was pressed then an empty value is assigned to a variable.

At this point it's important to note that if you try to put more than 10 characters into the buffer at one time, all those over the 10th character will be lost.

Since the GET statement will keep going even when no character is typed, it is often necessary to put the GET statement into a loop so that it will have to wait until someone hits a key or until a character is received through your program.

Below is the recommended form for the GET statement. Type NEW to erase your previous program.

```
10 GET A$ : IF A$ = "" THEN 10
```

Notice that there is NO SPACE between the quote marks ("") on this line. This indicates an empty value and sends the program back to the GET statement in a continuous loop until someone hits a key on the computer. Once a key is hit the program will continue with the line following line 10. Add this line to your program:

```
100 PRINT A$;: GOTO 10
```

Now RUN the program. Notice that no cursor ■ appears on the screen, but any character you type will be printed in the screen. This 2-line program can be turned into part of a screen editor program as shown below.

There are many things you can do with a screen editor. You can have a flashing cursor. You can keep certain keys like CLR/HOME from accidentally erasing the whole screen. You might even want to be able to use your function keys to represent whole words or phrases. And speaking of function keys, the following program lines give each function key a special purpose. Remember this is only the beginning of a program that you can customize for your needs.

```
20 IF A$ = CHR$(133) THEN POKE 53280,8:GOTO 10
30 IF A$ = CHR$(134) THEN POKE 53281,4:GOTO 10
40 IF A$ = CHR$(135) THEN A$="DEAR SIR:"+CHR$(13)
50 IF A$ = CHR$(136) THEN A$="SINCERELY,"+CHR$(13)
```

The CHR$ numbers in parentheses come from the CHR$ code chart in Appendix C. The chart lists a different number for each character. The four function keys are set up to perform the tasks represented by the instructions that follow the word THEN in each line. By changing the CHR$ number inside each set of parentheses you can designate different keys. Different instructions would be performed if you changed the information after the THEN statement.

HOW TO CRUNCH BASIC PROGRAMS

You can pack more instructions—and power—into your BASIC programs by making each program as short as possible. This process of shortening programs is called "crunching."

Crunching programs lets you squeeze the maximum possible number of instructions into your program. It also helps you reduce the size of programs which might not otherwise run in a given size; and if you're writing a program which requires the input of data such as inventory items, numbers or text, a short program will leave more memory space free to hold data.

ABBREVIATING KEYWORDS

A list of keyword abbreviations is given in Appendix A. This is helpful when you program because you can actually crowd more information on each line using abbreviations. The most frequently used abbreviation is the question mark (?) which is the BASIC abbreviation for the PRINT command. However, if you LIST a program that has abbreviations, the Commodore 64 will automatically print out the listing with the full-length keywords. If any program line exceeds 80 characters (2 lines on the screen) with the keywords unabbreviated, and you want to change it, you will have to re-enter that line with the abbreviations before saving the program. SAVEing a program incorporates the keywords without inflating any lines because BASIC keywords are tokenized by the Commodore 64. Usually, abbreviations are added after a program is written and it isn't going to be LISTed any more before SAVEing.

SHORTENING PROGRAM LINE NUMBERS

Most programmers start their programs at line 100 and number each line at intervals of 10 (i.e., 100, 110, 120). This allows extra lines of instruction to be added (111, 112, etc.) as the program is developed. One means of crunching the program after it is completed is to *change the line numbers to the lowest numbers possible* (i.e., 1, 2, 3) because longer line numbers take more memory than shorter numbers when referenced by GOTO and GOSUB statements. For instance, the number 100 uses 3 bytes of memory (one for each number) while the number 1 uses only 1 byte.

PUTTING MULTIPLE INSTRUCTIONS ON EACH LINE

You can put more than one instruction on each numbered line in your program by separating them by a colon. The only limitation is that all the instructions on each line, including colons, should not exceed the standard 80-character line length. Here is an example of two programs, before and after crunching:

BEFORE CRUNCHING:

```
10 PRINT "HELLO. . .";
20 FOR T=1 TO 500:NEXT
30 PRINT "HELLO, AGAIN . . ."
40 GOTO 10
```

AFTER CRUNCHING:

```
10 PRINT "HELLO . . .";:FORT=1TO
   500:NEXT:PRINT"HELLO,
   AGAIN . . .":GOTO10
```

REMOVING REM STATEMENTS

REM statements are helpful in reminding yourself—or showing other programmers—what a particular section of a program is doing. However, when the program is completed and ready to use, you probably won't need those REM statements anymore and you can save quite a bit of space by removing the REM statements. If you plan to revise or study the program structure in the future, it's a good idea to keep a copy on file with the REM statements intact.

USING VARIABLES

If a number, word or sentence is used repeatedly in your program it's usually best to define those long words or numbers with a one or two letter variable. Numbers can be defined as single letters. Words and sentences can be defined as string variables using a letter and dollar sign. Here's one example:

BEFORE CRUNCHING:

```
10 POKE 54296,15
20 POKE 54276,33
30 POKE 54273,10
40 POKE 54273,40
50 POKE 54273,70
60 POKE 54296,0
```

AFTER CRUNCHING:

```
10 V=54296:F=54273
20 POKEV,15:POKE54276,33
30 POKEF,10:POKEF,40:POKEF,70
40 POKEV,0
```

USING READ AND DATA STATEMENTS

Large amounts of data can be typed in as one piece of data at a time, over and over again . . . or you can print the instructional part of the program ONCE and print all the data to be handled in a long running list called the DATA statement. This is especially good for crowding large lists of numbers into a program.

USING ARRAYS AND MATRICES

Arrays and matrices are similar to DATA statements in that long amounts of data can be handled as a list, with the data handling portion of the program drawing from that list, in sequence. Arrays differ in that the list can be multi-dimensional

ELIMINATING SPACES

One of the easiest ways to reduce the size of your program is to eliminate all the spaces. Although we often include spaces in sample programs to provide clarity, you actually don't need any spaces in your program and will save space if you eliminate them.

USING GOSUB ROUTINES

If you use a particular line or instruction over and over, it might be wise to GOSUB to the line from several places in your program, rather than write the whole line or instruction every time you use it.

USING TAB AND SPC

Instead of PRINTing several cursor commands to position a character on the screen, it is often more economical to use the TAB and SPC instructions to position words or characters on the screen.