BASIC PROGRAMMING

With Applications In Business

Bryce A. Martens
Ronald J. Tortorelli

BASIC PROGRAMMING

With Applications in Business

Bryce A. Martens **Ronald J. Tortorelli**

San Francisco State University

Kendall/Hunt
Publishing Company

To Donna and Ruth

This edition has been printed directly from the authors' manuscript copy.

Library of Congress Catalog Card Number: 84-52320

ISBN 0-8403-3523-7

Printed in the United States of America

B 403523 01

BASIC PROGRAMMING WITH APPLICATIONS IN BUSINESS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

SAMPLE PROGRAMS

Preface

This book was developed for use by individuals taking their first course in BASIC programming. No prior knowledge of any programming language is required. The objective of the book is to teach BASIC in the context of business applications. The sample programs that appear throughout this book were developed for use on a Digital Equipment Corporation Model PDP/11 computer with the RSTS/E operating system, but they will work on most computers that support BASIC with only minor changes. The emphasis in the book is on programming principles, not on any particular type of computer hardware.

Two essential types of computer systems appear in the business community today: a multi-user environment, where many people are served by the same computer processor; and a single-user environment, in which the processor and its input/output units serve only one person. Microcomputers are the driving force behind single-user systems. Both of these environments are covered in this book.

Chapter 1 describes the important role of computers in the business world and how human beings interact with the computer through system commands. Chapters 2 and 3 explain how BASIC statements are composed and how special symbols are used to process data values. Chapter 4 presents a systematic approach to program development. Chapter 5 describes various methods of input and output and Chapter 6 describes how a program is used to facilitate decision making. Chapters 7-9 describe several features of the BASIC language (loops, arrays, and subroutines) that can save the user a considerable amount of time when long repetitive procedures have to be programmed. Chapter 10 deals with processing data that is stored in disk files. Chapter 11 presents various techniques for sorting and searching. Chapter 12 describes some special functions built into the BASIC language which enable the user to easily derive various mathematical quantities and manipulate data that appears as text.

We would like to express our thanks to Dr. David C. Whitney of San Francisco State University for his encouragement and support. We are also grateful to Mr. Richard N. Thunes for his assistance in the publication of this book.

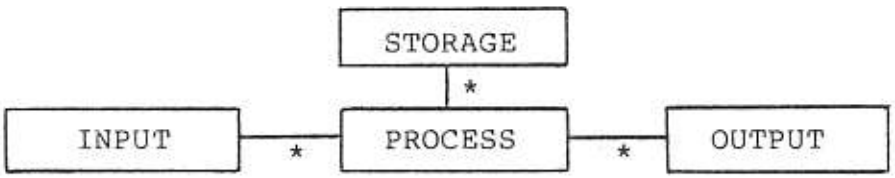
CHAPTER 1

SYSTEM INFORMATION AND COMMANDS

A. OVERVIEW

1. Computer Systems and the Business Organization

The general procedure that a computer follows can be summarized in a graphical manner by what is often called the Universal Flowchart:



* COMMUNICATIONS

Each part of this diagram represents both a function being performed and a specific piece of equipment that performs the function. The PROCESS function takes data from the INPUT function, transforms it in some way and delivers it to the OUTPUT function. In our case the input data are business transactions. These are processed in ways that will produce information to be used both internally (by management) and externally (by customers, creditors, suppliers, government, etc). Most transactions require access to some previously stored data which must be updated or referenced in some way, so the STORAGE function is added to the diagram. The equipment that performs the data processing function is collectively called hardware, and a separate hardware item is called a device.

The most common type of device used for each function is listed below:

Function	Hardware device
INPUT	Computer terminal keyboard
PROCESS	Central Processing Unit(CPU)
OUTPUT	Printer or terminal screen
STORAGE	Magnetic Disk, Magnetic Tape
COMMUNICATIONS	Computer Channel

It must be noted that the functions can be accomplished in several ways. For instance, INPUT may be accomplished by entering data on a terminal keyboard or by retrieving data that has been previously stored, or by both. The OUTPUT function may mean printing something on paper, placing data back into the storage area, or sending it to another computer system. Therefore the list above shows only the primary use of each device.

The function of COMMUNICATIONS involves getting data from one point in the system to another. The Universal Flowchart shows the communications function by simple lines connecting the system components The diagram does not indicate the physical distance between each component. In reality this distance can be across a room or halfway around the world. The hardware devices used for communcations are various types of sophisticated electronic equipment. The common term "channel" is used for the collection of equipment that carries that data, and it may include wire cable, a microwave transmitter, a fiber optic cable, satellites, or other devices.

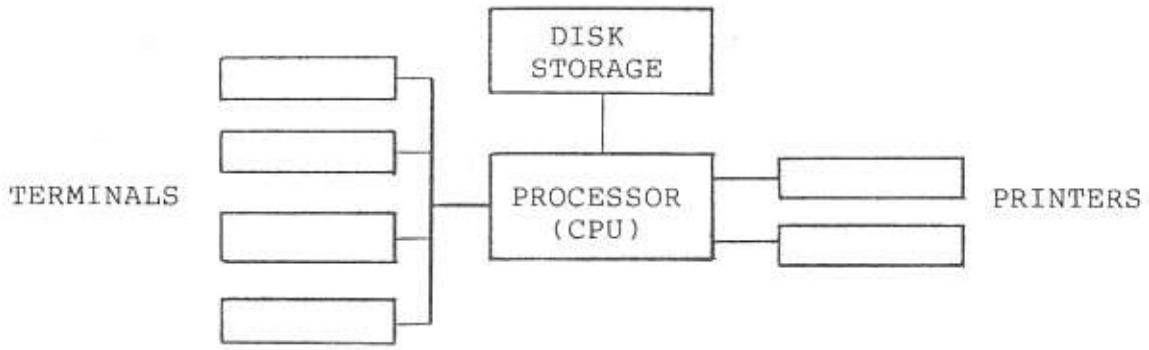
We have described what a computer system can do, but not how it is done. The hardware has many capabilities but cannot exercise any of them without instruction from a human being. An analogy can be drawn to the automobile. It is a large piece of equipment (hardware) that has the capability of moving people and cargo while traveling at great speeds. Also it can move in many directions. But it will not perform any of these "functions" unless a human being makes a decision on who is to ride, what cargo is to be carried, and which direction is to be taken. Even when the car is in motion performing these functions, it must be under constant control of the human being. So it is with the computer -- it has many capabilities but will not use any of them unless a human being "commands" that something be done. A business function may involve many steps and include many calculations, and the order in which these steps are taken is important. A computer program is the means that a human being uses to direct the computer in performing some function. The collective name for computer programs is software. The hardware devices that input and store data also input and store programs. A stored program can be called on again and again to process new input data.

A computer program is a set of instructions which must be carried out in a very specific order to accomplish some task or solve some problem. The general term for such a formal procedure is an algorithm. A program is an algorithm whose steps have been automated. A computer language is composed of a set of characters and a number of rules on how those characters can be put together to form computer program instructions.

Many computer languages have been created, each with its own set of characters and rules. Some languages are meant to serve specific applications and some are general-purpose. The language that is easiest for the beginning student is BASIC. Its name is an acronym that proclaims that fact: Beginners' All-purpose Symbolic Instruction Code.

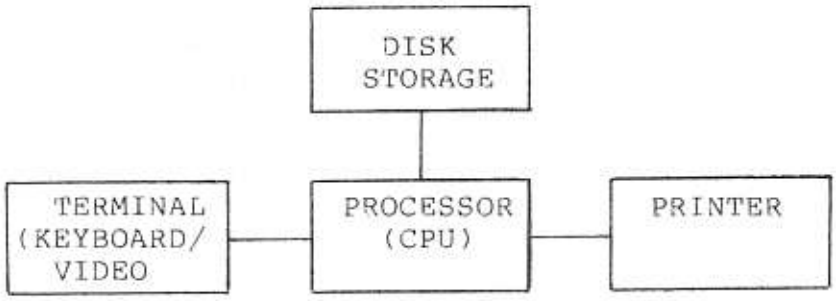
2. The Computer User Environment

Through the 1970s most computers were serving people in a multi-user environment. The capability of multiprogramming (storing more than one program in the processor at once) created the "timesharing" situation, where many users are served at the same time. Actually, a processor is concerned with only one program at any instant, but it can switch back and forth between programs at extremely fast speeds, giving each user the impression that the computer is dedicated to him or her alone. A typical multi-user computing environment might be diagrammed as follows:



The emphasis here is on shared resources. Each user shares the central processing unit (CPU), disk storage and output devices with other users in the system. In this environment management can keep accurate accounting records on how much of the computing resources are used by individuals and departments. They can also control the number of users and the type of programs that are run. The disadvantage of the multi-user system is that equipment malfunctions affect all users and valuable time can be wasted by waiting for other programs of higher priority to be completed.

The 1980s brought the microprocessor into general use. This "computer-on-a-chip" made computing devices relatively inexpensive while retaining much of the processing power of larger computers. Individuals could now work alone without having to share resources with others. The diagram of a typical stand-alone computing environment looks like:



The capability of user-independence was received with great enthusiasm by the business community, and microcomputers began to pour into the office. But the advantage of "personal" computers was eroded by other factors. There is very little data processed in the office by one individual that is not needed somewhere else in the company. Intra- and inter-departmental communication is essential for an ongoing business. A common data bank avoids the tremendous duplication of storage that individual computers can generate, not to mention the proliferation of storage media and devices. Microcomputers made by different manufacturers can present a nightmare of hardware compatibility problems, and management control in regard to security matters becomes unwieldy.

The current trend in business is to integrate these various office technologies. Microcomputers can be used as individual work stations, with limited processing and storage capabilities. At the same time they can be tied together or to a large "host" computer by means of a communication network. Thus company data resources can be shared by individual users. Being that these stand-alone computers are tied together into what is virtually a multi-user system, both user environments are mentioned in this book. The programming examples given in the chapters were developed on a timesharing system, but most of the BASIC statements apply to microcomputers as well. Appendix II gives a summary of the important BASIC sytem commands as implemented on the leading microcomputers.

B. SIGNING ON TO THE SYSTEM

1. Multi-user Environment

Since the multi-user system has a common storage area, there must be some way of telling one user from another. Something as simple as a different file name would not serve this purpose because many users may wish to use the same file names to store programs and data. The aspect of security is also important. What can be done to assure that a certain file is not accessed by an unauthorized user? The most common approach to this situation is to implement an accounting scheme similar to one used by banks. Each transaction in a bank requires that the customer supply two things, his account number and signature. The account number allows the bank to keep an accurate history of transactions and the current balance for each customer. The signature adds a certain level of security to the operation, because now an unauthorized customer would have to obtain someone's account number and forge his signature in order to make a transaction. This procedure is not foolproof, but works in most cases. The account number and "password" required to enter a multi-user computer system mimic the two items required for the banking transaction. The account number can then be connected to all files that the user stores and the password can act as a personal item, a "signature" that is known only by the user to which it was issued. This scheme, just like the bank's, is not foolproof. The ultimate responsibility of system security lies with human beings, in this case the company management and the individual users.

The terms logging on or signing on refer to the procedure of properly hooking up a terminal to the processor and being accepted by the system as an authorized user. The steps in the procedure will vary in number, type and order. In general they will include:

1) Power up. In most timesharing systems all terminals are left with their power on as long as the system is operating, so users usually need not be concerned with this step.

2) Communications link. A signal must be transmitted to the communications system to indicate that a new user wants to sign on. This may be as simple as pressing one of the keys on the computer terminal. Some data may have to be entered which describes such things as transmission speed and choice of computer (if more than one processor is available). Often there are more terminals than physical links to the computer and the user receives a message from the communication system that he or she must wait. The user's entry request is then placed in a queue (a waiting line).

3) Enter account number and password. After a communication line is established, a set of computer programs collectively called the operating system takes control of the computer session. The user is now considered to be in the system mode. The operating system prompts the user for an account number and password and checks the data entered against an official list that is stored in the data bank. If there is a match the user is considered "signed on."

4) Specify computing mode. The system mode is the starting point for entering a number of different computing modes. The user may want to enter a program in a certain language (like BASIC), edit files, delete files, or other tasks. The user must specify which mode is desired. Many timesharing computers are set up to go directly into the BASIC mode, making this step automatic for the user. Once the user is in the proper mode (in our case BASIC) he or she can write new programs, change old programs, or execute a program to process data for some application.

2. Single-user Environment

A stand-alone, or self-contained, computer serves only one user. All the hardware components (terminal, CPU, printer, storage device) are "dedicated" to the user of the system. The single user system we will discuss is the microcomputer, or "personal" computer. We will assume that the type of storage medium used is a small flexible ("floppy") disk. The administrative tasks that must be performed in order to use a microcomputer are not as involved as a timesharing system. Since there is only one user, an account structure is really not needed. All files are accessible by the person that logs on the system. Security in this case is usually left entirely up to the owner of the microcomputer. If a number of these computers are owned by a company, then management must take special care that only authorized personnel have access to them. The log on and log off procedures are quite simple:

1) Power up. The power must be turned on at the beginning of each session. Leaving the power on for extended periods of time when the computer is not in use is considered an inefficient use of the computer hardware.

2) Load operating system. A "system disk" containing the operating system must be inserted into one of the disk drive units. This disk comes with the computer when it is purchased. A special key on the terminal is pressed which loads the operating system into processor memory. This is called "booting the system." An operating system program then prompts the user for the date and time.

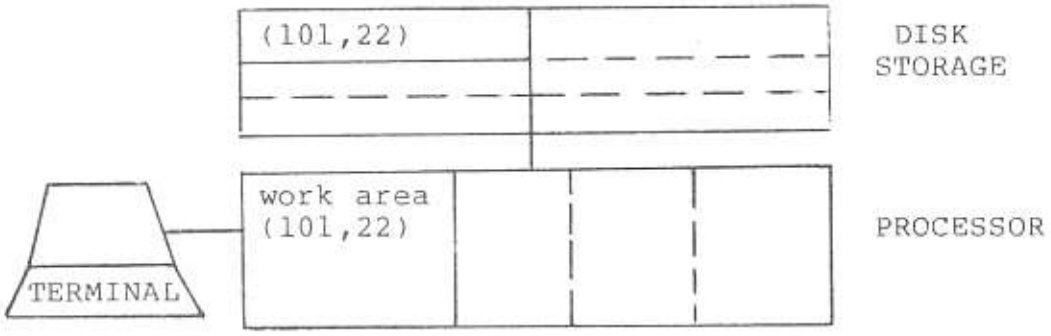
3) Specify computer mode. The user specifies the computer mode as described earlier (BASIC programming, text editer, etc). The operating system may have to load other programs in from storage to bring the user into the requested mode.

C. INTERACTING WITH THE COMPUTER

1. Temporary and Permanent Storage

Most people using a computer for the first time meet a roadblock after they have logged on to the system. Entering the account number and password is done by following very specific instructions, but after these are completed the question "Where do I go from here?" usually arises. The beginner is often confused as to whether he or she should initiate some action or if the computer is going to do something instead. As mentioned previously, the computer has the capability to do many things, but will not take any action unless directed to do so by a human being.

To understand what should be done after logging on, we must first take a closer look at the processor memory. The user's account number was accepted and verified by the operating system, the set of computer programs that control and coordinate all internal computer activity. The operating system has now reserved a part of the processor memory called a work area for exclusive use by the person sitting at the terminal and has identified this memory area with the account number entered. Only a part of the processor memory is assigned because, in a timesharing system, the memory must be divided among many users. The individual user, however, is not directly affected by the other users and for all practical purposes can assume that this part of memory is the processor memory. To visualize this we will draw the timesharing system diagram again as it would appear right after someone has logged on. For the example let us say that the user's account number is 101,22.



The work area is a temporary storage area. As its name implies, it is the place where programs are "worked on", that is, created, changed and executed to produce output. When the user logs off, the work area will be cleared and given to another user. Therefore some place is needed to store data and programs when the user is not on-line. The disk acts as a permanent storage area. An analogy can be drawn to a manual data processing system (one without computers). A person removes documents from a file cabinet, works on them while sitting at a desk, then returns them to the file cabinet. The desk top, in this case, is a temporary storage area and the file cabinet a permanent storage area.

2. System Commands

All of the system commands in some way deal with the work area. Programs are created in the work area, are saved by moving them from the work area to the disk, and are retrieved by moving them from the disk back into the work area. Commands are not part of a computer program. They are directives to the operating system to take some specific action with a program. Commands are always executed immediately by the operating system. The words "computer" and "operating system" are often used interchangably, for instance, "the computer executed a command."

The lines of a computer program, called statements or instructions, will be described in Chapters 2 and 3. It is necessary to complete the first three chapters of this book in order to fully understand the interaction between system commands and program statements. The most important system commands are explained below in the order that a beginning programmer would be most likely to use them.

a) Naming and Saving Programs

Before entering a new program, the NEW command is given. This clears the work area of any program lines that may be present and assigns it an identifying label. The general format of the NEW command is

```
NEW name
```

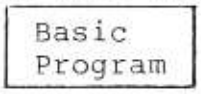
where "name" is a set of alphanumeric characters (letters or numbers). The first character must be a letter. The name may contain six characters maximum. Some examples:

```
NEW PAYROL
NEW XYZ1
NEW PROG
```

Note: It will be implied that at the end of every line typed on the terminal the carriage return key must be pressed. This key is usually marked: RETURN.

If the command NEW PROG was entered, the work area would be named PROG and it would be empty. This is like puting a label on an empty file drawer. Naming the work area in this way is significant because later, after computer program statements have been entered, the contents of the work area will be saved on the disk, and the the name of the work area will become the disk file name. The computer always responds to a command, so after NEW PROG is entered the word Ready appears on the video screen, signifying that the command has been performed.

At this point the user would start entering lines of a BASIC program. Since only commands are being described in this section we will indicate a BASIC program by the symbol:



When all program lines are entered, the user may want to save these lines on the disk so that they may be retrieved and used again later. The SAVE command will accomplish this task. Only the word SAVE has to be entered since the file that is saved on the disk will have the same name as the work area.

We will summarize the discussion above by listing the commands and computer responses in their proper sequence. Let's say the user has logged on under account 101,22 and wishes to enter a program and call it PROG.

	Comments
NEW PROG	work area is named
Ready	computer response
Basic Program	program lines are entered
SAVE	program is saved on disk
Ready	computer response

It is important to note that when the SAVE command is given, a copy of the program in the work area is moved to the disk. Therefore the program is still in the work area.

Very rarely will a program produce correct answers the first time, so changes will have to be made to program lines. The rules for making such changes (called editing) are discussed in the next chapter. When changes have been made the user will want to store the new version of the program on the disk and delete the old version. This can be done with the REPLACE command. This command is given when the user wants to take the program in the work area and store it on the disk under a file name which already exists.

But how does one retrieve a program that has been stored? If the user logs off the system and returns the next day to make changes to a program, the disk file must be brought into the work area. this is accomplished by the OLD command. Its general format is:

OLD filename

The computer will then move a copy of the program "filename" from the disk into the work area, and the file name will then be the name of the work area.

Some general rules can now be listed for storing and retrieving programs:

For a new program:
- Name the work area with NEW name
- Enter the computer program statements
- Store the program on the disk with SAVE

For a program which is already stored on the disk:
- Move a copy of the program into the work area with OLD name
- Edit the program lines
- Store the new version of the program on the disk with REPLACE

The RENAME command is not used very often but it is mentioned here because it can be helpful in some cases. The general form is:

RENAME filename

The command will change the name of the work area to "filename." If a user wanted to save both the current version of a program and an updated version, RENAME could be used. The complete procedure would be:

- Bring the program into the work area from the disk
- Rename the work area
- Make necessary changes to produce new version
- Save the work area on the disk

As an example, say a file called PROG existed on the disk and a user desired to make changes to the program and save the new version under the name PROG1. The following sequence of commands would be needed:

	Comments
OLD PROG	Retrieve current version
RENAME PROG1	Rename work area
	Edit program lines
SAVE	Save new version

Note that the SAVE command was used, not REPLACE. We are saving the work area under a name that did not exist before, thus it is a new entry in the disk catalog and SAVE is proper.

b) Listing and Executing Programs

An important distinction must be made between the work area and the video screen on the computer terminal. They are not the same thing! The amount of data that can be stored in the work area is much larger than what will fit on the screen. We will define a line typed by the user (a command, program statement, etc) as an "entry", and any information delivered by the processor to the screen as a "response." The video screen is only a moving "log" of transactions, that is, it shows only the latest set of entries and responses that have occurred. Quite often a user will enter a program line and later discover that there is an error in the statement. If the line is re-entered then the screen will show both old and new lines, but only one of them (the latest one) will remain in the work area. When the screen fills up, it looks as if the lines disappear off the top. This does not mean that the work area is full, only that the screen must make room for the next transaction to be displayed.

After a number of program lines have been entered, and various changes have been made, how does one see what is actually in the work area? This is the purpose of the LIST command, It brings a copy of the program in the work area to the video screen. In this case the screen only acts as a "window" into the work area. If the program has more lines than will fit on the screen then each new line will be displayed at the bottom of the screen and one line will move off the top. But this does not mean that the program has been altered in any way. Beginning users are often fearful that LISTing a program will cause it to leave the work area. Remember, the LIST command only brings a copy of the program in the work area to the screen.

There are several options that come with the LIST command. These are presented in a general form as:

```
            LIST
or          LISTNH
or          LISTNH n-m
```

LIST will display the entire program

LISTNH will "list with no heading (NH)". This is an option designed to save the user some time. When the command LIST is entered, the computer will display the name of the program, the date and the time before starting to list the program lines. The LISTNH causes the computer to omit this heading information. It does not affect the work area in any way. The LIST command is usually used with this option.

LISTNH n-m will list only a specific range of line numbers, beginning with line "n" and ending with line "m". For example:

```
LISTNH 100-500
```

This command will display program line numbers 100 through 500. The heading information will not appear on the screen.

Since the composition of computer statements will be discussed in the next chapter we will not mention anything more about them here. Remember, a proper understanding of how system commands and computer program statements interact cannot be achieved until all the information in Chapters 1-3 has been presented, so the reader is encouraged to study these chapters as one unit.

The lines of a program can be entered and saved, but eventually users will want the program to do something. This is called executing the program. The command that starts execution is RUN. This will display the actual output of the program. Entering and listing program lines is similiar to writing down the various steps that go into the recipe for making a cake. Simply writing down the instructions does not produce a cake! After obtaining the recipe, one must actually perform (or execute) each instruction. The LIST command only displays the program statements. RUN will display the program output (the results of calculations, report headings, list of data items, etc). The command also has the no-heading option (RUNNH). This saves some more time since the program name, date and time would be displayed if only RUN were entered. RUNNH will be used with all the examples in this book.

Note on Hardcopy Output

We have used the word "display" to mean copying program lines or output from the work area to the video screen. How does one get a "hardcopy" listing, that is, a listing on a piece of paper? This depends on the type of terminal and printer being used and how they are connected to the processor.

In a multi-user environment, the printing devices are usually shared by many users. Since only a few users will be printing (hardcopy output) at any given time, a group, or cluster, of terminals may be connected to a single printer. A printer switch will be placed on each terminal. When this switch is in the "on" position the terminal is connected to the printer, and anything displayed on the terminal screen is simultaneously printed on paper. This includes both the program listing (from the LIST command) and program output (from the RUN command). When the printout is finished the user turns the printer switch to the "off" position, disconnecting the terminal from the printer. Obviously only one person at a time can use the printer. There is often a pilot light next to the printer swith. If this light comes on when the switch is turned on, then the user knows that the printer is available. If not, the user must wait until the printer is free.

In a single-user environment there is usually a printing device connected directly to the processor, so no printer switch is needed. Hardcopy listing of program statements are produced with a single command. On most microcomputers this command is:

```
LLIST
```

This will printout the statements of the program in the processor memory. Getting a hardcopy listing of program output is usually accomplished with a special PRINT statement in the BASIC program. See Chapter 5 for details.

c) Deleting Lines and Programs

An entire group of computer statements can be deleted from a program by the DELETE command. The general form is:

```
DELETE n-m
```

which deletes all lines from line n through line m. For example:

```
DELETE 100-500
```

will delete all lines from line 100 through line 500.

Making changes within an existing program statement is called editing the program text. There are various methods of editing, some of which are discussed in Chapter 2 (see Fixing Mistakes).

Deleting and editing computer statements has to do with programs that are in the work area. How does one delete a program that is stored in the disk catalog? Even if the user brought the program into the work area and deleted all the lines, the filename would still exist in the catalog. This deletion task can be accomplished with the UNSAVE command. The general form is:

```
UNSAVE filename
```

The work area is unaffected by the command. No transfer of program statements or data is made between the processor and the disk. The file "filename" is simply deleted from the disk and from the user's catalog listing.

d. Disk Catalog

The user will naturally save various programs on the disk, and after awhile it may be difficult to recall specific file names. The CAT command will display a list of file names that are on the disk under the user's account number. The form of the command is simply:

CAT

The command does not affect the work area so it can be used at any time. Various data are listed with the catalog. The file name is shown as part of a complete "file specification", which has the general form "filename.extension." The extension is a set of three characters that can be used by the programmer as a second level of identification. Each BASIC program file will list in the catalog with the extension "BAS." For example if the user had saved a program called PROG it would list in the catalog as PROG.BAS. The user does not have to be concerned with the extension in any of the commands that have been described so far, since they all deal with program files. The extension will be discussed in more detail in Chapter 10 (Using Data Files).

D. CONTROL KEYS

Control keys are somewhat like commands, but they do not cause data to move in or out of the work area. It is often desired to stop a program that is listing or executing before the command is completed. For example if the LIST command were entered and there were many program lines, it may take a long time before the entire program is displayed or printed. If the user decided after the command had been given that he or she did not want to see the entire program displayed there would normally be no way of stoping the process. In another situtation the RUN command may be entered and the user sees from the first few lines of output that there are errors in the program. There should be some way of avoiding the long wait until all of the output has been displayed.

The control keys are useful in these situations. the CONTROL-C will halt a program that is listing or executing, and the operating system will simply wait for a new command. A CONTROL-C is entered by holding down the key marked CONTROL and pressing the letter C. The control does not affect the work area, so it is safe to use at any time.

The user may wish to start a listing or run, halt it for awhile, then resume the process. This can be accomplished with the control keys CONTROL-S and CONTROL-Q. The first will stop the display of program lines on the screen that was started with the LIST command and the latter will resume the listing where it left off. The format is the same; hold the CONTROL key down and press S or Q. The same applies to program output (after the RUN command has been entered). When a control key is pressed, it will appear on the screen as ^C,^S or^Q.

E. SIGNING OFF THE SYSTEM

1. Multi-User System

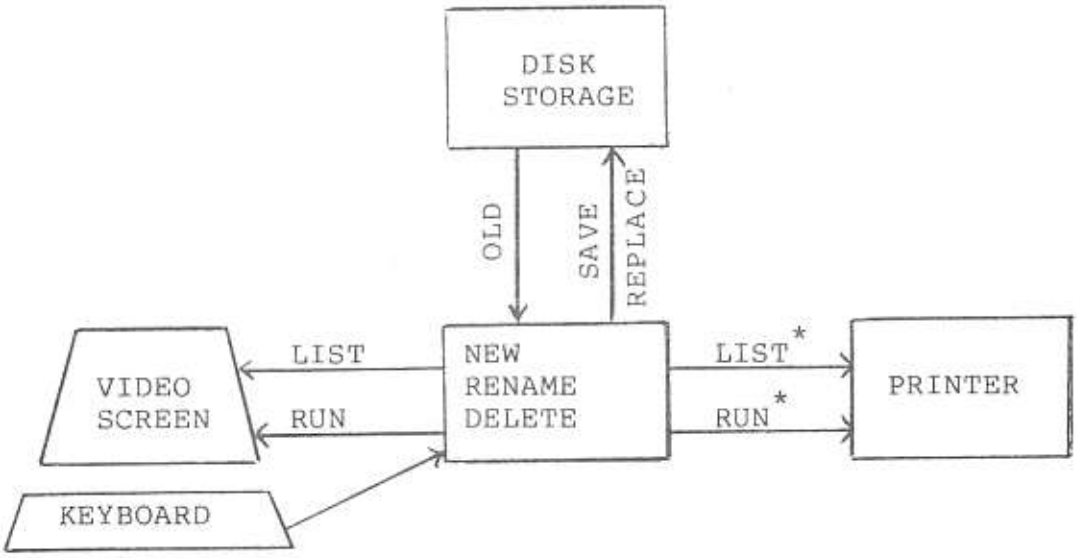
After all work is finished and the user is ready to relinquish the terminal, a message must be sent to the operating system that the computer session is to end. This process is called logging off or signing off. It is usually done with a simple command (such as BYE) typed on the terminal. The operating system will display some accounting information, such as the total elapsed time and total processor time, place the user's account number on an inactive status, and return control to the communications system which waits for another user.

2. Single-user System

After all work is finished the user logs off by simply removing the disks from the disk units and shutting off the power.

F. SYSTEM COMMAND SUMMARY

The commands described so far are listed on the system diagram to show which hardware devices are affected in each case:



* When the printer switch is on, LIST and RUN will produce a printed output in addition to the display output.

SYSTEM COMMAND SUMMARY

Naming and Saving Programs

NEW name - Assigns a label of "name" to the work area. Deletes any program lines currently in work area.

SAVE - A copy of the program in the work area is saved on the disk catalog.

REPLACE - Previous version of program on disk is replaced by program in work area.

OLD filename - Moves a copy of program "filename" from the disk catalog into the work area.

Listing and Executing Programs

LIST - Displays all program lines currently in work area. Listing preceded by program name, and date/time. LISTNH option will omit heading information. If printer switch is on, hardcopy output will be produced while program is being displayed.

LIST n-m - Displays program lines n through m. Heading information displayed. LISTNH n-m option permitted.

RUN - Executes program in work area. Program name, date and time are displayed. RUNNH will omit heading information. If printer switch is on, hardcopy output will be produced while program is executing.

Deleting Lines and Programs

DELETE n-m - Delete lines n through m from the work area.

UNSAVE filename - Remove file "filename" from the catalog.

Disk Catalog

CAT - Displays all filenames currently in the user's disk catalog. Work area is unaffected.

Control Keys

CONTROL-C - Terminates program listing-in-progress or program execution-in-progress. Work area is unaffected.

CONTROL-S - Halts program listing-in-progress or program execution-in-progress. Press CONTROL-Q to continue.

CONTROL-Q - Resumes listing or execution of program that was halted by CONTROL-S.

CHAPTER 2

BASIC STATEMENTS

A. OVERVIEW

Chapter 1 described how commands were the vehicle for moving programs in and out of the work area. In this chapter we look at how actual program statements are constructed. Each program statement must have a line number or statement number. This is an essential difference from system commands, which are acted on immediately by the operating system. If a line is entered that begins with a number, the operating system will accept this as a BASIC statement and simply place it in the work area. It will keep accepting other statements until a command is given to do something with the statements (remember the analogy to the cake recipe in the last chapter). Therefore statements do not produce any direct action by the computer. All the statements in the work area are acted on as a group when a command is given (RUN, LIST, SAVE, etc).

1. General Form

All BASIC statements have the general form:

line # keyword or keyphrase parameters

line #: Establishes implicit sequence for program execution. Required for every new program line.

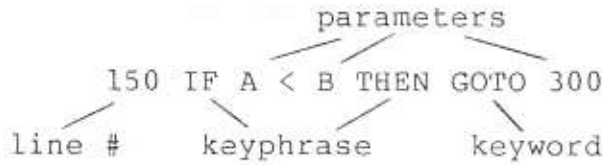
keyword or keyphrase: Establishes the instruction type (operation) that is to be performed.

parameters: Data items that are to be involved in the operation.

Example:

Statements are sometimes formed by a series of keywords or keyphrases:

Example:



2. General Categories

Program statements fall into two general categories: executable and non-executable. There are various statement types within each of these categories. Executable statements will be referred to as instructions, since they result in some definite action on program data or the order in which subsequent instructions will be executed. Nonexecutable statements do not affect data values or program control directly, but must be included to support the function being performed by certain executable statements.

The tables below list each statement type, the BASIC keywords or keyphrases included in that type, and the chapter number in this book where the BASIC statements are explained in detail. The tables are followed by a general description of the functions performed by each statement type.

a) Executable Statements

Statement Type	BASIC keyword or keyphrase	Chapter Reference
Assignment	LET or =	Ch 4
Input	READ, INPUT, INPUT #	Ch 5,10
Output	PRINT, PRINT USING, PRINT #	Ch 5,10
Decision	IF-THEN-ELSE	Ch 6
Transfer	GOTO, ON-GOTO, ON-GOSUB	Ch 6
Predefined Process	FOR-NEXT, GOSUB-RETURN, DEF	Ch 7,9
Program Termination	END, STOP	Ch 2,5
File Control	OPEN, CLOSE	Ch 10

General Description:

Assignment - Associates a constant with a variable. A constant, usually called a data value, can be a number or a string of characters. The values can be located in a data storage area or can be the result of an operation (numeric calculation or character manipulation).

Input - Retrieves data values from an external source (the terminal keyboard or a disk file) and associates them with program variables.

Output - Moves data values from program variables to an external storage area (disk file), to the display screen, or to the printer.

Decision - Compares two values and takes specific action based on the results of the comparison.

Transfer - Directs the program to alter its normal sequence of statement execution.

Predefined Process - Allows the user to combine several of the functions listed above into one program statement.

File Control - Allows the user to access data files stored on the disk catalog.

b) Nonexecutable Statements

Statement Type	BASIC keyword or keyphrase	Chapter Reference
Specification	DIM	Ch 8
Data Storage	DATA	Ch 5
Documentation	REM, !	Ch 2

General Description

Specification - Reserves processor memory locations for specific program variables.

Data Storage - Provides an internal storage area for data values that will later be associated with variables when the program is executed.

Documentation - Allows programmer to include, within the program statements, an expanded description of the specific function being performed by each instruction.

B. NUMBERING AND EDITING STATEMENTS

1. Line Numbers

a) Placement

A line number is a positive integer that must be placed at the start of each new program line. A distinction must be made between video terminal lines and program lines. The video line is simply the number of characters that will fit on one line of the video screen (usually 40, 64 or 80). This is not the limitation of a program line, which may extend over several video lines. A program line is continued on the next video line by insertion of the line feed character. This can be accomplished in two ways:

1) Implicit line feed. If the programmer keeps typing at the end of the video line, the computer will automatically insert a line feed character (not visible on the screen) and continue on the next video line. For example:

```
100 PRINT "THIS IS AN EXAMPLE OF A PROGRAM LINE THAT IS
TOO LONG TO FIT ON ONE VIDEO SCREEN LINE"
```

2) Explicit line feed. If the programmer wishes to place one program line on several video lines for the purpose of clarity, a line feed may be inserted by pressing the key marked LINE FEED (the down arrow on some terminals). For example:

```
100 PRINT "THIS IS AN EXAMPLE OF A
PROGRAM LINE THAT IS TOO LONG TO
FIT ON ONE VIDEO SCREEN LINE"
```

A program line ends when the carriage return key (usually marked RETURN is pressed. When programs are listed in this book, it will be implied that each program line ends with a carriage return.

b) Limits

The range of line numbers is 1 to 32767. The upper limit is a function of computer design and may be different from one system to another.

c) Multiple Statements

Several computer statements may be placed on the same program line. The statements are separated by the colon (:) or backslash (\). For example, the program lines

```
10 A = 2
20 B = 3
30 C = A + B
```

could be rewritten as

```
10 A = 2: B = 3: C = A + B
```

Note that there is only one program line number. The rules for program line continuation still apply if one of the multiple statements extends past the end of the video line.

d) Execution and Terminal Entry

Statements are executed in increasing order by program line number, unless specific program instructions are given to do otherwise. For example:

```
10 A = 2
20 B = 3
30 C = A + B
40 PRINT "THE SUM IS",C
50 END
```

If this program were executed the following sequence would be followed:
The variable A is assigned a value of 2 (line 10)
The variable B is assigned a value of 3 (line 20)
The values of A and B are added and the result is assigned to the variable C (line 30)
The value of C is printed (line 40)
The program ends (line 50)

The computer work area will automatically put statements that are entered in increasing order by line number. Thus program lines can be entered in <u>any</u> order. For example, if the previous program were entered as

```
50 END
40 PRINT "THE SUM IS",C
30 C = A + B
20 B = 3
10 A = 2
```

and the user then typed the command LISTNH (which brings the contents of the work area to the screen), the program lines would be displayed as before, in increasing order.

e) Increment

The increment between line numbers is whatever the programmer desires, but it is strongly suggested that an increment of at least 10 is used, since additional lines are often inserted between existing lines. Even though most computers have a line renumbering function, the user often desires to have specific line numbers at different parts of the program, and a line renumbering operation will disrupt the desired line numbering scheme.

2. Fixing Mistakes

a) Backspacing

If the user is typing a program line, and the carriage return has not been entered yet, the cursor may be backspaced, just as on a typewriter. The last character on the screen will be deleted and a new character can be entered. The key which accomplishes the backspace function varies between terminal manufacturers. Some common examples are:

RUBOUT
DEL
BACKSPACE

b) Line Editing

If the user has already entered a carriage return, the program line can be changed in several ways:

1) If the line is short, simply type in the line again. There is only one space in the work area for each unique line number, so re-entering a certain line will overlay the previous version of the line. It is <u>not</u> necessary to re-enter all previous line numbers. For example, if the program listed above had been entered and the user desired to make a change in line 10, he or she would simply enter the new version of the line. A subsequent LIST command will show the new version of the line. A single line can be deleted by by typing the line number, then pressing RETURN. A group of line numbers can be deleted with the DELETE command.

2) For extensive line editing, a separate editing program must be used. This is usually provided as a utility of the operating system, and is implemented differently from one computer system to another. An editing program actually performs a word processing function, considering the program lines as text. Any character or group of characters within a line can be accessed directly and changed. Consult the manufacturer's system manual for details on the editing programs.

3. Program Termination

The STOP statement has the same effect as a CONTROL-S command that is entered while a program is being run. The program is halted until the user gives the proper command to continue. The difference lies in the method used to halt the program. The user can issue a CONTROL-S at any time and for any reason. The STOP statement causes the program to halt for some logical reason, that is, because of some decision made within the program. The programmer can set up certain criteria for deciding whether or not to STOP a program. If the program is under development, the programmer may want to check intermediate relults, then continue with the rest of the program. The STOP statement can be placed at strategic points in the program and it will always halt at the same places automatically. When a program being executed encounters the STOP statement it will display a message indicating the line number where the program was halted. For example:

```
STOP at line 270
```

The program execution is resumed by typing the word CONTINUE and then pressing the carriage return. There can be as many STOP statements in a program as desired.

The END statement indicates the physical end of the program listing. It is always the last statement in the program and thus always has the highest line number. If a program is expanded beyond its present length, the current END statement should be deleted, then re-entered with a proper line number. A missing END statement or more than one END statement may cause problems when the program is executed or when the user is saving the program on disk storage.

4. Documentation

Internal program documentation is very important. Programs use many symbols and program logic can get complex. The programmer needs to keep track of the meaning of each symbol and what function is being accomplished by each program section. Program documentation is similar to notes written in the borders of a textbook. These are informal notes to explain the text in a manner more understandable to the reader or to highlight certain points.

Program documentation can be accomplished in two ways: the REM statement and the exclamation point (!), both of which are non-executable statements.

The general form of the REM statement is:

line # REM comments

The line number and the word REM are mandatory. The comment can be any characters or symbols that the programmer desires; they have no effect on program execution. The exclamation point can be placed on any program line following the BASIC statement. After skipping at least one space, press the ! key. The rest of the line is considered a comment. For example, the previous program is rewritten with both types of documentation:

```
10 REM  THIS PROGRAM ADDS TWO NUMBERS
20 A = 2          ! ASSIGN VALUES
30 B = 3
40 C = A + B      ! COMPUTE SUM
50 PRINT "THE SUM IS",C   ! PRINT RESULT
60 END
```

Note: In most microcomputer systems the apostrophe (') is used instead of the exclamation point to indicate a remark statement.

C. BASIC STATEMENTS AND SYSTEM COMMANDS

The examples given in this chapter show only BASIC statements but do not show what will happen if those statements are executed. From now on all examples of complete programs will be follwed by the RUN command (with the no-header option, RUNNH) so the reader can see the resulting output.

As an example the program listed above will be executed. The following lines show every transaction (entry and response) that would occur after the user logged on the system:

```
                                    Comments
NEW PROG                            Work area is named PROG
Ready                               Computer response
10 A = 2                            Program lines are entered
20 B = 3
30 C = A + B
40 PRINT "THE SUM IS",C
50 END
RUNNH                               Command to execute program

THE SUM IS 5                        Program output

Ready                               Computer response
```

At this point the user can take any desired action -- add more program lines, save the program, retrieve an old program, or whatever. The operating system will assist the user in completing any action that is requested.

CHAPTER 3

PROCESSING DATA

A. OVERVIEW

A distinction must be made between a data value and the context in which that value appears. For example, if someone said "23" you would not know what they were talking about since that number could be connected with many things. But if someone said "John is 23 years old," it would place the number in a specific context. The data name (also known as the data field) is the context for a data value. The data name, unlike the data value, usually has meaning by itself. Referring to the example above, the phrase "John's age" does have a specific meaning to us, whereas "23" by itself does not. Data names only have to be associated with data values when the values are to be processed in some way. To make this clear we will take an expanded context -- several data names that are related by a mathematical formula:

NET SALARY = GROSS SALARY - DEDUCTIONS

Surely the meaning of this is clear to everyone, and no numbers are necessary. But if someone where to ask "What is John's net salary?" then we would need some numbers to associate with the terms in the above equation. If John's gross salary and deductions were $200 and $150 respectively, given the above context we could figure out his net salary. Therefore the context (data name) must exist and have a meaning before any data values can be processed.

Another important point about data is that given a certain context, many different values can be processed without changing that context. Given the equation above, we could also use it to figure out Frank's net salary if we had his gross salary and deduction values. Thus a context can be associated with many different values, but only one at a time.

In programming the data names (context) are called variables and the data values that they are associated with are called constants. The assignment statement is one way of accomplishing this association. The general form is:

line # LET variable = <expression>

- The keyword LET is optional in most versions of BASIC and will not be used in the examples of this book. Therefore the statement:

  100 LET A = 20

  can be written as:

  100 A = 20

- The rules for naming a "variable" will be introduced in the following section.

- The <expression> is any group of constants, variables, numeric or string expressions that are to be evaluated. The process of forming these expressions is explained below.

B. CONSTANTS AND VARIABLES

1. Constants

a) Numbers

Data values that are quantitative in nature are called numeric constants. Numbers can be represented as integers or decimals.

Integers are whole numbers (written without a decimal point).

Examples: 2 35 3487

In most computers integers are only allowed in the range -32768 to 32767.

Decimals are numbers written with a decimal point.

Examples: 24.36 0.023 5702.98

A decimal may also be represented in scientific notation, which has the general form

value E n

where E n means "times 10 raised to the power of n." As an example, the number 1874.32 could be represented as 1.87432E3, which means 1.87432 X 1000, or 1874.32.

Notes on Decimal Values
The allowable range for decimal numbers may vary among computer systems. Most computers have a limit as to the number of significant digits that can be represented. If more than this limit is entered, the computer will round off the number and represent it in scientific notation. Some computers will allow double precision decimals, which expand the range and the number of significant digits allowed. Check the system manuals provided by the manufacturer for details.

b) Strings

Qualitative or descriptive data values are called string constants. These are alphanumerics, any character or group of characters that together are considered a data value. The word "character" means any letter, digit or special symbol. String constants are always enclosed in quotes.

Examples:
```
"TOTAL"
"ABC"
"100 PENNSYLVANIA AVE"
"*** PAYROLL REPORT ***"
```

2. Variables

As mentioned before, constants must be associated with some context. Variables take on values of numeric or string constants. At any given time a program variable has only one value. Values are associated with variables in several different ways:

(1) Reading a value from an internal data storage area.

(2) Inputting a value from the terminal keyboard.

(3) Inputting a value from a disk file.

(4) Direct assignment.

(5) Assignment as the result of a calculation.

Examples of these methods are listed below for both numeric and string variables. The proper use of each method will become evident as more sample programs are developed throughout the book.

a) Numeric

A numeric variable is named by a single letter, or a letter followed by a single digit:

Examples: A X B3 C0 X9

Example of assigning numeric constants to numeric variables in BASIC are given below:

(1) Reading from internal data storage:
```
10 READ A
20 DATA 3
```

(2) Input from terminal:
```
10 INPUT A
```

(3) Input from disk file:
```
10 INPUT #1, A
```

(4) Direct assignment:
```
10 A = 3
```

(5) Assignment by calculation:
```
10 A = 3+2*7
```

Note:
Numbers never appear with commas in computer program statements (e.g. 2,364 4,568,936 etc). Thus A = 2100 is never entered as A = 2,100. It is possible to specify that a number appear with commas when it is output (displayed or printed). This will be explained in Chapter 5

b) String

A string variable is named in the same way as a numeric variable, but must be followed by a dollar sign.

Examples: A$ X$ B3$ C0$ X9$

X$ is pronounced as "X string". Examples of assigning string constants to string variables in BASIC are given below:

(1) Read from internal data storage:

```
10 READ A$
20 DATA "JOHN SMITH"
```

(2) Input from terminal:

```
10 INPUT A$
```

(3) Input from data file:

```
10 INPUT #1, A$
```

(4) Direct assignment:

```
10 A$ = "JOHN SMITH"
```

(5) Assignment by calculation.
The only "calculation" allowed on strings is concatenation (see section D of this chapter)
Example:

```
10 A$ = "JOHN" + "SMITH"
```

c) Subscripted Variables

When many data items are to be processed and each item must be assigned to a separate variable, the resulting list of data names becomes very difficult to manipulate in program statements. A computerized list, called an array, is composed of a number of memory storage locations, each of which can hold a data value. The context (data names) for these values are called array variables or subscripted variables. Arrays can be either numeric or string. There are special rules governing the use of these variables and they will be discussed in detail in Chapter 8.

d) Extended Variable Names

Many computers today allow BASIC variable names in an extended mode, that is, up to 30 characters can be used for any numeric or string variable name. The program examples used in this book use the most restricted mode of variable names so that they will apply to computers that do not feature the extended mode. If extended names are allowed, then BASIC statements such as

```
100 S = B + C
```

could be written as

```
100 SALARY = BASE + COMMISSION
```

C. ARITHMETIC OPERATIONS

1. Arithmetic Operator Symbols

Symbol	Function	Sample BASIC Statement
+	Add	10 X = A + B
-	Subtract	10 X = A - B
*	Multiply	10 X = A * B
/	Divide	10 X = A / B
** or	Exponent	10 X = A ** B
		10 X = A B

Note: The asterisk (*) is the only valid multiplication symbol. Other notations used in algebra, such as AB, A X B, and A(B) are not valid in BASIC for this purpose.

Variables may appear on the right hand side of an equal sign, as shown in the BASIC statements above. In a complete program each variable used in an equation must have previously been assigned a value. The program that follows should make this clear:

```
10 A = 2
20 B = 3
30 C = A + B
40 PRINT "THE SUM IS";C
50 END
RUNNH

THE SUM IS 5

Ready
```

In statement 30, the expression C = A + B means "add the value of the variable A to the value of B and assign the result to the variable C." If numeric variables appearing on the right hand side of an equal sign do not already have assigned values, most computer systems will assign them a default value of zero.

2. Order of Operation

Numerical expressions are evaluated according to a certain priority of the arithmetic operators:

Highest priority:	Parenthesis	( )
	Exponent	**
	Multiplication and Division	* /
Lowest priority:	Addition and Subtraction	+ -

Nested parenthesis ((expression)) are evaluated with the inner parenthesis having the highest priority (i.e., from the inside out).

Multiplication and Division have the same priority, as do Addition and Subtraction. If two operators with the same priority appear next to each other, they are executed from left to right.

Examples: Several BASIC statements will be listed, followed by the individual steps taken by the computer to evaluate the numerical expressions involved.

```
BASIC statement:  100 A = 7 + 3 * 2**2 / 3 - 1

Step 1:  A = 7 + 3 * 4 / 3 - 1          (Exponentiation performed)
Step 2:  A = 7 + 12 / 3 - 1             (Multiplication performed)
Step 3:  A = 7 + 4 - 1                  (Division performed)
Step 4:  A = 11 - 1                     (Addition performed)
Step 5:  A = 10                         (Subtraction performed)

BASIC statement:  100 A = 7 + ((3 * 2)**2 / 3) - 1

Step 1:  A = 7 + (6**2) / 3 - 1         (Inner parenthesis )
                                        (Multiplication performed)
Step 2:  A = 7 + (36/3) - 1             (Exponentiation performed)
Step 3:  A = 7 + 12 - 1                 (Division performed)
Step 4:  A = 19 - 1                     (Addition performed)
Step 5:  A = 18                         (Subtraction performed)
```

3. Mathematical Notation

Several algebraic equations will be listed together with equivalent BASIC statements to demonstrate use of the arithmetic operators:

Algebraic Equation	BASIC Statement
$z = (a + b \cdot c)^2$	`100 Z = (A + B * C)**2`
$z = \left(a + \frac{b \cdot c}{2}\right)^2$	`100 Z = (A + B * C/2)**2`
$a = \frac{p \cdot i}{(1 + i)^{-n}}$	`100 A = (P * I)/((1 + I)**-N)`

4. Comparing Data Values

a) Relational Operators

Two numbers or strings can be compared to find out their numeric or alphabetical order. This is accomplished with the relational operators: >, <, <=, >=, <>, =.

For example, `100 IF A < B THEN A = A + 1`

or `100 IF N$ < "SMITH" THEN 200`

b) Logical Operators

A relation can be restricted or expanded by using the logical operators AND and OR.

For example, `100 IF A < B AND C < D THEN 150`

or `100 IF N$="SMITH" OR N$="JONES" THEN 300`

See Chapter 6 for a complete explanation of relational and logical operators.

D. STRING OPERATIONS

1. Concatenation

The only arithmetic operator used with strings is the plus sign(+). When two strings are "added" they are brought together, or concatenated, to form a new string:

```
Example: 10 A$ = "JOHN "
         20 B$ = "SMITH"
         30 C$ = A$ + B$
         40 PRINT C$
         50 END
         RUNNH

         JOHN SMITH

         Ready
```

Blanks are only significant if they are embedded inside quotes. If the blank were left out after JOHN in statement 10, the resulting string C$ would have the value of "JOHNSMITH".

2. String Functions

Several predefined functions exist (MID, LEFT and RIGHT) which allow extraction of characters from within an existing string. These functions and other miscellaneous string functions are explained in Chapter 12.

E. SAMPLE PROGRAM

The following program gives examples of elementary arithmetic procedures. All the transactions that would take place (entering commands and instructions, and computer responses) are included. The PRINT instruction will be explained in Chapter 5, but its use in this example should be obvious.

```
NEW TEST1
Ready

100 X = 5
110 Y = 2
120 S = X + Y
130 D = X - Y
140 P = X * Y
150 Q = X / Y
160 A = (X + Y)/2
170 PRINT "THE VALUES ARE";X;"AND";Y
180 PRINT
190 PRINT "THE SUM IS";S
200 PRINT
210 PRINT "THE DIFFERENCE IS";D
220 PRINT
230 PRINT "THE PRODUCT IS";P
240 PRINT
250 PRINT "THE QUOTIENT IS";Q
260 PRINT
270 PRINT "THE AVERAGE IS";A
280 END
RUNNH

THE VALUES ARE 5 AND 2

THE SUM IS 7

THE DIFFERENCE IS 3

THE PRODUCT IS 10

THE QUOTIENT IS 2.5

THE AVERAGE IS 3.5

Ready
```

CHAPTER 4

HOW TO DESIGN A PROGRAM

A. OVERVIEW

There are many ways to design a computer program. One way is for the programmer to design the program while sitting at the computer terminal. If the program is simple, this method can be advantageous. However, if the program is at all complex, this approach can be a waste of time. The programmer cannot be certain that the design is complete or totally correct. Much time is spent testing the program and rewriting sections to make them work properly. This usually takes a great deal of time and can be very frustrating to the programmer.

Another approach is for the programmer to follow a systematic "checklist" for designing and implementing a program. The logic design is done on paper, not on the terminal. This gives the programmer a chance to "walk through" the logic and test it for accuracy and completeness. It also ensures that by the time the programmer gets to the terminal, the program is written, pre-tested, and ready to go. The obvious advantage is guaranteeing the accuracy of the program before it is entered in the computer. The apparent disadvantage is that this structured approach may initially take more time than the previous method. However, in the long run, the experience gained by using this method makes it the best of all the alternatives.

This chapter focuses on this systematic approach. The following section presents a detailed procedure for designing and implementing a computer program. The procedure is summarized in a convenient checklist. Finally, two programming design tools, namely flowcharts and pseudocode, are presented, along with equivalent BASIC statements.

B. PROGRAM DESIGN PROCEDURES

1. Preliminary Steps

a. Define the objectives of the program. Determine what information is to be produced by the program, what input data will be used to generate the information, and how the data will be processed.

   This step is the most important one in this entire procedure. It it is done correctly, it will lead to an accurate solution to the problem.

b. List examples of the problem as it would be solved without a computer. It the computer program is to replace a manual procedure, get a copy of the procedure. This will be used in the design of the program's logic. Also, the examples will serve as test cases that will be used later.

c. Design an output plan for the program. This can be a printed report, a screen format, a data record layout, or any organized list of the output information that is needed.

d. List the input data that will be used to produce the information on the output plan. This may involve individual data items, data records, complete data files, or portions of a data base.

e. Define the processing steps needed to transform the input data into the output information. Use the procedures and examples from above to help with this step.

f. Document all of the above steps for later reference. This step is very important for many reasons. First, it becomes a tool to help with the rest of the steps in this procedure. Also, it becomes a permanent record of the program development process that can be used in the future (for program maintenance and/or modification).

2. Designing the Program Logic

   a. Write a simple outline for the program (using a flowchart or pseudocode). Use the information from the preliminary steps. Remember that every program has input, processing and output steps.

   b. Write a detailed outline of the program (again with a flowchart or pseudocode). This outline should be an expanded version of the previous one. It should list all data input, all information output, all of the processing steps, plus any decisions, transfers, predefined processes, and miscellaneous steps. It is this outline that will be used to write the program.

   c. Walk through the detailed outline using the test case data from preliminary steps. Make sure that all of the processing steps work correctly to produce the right information.

   d. Evaluate the program outline for effectiveness, clarity, and efficiency. Does the outline solve the problem effectively? Is the solution clear and easy to follow? Does it solve the problem efficiently? These criteria make for a high-quality program and should not be ignored.

   e. Document all of the above steps. Again, this important step will help in the rest of the steps and/or any future reference to the program.

3. Writing the Program in BASIC

   a. Define the data variables that will be used in the program. These may come from the input data, the processing steps, the output information, plus any decisions, transfers, or predefined processes. Keep a legend that shows the name and meaning of each variable name.

   b. Write the input section of the program. If the program inputs data from the terminal, be sure to include prompts for the user that explain what data items are to be entered.

   c. Write the processing section. Follow the detailed outline from the previous steps. Be sure to write the BASIC statements in the same order as the logic outline.

   d. Write the output section of the program. Follow the output plan from an earlier step. Include any necessary headings to help identify the information being output. See Chapter 5 for details on output formatting.

e. Write the additional sections of the program as needed. This includes the decisions, transfers, and predefined processes not covered in the three previous steps. Follow the detailed outline exactly in order to complete the program.

f. Walk through the BASIC program using the test case data from the preliminary steps. Evaluate the program using the same criteria that were used for the program logic design. When the program passes this test, it is ready to be entered into the computer.

g. Again, document all of these steps. Initially, the written-out BASIC program will be entered into the computer as is. Any future work on the program will require the variable legend and program development notes.

4. Entering the Program Into the Computer

a. Sign onto the system (see Chapter 1 for details).

b. Clear the work area and assign the program name. This is normally done with NEW command (see Chapter 1).

c. Enter the BASIC program and remarks. Follow the previous recommendations for line numbering increments. Make the program visually clear by indenting "nested" sections and using REM statements to separate program segments.

d. Get a hardcopy listing of the program for later reference.

e. Move a copy of the program from the work area to the disk disk catalog (using the SAVE command).

5. Testing and Debugging the Program

a. Execute the program (using the RUN command).

b. If the program does not run, compare error messages to the hardcopy listing of the program. Check for correct spelling and punctuation of the program statements. Edit the mistakes as necessary and re-execute the program.

c. Use the test case data (from the preliminary steps) to validate the completeness and accuracy of the information produced by the program. Choose additional test data in order to check all logic paths, decisions, and transfers.

d. Document the test results for future reference.

6. Program Performance Standards

   a. Verify that the program meets the original objectives and solves the problem. Check to see how flexible the program is at dealing with unexpected changes in the type and/or volume of data processed.

   b. Make sure that the end-user accepts the program as the solution to a problem.

   c. Guarantee that the program follows the programming standards that are being used.

   d. If the program is part of a system of programs, make certain that it is compatible with the entire system.

   e. Complete all of the documentation for the program, including:
      1. User operating instructions
      2. Program design notes (all of these steps)
      3. Comments within the program (REM or !)
      4. Data structure (variables, record layouts, files)
      5. Test run results
      6. Suggestions for future program modifications

All of these steps are summarized on the following page. This PROGRAM DESIGN & IMPLEMENTATION CHECKLIST should be followed for every programming project done by the reader.

PROGRAM DESIGN & IMPLEMENTATION CHECKLIST

1. Preliminary Steps
   a. Define the objectives of the program.
   b. List examples, procedures, and test cases.
   c. Design output plan.
   d. List the input data.
   e. Define the processing steps.
   f. Document.

2. Designing the Program Logic
   a. Write simple outline.
   b. Write detailed outline.
   c. Walk through detailed outline.
   d. Evaluate for effectiveness, clarity, and efficiency.
   e. Document.

3. Writing the Program in BASIC
   a. Define the variables.
   b. Write the input section.
   c. Write the processing section.
   d. Write the output section.
   e. Write additional sections (decisions, transfers, etc).
   f. Walk through and evaluate.
   g. Document.

4. Entering the Program into the Computer
   a. Sign onto the system.
   b. Clear the work area and assign the program name (NEW).
   c. Enter the BASIC program and REMark statements.
   d. Get hardcopy listing of the program (LIST).
   e. Save the program on the disk catalog (SAVE).

5. Testing and Debugging the Program
   a. Execute (RUN) the program.
   b. Check for correct syntax and edit as necessary.
   c. Use test case data to verify program.
   d. Document.

6. Program Performance Standards
   a. Verify that program meets objectives.
   b. Verify end-user acceptance.
   c. Follow programming standards.
   d. Verify system compatibility.
   e. Complete all documentation.

C. PROGRAM DESIGN TOOLS

This section presents two tools used in program logic design, namely flowcharts and pseudocode. These are valuable techniques used in the development and documentation of computer programs. The following list points out some of the advantages of flowcharts and pseudocode:

1) A graphic or narrative solution to a complex logic problem can be easily understood by programmers and non-programmers alike. Thus flowcharts and pseudocode become important tools of communication between a programmer and an end-user.

2) Logic development is aided by visualizing the solution as a picture or listing the processing steps in outline form. Also, flowcharts and pseudocode are independent of any specific computer language. This means that they can describe the flow of processing events for virtually any data processing system that will be programmed in virtually any computer language.

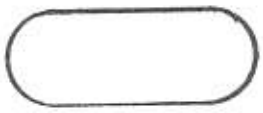
3) Flowcharts and pseudocode can lead directly to the program itself. A common technique is to take each symbol from the flowchart or each step from the pseudocode and turn it into one statement in a particular programming language. This continues until the flowchart/pseudocode is complete and the entire program is written.

4) The program development techniques discussed earlier stressed the importance of documentation. Flowcharts and pseudocode are excellent types of material to be included in any description and documentation of a computer program.

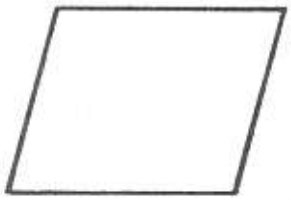
1. Flowchart Symbols

A flowchart is a graphic representation of a procedure. The following symbols are standard and common for data processing flowcharts:

Beginning and/or ending of a procedure:

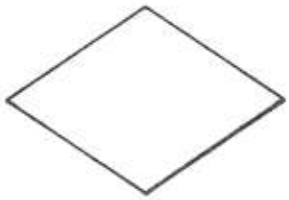


Input and/or output:



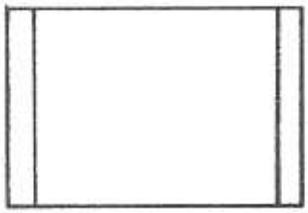
Processing of data:

Decisions (comparisons):



Transfers (branches) and/or sequential processes:

Predefined processes:



2. Pseudocode Vocabulary

Pseudocode is a narrative representation of a procedure. It is written in outline-like form using specific words and structured style. The following vocabulary is standard for data processing pseudocode:

Beginning of a procedure:	BEGIN
Ending of a procedure:	END
Input:	INPUT data item(s)
Output:	OUTPUT data item(s)
Processing of data:	ASSIGN variable = expression
Decisions (comparisons):	IF-THEN-ELSE
Transfers (branches):	GOTO

3. Equivalent BASIC Statements

Flowchart symbols and pseudocode have equivalent elements in the BASIC language. These are shown below:

Beginning of a procedure:	none
Ending of a procedure:	END
Input:	READ, INPUT, INPUT #
Output:	PRINT, PRINT USING, PRINT #
Processing of data:	LET variable = expression
Decisions (comparisons):	IF-THEN-ELSE
Transfers (branches):	GOTO, ON-GOTO
Predefined processes:	FOR-NEXT, GOSUB-RETURN

4. Example: Checking account balance

The following example is presented in order to show how flowcharts and pseudocode can be used to represent the solution to a logic design problem. The problem involves keeping a running balance for a checking account. The account has a beginning balance and is continually adjusted with deposits or checks until the procedure is stopped. The objective is to compute and output the new account balance after each transaction.

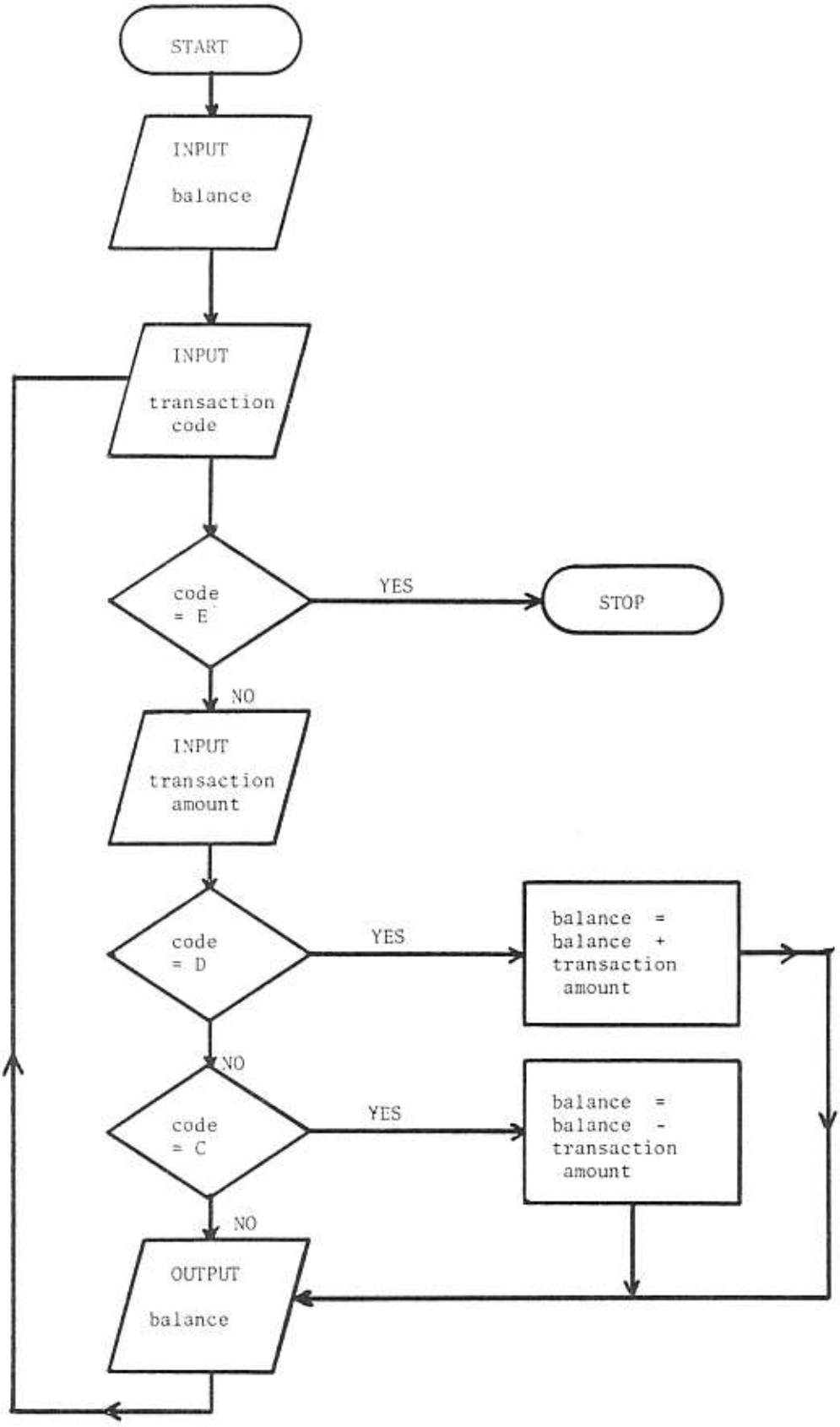
The process begins with the input of the account balance. Next a transaction code is input (D for deposit, C for check, or E to end the procedure). Then a transaction amount is entered. Based on the previous code, the amount is either added to the balance (deposit) or subtracted from the balance (check). The new balance is output and the program returns to the transaction code input step. This continues until the ending code is entered.

The pseudocode for this problem is shown below and the corresponding flowchart appears on the next page. A version of this program written in the BASIC language is presented in Chapter 6.

Checking account balance pseudocode:

```
1.  BEGIN

2.  INPUT balance

3.  INPUT transaction code

4.  IF transaction code = "E"
       THEN END

5.  INPUT transaction amount

6.  IF transaction code = "D"
       THEN ASSIGN balance = balance + transaction amount
            GOTO step 8

7.  IF transaction code ="C"
       THEN ASSIGN balance = balance - transaction amount

8.  OUTPUT balance

9.  GOTO step 3
```

Checking account balance flowchart:



CHAPTER 5

INPUT AND OUTPUT

A. OVERVIEW

So far, this book has concentrated on the essential information needed to design BASIC programs and begin working with a computer. In this chapter, we focus on two of the most important aspects of processing data: input and output. Without these two things, there would be no data to process and no information produced.

Input consists of the data items given to the BASIC program for processing. When the processing is complete, the program outputs information. Input data can get into the computer from a variety of sources. If the amount of data is small, the terminal keyboard can be used to enter it. For moderate amounts of input data, the BASIC program itself can contain the input data (see DATA statements in section C of this chapter). For large quantities of input data, external data files are used (these are discussed in Chapter 10).

Information can be output to a variety of destinations too. The terminal screen is the most common for small amounts of output. A printer attached to the terminal can be used to get hardcopy output of larger amounts of information. Also, external data files can be used as the destination for program output.

This chapter is divided into four sections. First we will discuss two methods of data input: the INPUT statement and the READ and DATA statements. Then we will cover output with a complete description of the PRINT statement (and all of its options). Finally we will cover how to obtain hardcopy output.

B. INPUT STATEMENT

For small amounts of input data, the BASIC language provides the INPUT statement. This is used to input data items from the terminal keyboard at the time a program is executed. This means that the data values can be changed each time the program is used.

1. Single Field

The general form of the INPUT statement for a single data value is

```
line # INPUT variable
```

For example, the BASIC statement

```
10 INPUT A
```

is interpreted to mean "accept a numeric data value from the terminal keyboard and assign it to the variable A."

When an INPUT statement is executed, the computer prompts the user with a question mark, alerting him or her that the program is waiting for input data. After a value is typed, the user must press the RETURN key to signify that the input is complet complete. Another example of the INPUT statement is:

```
20 INPUT X$
```

This means "accept a string data value from the terminal and assign it to the variable X$." Again, when this statement is executed, a question mark prompt will be displayed.

2. Multiple Fields

The general form for INPUTting multiple data fields is:

```
line # INPUT variable,variable, . . ., variable
```

For example, this statement

```
10 INPUT A,B,C
```

is interpreted to mean "accept three numeric data values from the terminal and assign them to variables A, B, and C respectively." When this statement is executed, the computer will display one question mark for each field (if the RETURN key is used after each value is entered) or accept all three values at once, each separated by a comma. So, the options are:

? data value 1 or ? data value 1, data value 2, data value 3
? data value 2
? data value 3

The INPUT statement with multiple data fields can also mix numeric and string input in one statement. For example, the statement

```
40 INPUT A,N$,D,E$
```

means "accept four data values from the terminal: a number, a string, another number, and another string; and assign them to variables A, N$, D, and E$ respectively."

3. INPUT with User Prompt

The examples of INPUT shown thus far have only produced a question mark as the user prompt. The INPUT statement may include a literal prompt to be displayed on the terminal screen prior to the question mark. This greatly assists the user in understanding what the computer is asking for. Without some kind of specific prompt, the user sees only a question mark. How does the user know what the computer wants?

The general form of this type of INPUT statement is:

```
line # INPUT "prompt";variable
```

The following short program shows how this works:

```
10 INPUT "WHAT IS YOUR NAME";N$
20 INPUT "HOW OLD ARE YOU";A
30 END
RUNNH

WHAT IS YOUR NAME? _____________________________
HOW OLD ARE YOU? _____________

Ready
```

Notice that when the program is executed (using the RUNNH command), the prompts are displayed prior to the question marks. In this program, the first data item input is assigned to the variable N$ and the second item input is assigned to the variable A. The user prompts make it very clear what input is expected by the program.

This type of INPUT statement can take on different forms as shown in the following examples.

```
10 INPUT "WHAT IS YOUR NAME AND AGE";N$;A
RUNNH

WHAT IS YOUR NAME AND AGE? _____________,___________

Ready


10 INPUT "WHAT IS YOUR NAME";N$;"YOUR AGE";A
RUNNH

WHAT IS YOUR NAME? _____________________________
YOUR AGE? ___________________

Ready
```

C. READ AND DATA STATEMENTS

For moderate amounts of input data the BASIC language provides the READ and DATA statements. In this case, the READ statement performs the input operation. The DATA statement stores actual input values for the program and is a nonexecutable statement.

The general format of these statements is:

```
line # READ variable, variable, . . ., variable

line # DATA value, value, . . ., value
```

Consider the following example:

```
10 READ A,B,C
20 DATA 2,5,3
```

The data values in line 20 will be assigned to the program variables in the same order that they appear in the READ statement. After the above statements are executed, the variables will have the following values: A=2, B=5 and C=3.

The same thing could have been accomplished with three assignment statements as follows:

```
10 A=2
20 B=5
30 C=3
```

But if the data had to be changed, the executable statements would have to be changed. When READ and DATA statements are used, only the DATA statements are affected when data values must be changed. (Note: For programs requiring large amounts of data, external files are used. In these cases, only the external files change; the file containing the program remains the same.)

READ and DATA do not have to be consecutive statements. It is the order of the variables and the corresponding data values that is important. For example, the following three programs produce the same results:

```
10 READ A,B,C
20 DATA 2,5,3
30 END
```

```
10 READ A,B,C
20 DATA 2
30 DATA 5
40 DATA 3
50 END
```

```
10 READ A,B
20 READ C
30 DATA 2
40 DATA 5,3
50 END
```

When a READ statement is executed, the computer will find the next DATA value that has not already been read.

String data values can be input in the same manner using READ and DATA statements. For example:

```
10 READ A$,B$,C$
20 DATA "JONES","SMITH","BROWN"
```

Notice that the data values in the DATA statement are each enclosed in quotation marks. These indicate string values and are required in the DATA statement. After the above statements are executed, the variables will have the following values: A$="JONES", B$="SMITH", and C$="BROWN"

Numeric and string variables may be mixed in the same READ statement. The user must be careful to match the proper type of value (numeric or string) to its corresponding variable. For example:

```
10 READ N$,X,J$
20 DATA "SMITH",300,"JONES"
```

The user must also be careful to match the number of variables with the number of data values. The following example shows the case where too few items are included in the DATA statement:

```
10 READ A,B,C,D
20 DATA 10,25,30
30 END
RUNNH

Out of data at line 10

Ready
```

The READ statement contains four variables, but there are only three values in the DATA statement. When the program is executed, the error message (Out of data at line 10) is displayed. This message indicates that the computer has run out of data while trying to execute a READ statement (in this case, line 10).

The opposite case, where there are more data values than there are variables does not create a problem. The extra values are simply ignored. For example:

```
10 READ A,B,C
20 DATA 10,15,20,25,30,35,40
```

The first three data values are assigned to the variables and the rest of the values are ignored (until another READ statement).

A final note: DATA statements can appear anywhere in a BASIC program (since they are nonexecutable statements). It is often helpful to group them together near the end of the program so as to clearly distinguish executable instructions from data.

D. PRINT STATEMENT

Programming would be meaningless without a way to output the information created, calculated, and produced within a program. In BASIC, the PRINT statement is used to output information to the terminal screen or a printer. (A variation of the PRINT statement is also used to output data to external files. This is covered in Chapter 10.)

The general form of the PRINT statement is:

```
line # PRINT list of data fields
```

where the list of data fields can include one or multiple data items of the following type: numeric constants, variables, and expressions, string constants, variables, and expressions. These variations are covered in the following sections.

1. Constant values

The PRINT statement can be used to output constant data values, one per line, as follows:

```
line # PRINT numeric constant or "string constant"
```

Notice that the string constant is enclosed in quotation marks. The following program shows how single constant data values can be output:

```
10 PRINT 52.7
20 PRINT "HELLO THERE"
30 END
RUNNH

52.7
HELLO THERE

Ready
```

2. Variables

The PRINT statement can also be used to output variables (again one per line), as follows:

```
line # PRINT variable
```

The form is the same for both numeric and string variables as the following program shows:

```
10 N$="GOLDEN GATE BRIDGE"
20 X=47.32
30 PRINT N$
40 PRINT X
50 END
RUNNH

GOLDEN GATE BRIDGE
47.32

Ready
```

3. Expressions

The PRINT statement is also used to output the result of a calculated expression. The form for this is:

```
line # PRINT expression
```

Expressions can involve variables, constants, and operators. For more information on expressions, review Chapter 3. The following program shows how expressions can be used in PRINT statements:

```
10 PRINT (10*2)/5
20 PRINT "JOHN " + "SMITH"
30 END
RUNNH

4
JOHN SMITH

Ready
```

4. Blank lines

A PRINT statement with no parameters will output a blank line as shown in the following example:

```
10 PRINT "REVENUE"
20 PRINT
30 PRINT "EXPENSES"
40 END
RUNNH

REVENUE

EXPENSES

Ready
```

5. Multiple Fields with Semicolons

The PRINT statement can also be used to output multiple data items on one line, as follows:

```
line # PRINT field;field;. . .; field
```

Note: The semicolons used to punctuate this type of PRINT statement cause the following formatting to occur on the output line:

One blank space is printed before and after numeric fields

No blank spaces are printed around string fields

The following program demonstrates this punctuation:

```
10 PRINT 10;5;27.3;10*2
20 PRINT "JOHN";"SMITH"
30 PRINT "THE ANSWER IS";4**2
40 END
RUNNH

10  5  27.3  20
JOHNSMITH
THE ANSWER IS 16

Ready
```

Notice that line 10 was output with blank spaces around each numeric field, while line 20 was output with no blank space between the two string fields. Line 30 was output with one blank space before the printing of the numeric expression.

6. Multiple Fields with Commas (Print Zones)

A greater formatting capability exists for the PRINT statement when commas are used for punctuation. In this case, data fields are automatically printed in "zones." A print zone is fourteen characters wide as shown in the following diagram:

```
/--------------/--------------/--------------/----- etc. ----/
     zone 1          zone 2          zone 3           etc.

(14 characters) (14 characters)    etc.
```

The general form of this type of PRINT statement is:

```
line # PRINT field,field,. . ., field
```

Note: The commas used to separate the fields cause the values to be displayed automatically in their respective print zones.

Since most terminal screens display a maximum of eighty characters per line, there are five complete print zones on each line. For example:

```
10 PRINT 10,5,27,3,10*2,15
20 PRINT "JOHN","SMITH"
30 PRINT 53.7,"MAIN","STREET","USA",98765
40 END
RUNNH

10            5             27.3          20            15
JOHN          SMITH
53.7          MAIN          STREET        USA           98765

Ready
```

If a particular field is longer than fourteen characters, it will be printed in two zones and the next value will be printed in the following zone.

It is possible to skip over print zones by using leading or imbedded commas in the PRINT statement. For example,

```
10 PRINT ,,"HELLO"
```

will print HELLO in print zone 3, and,

```
20 PRINT 10,,25
```

will print the number 10 in zone 1, skip zone 2, and print the number 25 in zone 3.

7. TAB Function

Another option for print formatting is the TAB function. This is used to specify the exact position on a line where a data field is to be printed. The general form is:

```
line # PRINT TAB(value);field
```

where value = a numeric constant, variable, or expression used to determine a horizontal print position

field = any data item to be printed

For example:

```
10 PRINT TAB(30);"HELLO"
```

causes the computer to print HELLO immediately after horizontal position 30 (i.e., at position 31).

It is possible to use multiple TAB functions on one line as long as the successive TAB values are listed in increasing order. For example:

```
10 PRINT TAB(20);"HELLO";TAB(50);"GOODBYE"
```

This prints HELLO immediately after position 20 and GOODBYE immediately after position 50.

Example:

```
10 PRINT TAB(10);A;TAB(35);B;TAB(60);C
```

This prints the value stored in variable A at position 11, the value for B at position 36, and the value for C at position 61.

The TAB value can also be a calculated expression as shown in the following example:

```
10 T=20
20 PRINT TAB(T*2);"HELLO"
```

This statement performs a TAB function to horizontal position 40 (T*2=40) and then prints HELLO.

8. PRINT USING Statement

The greatest flexibility and power over output formatting comes with the PRINT USING statement. The other methods for output formatting (e.g., semicolons, commas, and the TAB function) are often inadequate to handle the demands of specific output requirements. For these cases, the user can specify the exact output for any given line of information with the PRINT USING statement. The general form is:

```
line # PRINT USING output format string, list of fields
```

where:

output format string = a string constant or variable containing various special symbols (shown below) that are used to describe the desired output format

list of fields = specific data items to be output according to the output format string

The special symbols used in the output format string are described in the table below:

For Numeric Fields

Symbol	Meaning
#	used to represent each digit in a numeric field
*	used to fill in leading zeros in a numeric field with asterisks
.	represents the decimal point position in a numeric field
,	represents the commas(s) to be printed in a numeric field
$	prints a stationary dollar sign preceding a numeric value
$$	prints a dollar sign immediately preceding a numeric value (i.e., a floating dollar sign)
-	used immediately after a numeric field to indicate a negative value (if not used, a minus sign would appear immediately preceding a negative numeric).

For String Fields

Symbol	Meaning
!	used to represent a single-character string field
\ \	used to represent a multiple-character string field. Each \ symbol stands for one character and each blank space between the symbols stands for one character.

Note: If the actual field exceeds the number of spaces alloted to print it, extra characters are truncated (not printed) on the right-hand side of the field.

Each of the following examples gives a sample PRINT USING string format, several data values to be printed with that format, and the results of the formatting operation, namely how the values actually appear on the output line.

PRINT USING symbols	Actual Data Values	Output
###	5	5
	52	52
	527	527
***	5	**5
	52	*52
	527	527
##.#	2	2.0
	52.7	52.7
#,###.##	5	5.00
	5274.23	5,274.23
$##,###.##	5	$ 5.00
	5274.23	$5,274.23
$$#,###.##	5	$5.00
	5274.23	$5,274.23
###	527	527
	-527	-527
###-	527	527
	-527	527-
!	"HELLO"	H
	"B"	B
\ \	"HELLO"	HELLO
	"HI THERE"	HI TH
	"GOODBYE"	GOODB

The following programming examples show various applications of the PRINT USING statement:

Example:
```
10 PRINT USING "###.##",A
```

This means "print the value of the variable A showing three digits, a decimal point, and two more digits."

Example:
```
50 PRINT USING "## \     \ ###.##",N$,T$,R
```

This means "print the values of N, T$, and R using the format specified in the output format string."

Note: It is usually easier to define the output format string with a variable prior to its use (as opposed to the string constants shown in the two previous examples). The PRINT USING statement functions the same, but the program is easier to read.

Example:
```
40 A$ = "## \     \ ###.##"
50 PRINT USING A$,N,T$,R
```

The output format string can also include literal messages consisting of characters without reserved meanings as follows:

```
10 A$ = "THE ANSWER IS ###.##"
20 B = 235.71
30 PRINT USING A$,B
40 END
RUNNH

THE ANSWER IS 235.71

Ready
```

The list of fields to be output with a PRINT using statement can contain any mix of numerics, strings, constants, variables, or expressions as long as the order of the list matches the order of the special symbols contained in the output format string. For example:

```
10 A$ = "\       \ $$#,###.##"
20 X = 2000
30 Y = 1500
40 PRINT USING A$, "REVENUE",X
50 PRINT USING A$, "EXPENSES",Y
60 PRINT USING A$, "PROFIT",X-Y
70 END
RUNNH

REVENUE   $2,000.00
EXPENSES  $1,500.00
PROFIT      $500.00

Ready
```

A final note on PRINT USING: Programs frequently print headings to identify columns of printed data. With formatted output, the easiest way to align the headings and the data is to define the output format immediately after the headings are printed. For example:

```
 10 PRINT "ITEM        COST    MARKUP    PRICE"
 20 A$ =  "\         \ ###.##  ##.##     ####.##"
  .
  .
  .
150 PRINT USING A$,I$,C,M,P
  .
  .
```

E. HARDCOPY OUTPUT

Getting a listing of program statements or output on a piece of paper is called hardcopy output. This topic was discussed in Chapter 1 for both multi- and single-user environments. A multi-user environment with shared output resources (e.g., printers) needs some mechanism such, as a printer switch, to connect an individual terminal with the printer. When the printer is connected, or "engaged" this way, it will simultaneously print anything that is displayed on the terminal screen. This may be a program listing (result of the LIST command) or program output (result of the RUN command).

In a single-user environment, a system command is usually all that is needed to get a program listing. The command on most micro-computers is LIST. For program output, however, a special PRINT statement is included in most microcomputer versions of BASIC to direct program output to the system printer. The standard statement is

```
line # LPRINT list of data fields
```

where "list of data fields" contains all of the data items (i.e., constants, variables, and expressions) to be printed. Note the difference from the PRINT keyword, which directs output to the terminal screen. Quite often, an LPRINT USING statement is also available for special formatted output.

PROGRAM 5-1 CURRENCY CONVERSION

This sample program shows the simple input, processing, and output statements needed to convert U.S. dollars to Japanese yen. The conversion is based on an exchange rate of 230 yen per dollar.

The first step is the printing of a message that identifies the program. Next, the user is prompted for and inputs the number of dollars that are to be converted. Then, the program computes the equivalent number of yen by multiplying the dollar value by 230. Finally, the program outputs the yen amount and the job is complete.

There are two data variables in this program; D is used for the dollar amount and Y is used for yen. The other data in this program consists of the literal messages that are included in the input and output operations and the numeric constant 230 used in the conversion formula.

PROGRAM 5-1

```
100 REM CURRENCY CONVERSION
110 REM
120 REM THIS PROGRAM CONVERTS U.S. DOLLARS TO
130 REM JAPANESE YEN BASED ON THE EXCHANGE RATE
140 REM OF 230 YEN PER DOLLAR.
150 REM
160 REM VARIABLES:     D=DOLLAR AMOUNT
170 REM                Y=YEN AMOUNT
180 REM
190 PRINT "CURRENCY CONVERSION PROGRAM"
200 PRINT
210 INPUT "HOW MANY DOLLARS DO YOU HAVE";D
220 Y = D*230
230 PRINT
240 PRINT "YOU NOW HAVE";Y;"YEN."
999 END
RUNNH

CURRENCY CONVERSION PROGRAM

HOW MANY DOLLARS DO YOU HAVE? 100

YOU NOW HAVE 23000 YEN

Ready
```

PROGRAM 5-2 FORM LETTERS

This next sample program shows a familiar business word and data processing function -- the fill-in-the-blank form letter. In this case, the user inputs various customer data including name, address, city, state, zip code, account number, and balance due. The program proceeds immediately to the output section and prints the form letter with the input data inserted at the appropriate places in the letter.

Of the seven data variables in this program, six are string or alphanumeric (used to store letters and/or numbers). The single numeric variable is used for balance due. The "processing" in this program consists of the concatenation of the string variables and the literal contents of the form letter.

```
100 REM FORM LETTERS                                    PROGRAM 5-2
110 REM
120 REM THIS PROGRAM PRINTS A FORM LETTER TO A CUSTOMER WHOSE
130 REM NAME, ADDRESS, AND CREDIT DATA HAS BEEN INPUT
140 REM
150 REM                    VARIABLES:
160 REM   N$=NAME          C$=CITY      A1$=ACCOUNT NUMBER
170 REM   A$=ADDRESS       S$=STATE     D=BALANCE DUE
180 REM                    Z$=ZIP CODE
200 PRINT "FORM LETTER PROGRAM"
210 PRINT
220 INPUT "WHAT IS THE CUSTOMER'S NAME";N$
230 INPUT "ADDRESS";A$
240 INPUT "CITY";C$
250 INPUT "STATE";S$
260 INPUT "ZIP CODE";Z$
270 INPUT "ACCOUNT NUMBER";A1$
280 INPUT "HOW MUCH IS CURRENTLY DUE";D
290 PRINT
300 PRINT "ATLAS FINANCE COMPANY"
310 PRINT "1984 MAIN STREET"
320 PRINT "SAN FRANCISCO, CA 94110"
330 PRINT
340 PRINT N$
350 PRINT A$
360 PRINT C$;", ";S$;" ";Z$
370 PRINT
380 PRINT "ATTENTION: ";N$
390 PRINT
400 PRINT "OUR RECORDS INDICATE THAT YOUR CHARGE ACCOUNT,"
410 PRINT "NUMBER ";A1$;", HAS A CURRENT BALANCE OF $";D;"."
420 PRINT "PLEASE PAY THIS AT YOUR EARLIEST CONVENIENCE."
430 PRINT
440 PRINT "SINCERELY"
450 PRINT
460 PRINT "CREDIT MANAGER, ATLAS FINANCE CO."
999 END
```

```
RUNNH

FORM LETTER PROGRAM

WHAT IS THE CUSTOMER'S NAME? JOHN DOE
ADDRESS? 100 MARKET STREET
CITY? CENTERVILLE
STATE? CALIFORNIA
ZIP CODE? 94444
ACCOUNT NUMBER? 1590
HOW MUCH IS CURRENTLY DUE? 257.89

ATLAS FINANCE COMPANY
1984 MAIN STREET
SAN FRANCISCO, CA 94110

JOHN DOE
100 MARKET STREET
CENTERVILLE, CALIFORNIA 94444

ATTENTION: JOHN DOE

OUR RECORDS INDICATE THAT YOUR CHARGE ACCOUNT
NUMBER 1590 HAS A CURRENT BALANCE OF $ 257.89.
PLEASE PAY THIS AT YOUR EARLIEST CONVENIENCE

SINCERELY

CREDIT MANAGER, ATLAS FINANCE CO.

Ready
```

PROGRAM 5-3 LOAN PAYMENT

This program computes a loan payment amount based on any principal, interest rate, and term. The user is prompted for and inputs the three necessary data items: the loan principal (the amount borrowed), the interest rate per period (in decimal form), and the number of payments to be made on the loan (the term).

Next, the loan payment is computed using the following formula:

$$\text{Payment} = \text{Principal} \times \frac{\text{Interest Rate}}{1 - (1 + \text{Interest Rate})^{-N}}$$

where N = number of payments

Finally the program outputs the loan payment value.

There are four variables used in this program: P represents the loan principal, I is the interest rate, N stands for the number of payments, and A is the amount of the loan payment.

```
100 REM LOAN PAYMENT                                    PROGRAM 5-3
110 REM
120 REM THIS PROGRAM COMPUTES THE LOAN PAYMENT AMOUNT
130 REM FOR ANY PRINCIPAL, INTEREST RATE, AND TERM.
140 REM
150 REM                    VARIABLES:
160 REM  P=PRINCIPAL AMOUNT    N=NUMBER OF PAYMENTS (TERM)
170 REM  I=INTEREST RATE       A=AMOUNT OF LOAN PAYMENT
180 REM
190 PRINT "LOAN PAYMENT PROGRAM"
200 PRINT
210 INPUT "WHAT IS THE LOAN PRINCIPAL";P
220 INPUT "INTEREST RATE (IN DECIMAL)";I
230 INPUT "HOW MANY LOAN PAYMENTS";N
240 A=P*I/(1-(1+I)^-N)
250 PRINT
260 PRINT "LOAN PAYMENT EQUALS $";A
999 END
RUNNH

LOAN PAYMENT PROGRAM

WHAT IS THE LOAN PRINCIPAL? 1000.00
INTEREST RATE (IN DECIMAL)? .08
HOW MANY LOAN PAYMENTS? 5

LOAN PAYMENT EQUALS $ 250.46

Ready
```

PROGRAM 5-4 BREAK-EVEN ANALYSIS

The "break-even point" is defined as the number of units that must be sold to make revenue equal expenses (producing no profit). Revenue (for a company selling 1 product) is equal to the number of units sold times the selling price per unit. Expenses are made up of two major items: fixed expenses and variable expenses. Fixed expenses (overhead) do not change with changes in production volume. Variable expenses do and are equal to the number of units sold times the unit cost. This sample program computes the break-even point for a single product. The break-even point is determined by the formula:

$$\text{Break-even point} = \frac{\text{Fixed Cost}}{(\text{Unit selling price} - \text{Unit Cost})}$$

The selling price, unit cost and fixed cost values are read from a DATA statement. The break-even point is computed, along with the revenue and total expenses at break-even.

```
100 REM BREAK-EVEN ANALYSIS                              PROGRAM 5-4
110 REM COMPUTE BREAK-EVEN FROM UNIT PRICE, UNIT AND FIXED COSTS
130 REM                    VARIABLES:
140 REM  P=UNIT SELLING PRICE     B=BREAK-EVEN POINT
150 REM  C=UNIT COST              R=REVENUE AT BREAK-EVEN PT.
160 REM  F=FIXED COST             E=EXPENSES AT BREAK-EVEN PT.
170 READ P,C,F            ! READ UNIT PRICE, UNIT COST, FIXED COST
180 B=F/(P-C)             ! COMPUTE BREAK-EVEN POINT
190 R=P*B                 ! COMPUTE REVENUE AT BREAK-EVEN POINT
200 E=(C*B)+F             ! COMPUTE EXPENSES AT BREAK-EVEN POINT
210 PRINT "BREAK-EVEN ANALYSIS PROGRAM"
220 PRINT
230 PRINT "UNIT PRICE";P
240 PRINT "UNIT COST";C
250 PRINT "FIXED COST";F
260 PRINT
270 PRINT "BREAK-EVEN POINT";B
280 PRINT "REVENUE AT BREAK-EVEN POINT";R
290 PRINT "EXPENSE AT BREAK-EVEN POINT";E
300 DATA 5.00, 3.50, 1500
939 END
RUNNH

BREAK-EVEN ANALYSIS PROGRAM

UNIT PRICE 5.00
UNIT COST 3.50
FIXED COST 1500

BREAK-EVEN POINT 1000
REVENUE AT BREAK-EVEN POINT 5000
EXPENSE AT BREAK-EVEN POINT 5000

Ready
```

PROGRAM 5-5 INCOME STATEMENT

This program prints a simple income statement for a company selling one product. The input data (input using the READ and DATA statements) consists of unit selling price, unit cost, fixed cost, and number of units sold. The program computes the income statement values using the following formulas:

```
Revenue       = Unit selling price  X  Number of units sold
Cost of sales = Unit selling price  X  Unit cost
Gross profit  = Revenue - Cost of sales
Net profit    = Gross profit - Fixed cost
```

Finally, the program outputs the income statement. Notice that the output appears in 2 columns as a result of using the comma to print data in print zones 1 and 2. The numeric data in print zone 2 is left-justified; that is, it does not line up the values on the decimal point.

```
100 REM INCOME STATEMENT                                    PROGRAM 5-5
110 REM
120 REM THIS PROGRAM PRINTS AN INCOME STATEMENT THAT SHOWS REVENUE,
130 REM COST OF SALES, GROSS PROFIT, FIXED COST AND NET PROFIT
140 REM                      VARIABLES
150 REM  P=UNIT SELLING PRICE      R=REVENUE
160 REM  C=UNIT COST               S=COST OF SALES
170 REM  F=FIXED COST              G=GROSS PROFIT
180 REM  N=NUMBER OF UNITS SOLD    X=NET PROFIT
300 READ P,C,F,N              ! READ DATA ITEMS
310 R=P*N                     ! COMPUTE REVENUE
320 S=C*N                     ! COMPUTE COST OF SALES
330 G=R-S                     ! COMPUTE GROSS PROFIT
340 X=G-F                     ! COMPUTE NET PROFIT
400 PRINT "INCOME STATEMENT PROGRAM
410 PRINT
420 PRINT "REVENUE";R
430 PRINT "COST OF SALES";S
440 PRINT "GROSS PROFIT";G
450 PRINT "FIXED COST";F
460 PRINT "NET PROFIT";X
470 DATA 5.00, 3.50, 1500, 2500
999 END
RUNNH

INCOME STATEMENT PROGRAM

REVENUE       12500
COST OF SALES 8750
GROSS PROFIT  3750
FIXED COST    1500
NET PROFIT    2250

Ready
```

PROGRAM 5-6 INCOME STATEMENT WITH FORMATTED OUTPUT

This program is the same as 5-5, except that the output appears in a specific format designed by the programmer (and not the usual print zones). The data variable A$ is used to represent output format. The first two characters in A$, the back slashes, show the starting and ending print positions of the string values to be printed (e.g., REVENUE, COST OF SALES). The next collection of characters shows the print position of the numeric values. The $$ stands for a "floating" dollar sign, the #'s represent digits of the numeric field, and the comma and the decimal point show where those will be printed in the output value.

The PRINT USING statement is used to output the formatted lines. The statement format, PRINT USING A$ followed by the two data items to be printed on the line, is used for all five lines of the income statement.

```
100 REM INCOME STATEMENT WITH FORMATTED OUTPUT       PROGRAM 5-6
110 REM
120 REM A SIMPLE INCOME STATEMENT THAT SHOWS REVENUE, COST OF
130 REM SALES, GROSS PROFIT, FIXED COST AND NET PROFIT
140 REM                    VARIABLES
150 REM  P=UNIT SELLING PRICE     R=REVENUE
160 REM  C=UNIT COST              S=COST OF SALES
170 REM  F=FIXED COST             G=GROSS PROFIT
180 REM  N=NUMBER OF UNITS SOLD   X=NET PROFIT
190 REM  A$=OUTPUT FORMAT STRING
300 READ P,C,F,N                  ! READ DATA ITEMS
310 R=P*N                         ! COMPUTE REVENUE
320 S=C*N                         ! COMPUTE COST OF SALES
330 G=R-S                         ! COMPUTE GROSS PROFIT
340 X=G-F                         ! COMPUTE NET PROFIT
350 A$="\             \   $$###,###.##"
400 PRINT "INCOME STATEMENT PROGRAM"
410 PRINT
420 PRINT USING A$, "REVENUE",R
430 PRINT USING A$, "COST OF SALES",S
440 PRINT USING A$, "GROSS PROFIT",G
450 PRINT USING A$, "FIXED COST",F
460 PRINT USING A$, "NET PROFIT",X
470 DATA 5.00, 3.50, 1500, 2500
999 END
RUNNH

INCOME STATEMENT PROGRAM

REVENUE          $12,500.00
COST OF SALES     $8,750.00
GROSS PROFIT      $3,750.00
FIXED COST        $1,500.00
NET PROFIT        $2,250.00

Ready
```

CHAPTER 6

DECISIONS AND TRANSFERS

A. DECISIONS

1. Comparing Data Values

A comparison between data items is accomplished with the IF statement. The items compared can be numeric quantities or strings. In the case of strings, the comparison will determine the alphabetical order of the strings. A comparison will show how one item relates to another (one is equal to the other, greater than the other, etc). To accomplish this task in a program statement, two new sets of symbols, called relational and logical operators, must be introduced:

a) Relational Operators

Symbol	Sample Relation	Meaning
=	X = Y	X equal to Y
<	X < Y	X less than Y
<=	X <= Y	X less than or equal to Y
>	X > Y	X greater than Y
>=	X >= Y	X greater than or equal to Y
<>	X <> Y	X not equal to Y

b) Logical Operators

Several relations can be included in one program statement by using the logical operators, AND and OR. This allows the programmer to test more than one characteristic of a data value or to test several sets of data values at the same time.

Symbol	Sample relation	Meaning
AND	A < B AND X > Y	A less than B and X greater than Y
OR	A < B OR X > Y	A less than B or X greater than Y

2. IF Statement

The general form of the IF statement is:

```
line # IF   relation   THEN   instruction   ELSE   instruction
                              or transfer          or transfer
```

The keywords IF and THEN are always part of the statement. The "relation" and "instruction or transfer" will change according to the criteria used for the comparison. The last part of the statement "ELSE instruction or transfer" is optional.

For example, take two numeric quantities, A and B. A typical comparison and resulting action might be stated as "If the value (quantity) of A is less than the value of B, then print the value of A". This could be accomplished in BASIC by the statement:

```
100 IF A < B THEN PRINT A
```

The relation is A < B and the resulting action (instruction) is PRINT A. Notice, however, that the action is taken only if the relation is true. If it is false, no action is taken; in this case the computer program would simply move to the next statement. The following program will put the IF statement listed above in the context of a decision making process:

```
10 READ A,B
20 DATA 2,3
30 IF A < B THEN PRINT "A IS LESS THAN B"
40 END
RUNNH

A IS LESS THAN B

Ready
```

By changing the numbers in statement 20, any two values can be compared by the same program.

Using the ELSE option the user can specify, in the same statement, what is to be done if the relation is false. The following program will compare the two variables A and B. If the value of A is less than the value of B, then the message SMALLER will be printed, otherwise the message EQUAL OR BIGGER will be printed.

```
10 READ A,B
20 DATA 4,5
30 IF A < B THEN PRINT "SMALLER" ELSE PRINT "EQUAL OR BIGGER"
40 END
RUNNH

SMALLER

Ready
```

In terms of the stated relation (A < B), the program logic is:

If the relation is true, SMALLER is printed.
If the relation is false, EQUAL OR BIGGER is printed.

The IF statement can be extended so that several levels of comparison can be performed. The program above cannot state specifically if A is equal to or bigger than B. But by extending the decision process in statement 30 the specific relation can be determined:

```
30 IF A < B THEN PRINT "SMALLER"
              ELSE IF A > B THEN PRINT "BIGGER"
              ELSE PRINT "EQUAL"
```

The logical operators can restrict or expand the criteria used to compare data values. The AND operator allows the programmer to restrict (become more selective about) a comparison. The general form for its usage is:

```
relation1 AND relation2
```

The result of the operation will be considered true only if both relation1 and relation2 are true.

For example, let us say that a list of personnel records is to be searched for all employees who are salaried and have an income over $300 per week. Each record contains three items: name, pay type (SALARIED or HOURLY), and regular weekly income. If these three values are read into the variables N$, T$ and I respectively, the following statement can be used to test each record:

```
100 IF T$ = "SALARIED" AND I > 300 THEN PRINT N$
```

This can be read as "If the pay type is salaried and the income is greater than 300 then print the name."

The OR operator allows an expansion of criteria involved in a comparison. The general form for its usage is

```
relation1 OR relation2
```

The result of the operation will be considered true if either relation1 or relation2 is true. The result is also true if both relation1 and relation2 are true.

Taking the same personnel records, if it were desired to print the names of all employees who are either salaried or have a regular income over $300 per week (which now may include some hourly employees) the following statement could be used:

```
100 IF T$ = "SALARIED" OR I > 300 THEN PRINT N$
```

B. TRANSFERS (BRANCHING)

1. Unconditional Branching

It is possible to bypass the normal order of statement execution (sequentially by line number) with the GOTO statement. This statement provides a way of transferring to another portion of the program. This action is also referred to as branching. The general form of the GOTO statement is

line # GOTO line number

where "line number" is any BASIC statement line number in the program. For example:

```
200 GOTO 500
```

Upon execution of this statement the program will "branch to" statement 500, that is, all statements between 200 and 500 will be skipped. In short, statement 500 is the next statement executed after statement 200. This type of branching is called unconditional, since there are no decisions made within the statement to determine if the branch will occur or not. It simply branches "without condition." The following example is an extension of the previous program. With the GOTO statement, a list of number pairs can be compared:

```
10 READ A,B
20 DATA 2,3,5,5,7,6
30 IF A < B THEN PRINT A;"LESS THAN";B
40 IF A = B THEN PRINT A;"EQUALS";B
50 IF A > B THEN PRINT A;"GREATER THAN";B
60 GOTO 10
70 END
RUNNH

2 LESS THAN 3
5 EQUALS 5
7 GREATER THAN 6
Out of data at line 10

Ready
```

When two values are compared, one must obviously be less than, equal to or greater than the other. One of the three IF statemnts (30, 40 or 50) will have a relation that is true and the other two will be false. The relation that is true will cause the message that follows to be printed. Statement 60 then directs the computer to branch to statement 10 and read two more values. (Remember, the READ statement always takes the next available DATA values that have not been read). The actual sequence of operations performed by the program (using the DATA values in statement 20) is:

Statement executed	Resulting Action
10	Values read into variables (A=2, B=3).
30	Relation true. Message "2 LESS THAN 3" printed.
40	Relation false. Program moves to next statement.
50	Relation false. Program moves to next statement.
60	Program control transferred to statement 10.
10	Values read into variables (A=5, B=5).
30	Relation false. Program moves to next statement.
40	Relation true. Message "5 EQUALS 5" printed.
50	Relation false. Program moves to next statement.
60	Program control transferred to statement 10.
10	Values read into variables (A=7, B=6).
30	Relation false. Program moves to next statement.
40	Relation false. Program moves to next statement.
50	Relation true. Message "7 GREATER THAN 6" printed.
60	Program control transferred to statement 10.
10	Program attempts to read another set of values into the variables A and B. Since there is no more data in the DATA statement, an error message is printed and program execution is halted.

2. Conditional Branching

Note that the program above ended with an error message. In this case no harm was done since all of the data had already been processed. But ending the program this way is referred to as a "hard exit", that is, forcing the program to stop by creating an error condition. A "soft exit", or normal ending can be accomplished by using another IF statement. An extra set of data values can be appended to the end of the DATA statement. The only purpose of these values is to signal the program that there is no more data:

```
10 READ A,B
20 DATA 2,3,5,5,7,6,0,0
25 IF A = 0 THEN 70
30 IF A < B THEN PRINT A;"LESS THAN";B
40 IF A = B THEN PRINT A;"EQUALS";B
50 IF A > B THEN PRINT A;"GREATER THAN";B
60 GOTO 10
70 END
RUNNH

2 LESS THAN 3
5 EQUALS 5
7 GREATER THAN 6

Ready
```

The relation in statement 25 (A = 0) was false for the first three pairs of values, but on the fourth pair (A=0, B=0) this relation was true and control was transferred to the END statement, accomplishing a "soft" ending to the program. This type of transfer is called a conditional branch, since it occurs only "on the condition that" a previous relation is true. The general form of the IF statement for a condition transfer can be written as:

```
line # IF relation THEN line number
```

where "line number" is the BASIC statement number to which the program will branch when the relation is true. Note that in the conditional branch the keyword GOTO does not appear. The "GOTO" function is implied when the line number appears in the statement after the keyword THEN.

a) Sentinel Values

The last two data values in the program above (0,0) were not processed in the same way as the other sets of values. Their purpose, again, was to provide a normal exit for the program. Data values used in this way are called sentinel values. The choice of what to use as a sentinel value is up to the programmer, but of course one would want to choose a number that is never a legitimate data value. Strings can also be used as sentinels.

Note that at the end of the DATA statement the sentinel value (A=0) was followed by another value (also zero). This was only a dummy value for B since the READ statement must be satisfied (enough data must be provided for all the variables listed) or else the program will not continue. Either A or B could have been used to test a sentinel value, but data must be provided for both.

b) Multi-branch operations

A multi-way branch can be set up in one statement so that a program can GOTO one of a list of specified lines based on the result of an expression. This is accomplished with the ON GOTO statement, a conditional branch statement which has the general form:

```
line # ON variable GOTO list of line numbers
```

It is assumed that the value of the "variable" has already been determined in a previous calculation. The variable must be an integer. If it is not, the ON GOTO statement will truncate it before making the branch decision. Based on the value of the variable, the program branches to one of the "list of line numbers." The line number transferred to is the one in the same sequential position as the value of the variable.

For example:

```
10 ON K GOTO 100,150,200
```

This is interpreted as:
If the value of K is 1, then branch to line 100
If the value of K is 2, then branch to line 150
If the value of K is 3, then branch to line 200

If the value of the variable is <= 0 or greater than the number of line numbers in the list, an error will result.

The program on elementary arithmetic operations, listed at the end of Chapter 3, will be re-written to demonstrate the ON-GOTO statement. The user will be prompted to input two data values which will be assigned to the variables X and Y. The user is then asked to enter an "operation key", a value which is assigned to the variable K. Based on the value of this number the following action will be taken:

If the value of K is	the program will calculate
1	the sum of X and Y
2	the difference X minus Y
3	the product of X times Y
4	the quotient X divided by Y
5	the average of X and Y

```
100 REM ARITHMETIC OPERATION CHOICE
110 INPUT "ENTER TWO NUMBERS";X;Y
120 INPUT "ENTER OPERATION KEY";K
130 PRINT
140 ON K GOTO 150,170,190,210,230
150 PRINT "THE SUM IS";X+Y
160 GOTO 999
170 PRINT "THE DIFFERENCE IS";X-Y
185 GOTO 999
190 PRINT "THE PRODUCT IS";X*Y
200 GOTO 999
210 PRINT "THE QUOTIENT IS";X/Y
220 GOTO 999
230 PRINT "THE AVERAGE IS";(X+Y)/2
999 END
RUNNH

ENTER TWO NUMBERS? 5,2
ENTER OPERATION KEY? 3

THE PRODUCT IS 10

Ready
```

PROGRAM 6-1 TOTALING AND AVERAGING

Computing the total and average of a list of numbers is a procedure that is common to many programs. When someone takes a total manually they actually add two numbers at a time and accumulate the total after each addition. For example, to total the numbers 1 + 8 + 4, we would say "1 plus eight equal 9, plus 4 equals 13." Automating this process in a computer program results in the same type of operation. To obtain an average, one needs a count of how many numbers were in the list (say N). The average, or arithmetic mean, is then simply the total divided by N.

The following program demonstrates the technique of totaling and averaging. The data consists of a number of records, each containing a name of a college and an enrollment figure. The purpose of the program is to find the average number of students in large colleges, where "large" is defined to be 10000 or more students. Thus a selection process is needed. Each record must be read and the enrollment figure compared to 10000. If the enrollment is 10000 or more then the figure is added to an accumulating total and a count is kept of how many such colleges were encountered. The particular college and enrollment are then printed. A string value (a college named "ZZZ") is used as the sentinel record. When this name is encountered it will be assumed that there is no more data and that the average can be computed and printed.

```
100 REM TOTALING AND AVERAGING                                  PROGRAM 6-1
110 N = 0                          ! INITIALIZE COUNTER
120 T = 0                          ! INITIALIZE TOTAL
130 READ C$,S                      ! READ COLLEGE NAME, ENROLLMENT
140 IF C$ = "ZZZ" THEN 200
150 IF S <= 10000 THEN 130         ! SKIP IF ENROLLMENT < 10000
160 N = N + 1                      ! COUNT NO. OF COLLEGES
170 T = T + S                      ! ACCUMULATE TOTAL ENROLLMENT
180 PRINT C$,S                     ! PRINT COLLEGE NAME, ENROLLMENT
190 GOTO 130                       ! REPEAT PROCEDURE
200 A = T / N                      ! AVERAGE ENROLLMENT
205 PRINT
210 PRINT "THE AVERAGE ENROLLMENT OF";N;"COLLEGES =";A
220 DATA "UCLA",25000,"CENTRAL",9800,"STANFORD",24000
230 DATA "HARVARD",23000,"LOCALVILLE",8500,"NOTRE DAME",22000
240 DATA "ZZZ",0
999 END
RUNNH

UCLA          25000
STANFORD      24000
HARVARD       23000
NOTRE DAME    22000

THE AVERAGE ENROLLMENT OF 4 COLLEGES = 23500

Ready
```

PROGRAM 6-2 TAX RANGES

Quite often a multi-branch decision is based on a number which is too big to use conveniently as an ON GOTO variable. In this situation a scaling factor can be used to convert the number to a small integer (1, 2, 3 etc). For example, let's say that the amount of tax taken from an employee's gross pay depends on the following table:

Salary Range	Tax Applied
0- 99	0%
100-199	10%
200-299	25%
300-399	40%
400-499	50%

A number of IF statements could be used to compare a given salary to each salary range and find the proper tax. But considering that the ON GOTO variable is truncated to an integer, we can divide the salary by 100 and use the resulting number (from 1-5) as the decision variable. The following program demonstrates this method:

```
100 REM TAX RANGES                                          PROGRAM 6-2
110 REM
120 REM VARIABLES: S=SALARY      T=TAX
130 PRINT,"TAX RANGES":PRINT
200 T=0
210 READ S
215 IF S < 0 THEN 999
220 IF S >= 500 THEN PRINT "SALARY OUT OF RANGE":GOTO 210
230 X = S/100
240 IF X < 0 THEN 300
250 ON X GOTO 260,270,280,290
260 T = S * .10 : GOTO 300
270 T = S * .25 : GOTO 300
280 T = S * .40 : GOTO 300
290 T = S * .50 : GOTO 300
300 PRINT "SALARY =";S,"TAX =";T
310 GOTO 200
400 DATA 150,75,430,350,-1
999 END
RUNNH

          TAX RANGES

SALARY = 150   TAX = 15
SALARY = 75    TAX = 0
SALARY = 430   TAX = 215
SALARY = 350   TAX = 140

Ready
```

PROGRAM 6-3 CHECKING ACCOUNT

This program shows how decisions and transfers can be combined to perform a useful function -- the balancing of a checkbook. The program starts by prompting the user for the beginning balance of the checking account. This is input into the variable B. Next, the program outputs a message telling the user to enter a transaction code. The three codes are "C" for check, "D" for deposit, and "E" to end the program. The code is input into the variable T$.

The program checks to see if T$ equals the code "E"; if so, the program transfers to the ending lines. If a code other than "E" is entered, the program continues its normal sequence. The user is next prompted for a transaction amount, which is input into the variable A.

Now the program uses decisions to adjust the account balance. If the transaction code is a "C" (meaning check), the transaction amount is subtracted from the balance to yield the new balance. If the transaction code is a "D" (meaning deposit), the new balance equals the old balance plus the transaction amount. Any other transaction code will not affect the balance.

The new balance is output and the program transfers back to the step prompting for a transaction code. This repetition continues until the user enters an "E" to end the program. At that point, the program outputs the final account balance and ends.

```
100 REM CHECKING ACCOUNT                                PROGRAM 6-3
110 REM
120 REM THIS PROGRAM KEEPS TRACK OF THE RUNNING
130 REM BALANCE OF A CHECKING ACCOUNT.
140 REM                  VARIABLES:
150 REM     B=CHECKING ACCOUNT BALANCE
160 REM     T$=TRANSACTION CODE (C=CHECK, D=DEPOSIT, E=END)
170 REM     A=TRANSACTION AMOUNT
200 PRINT "CHECKING ACCOUNT PROGRAM"
210 PRINT
220 INPUT "WHAT IS THE STARTING BALANCE";B
230 PRINT
240 PRINT "ENTER TRANSACTION CODE"
250 INPUT "C=CHECK, D=DEPOSIT, E=END";T$
260 IF T$="E" THEN 320
270 INPUT "ENTER TRANSACTION AMOUNT";A
280 IF T$="C" THEN B=B-A : GOTO 300
290 IF T$="D" THEN B=B+A
300 PRINT "THE NEW BALANCE IS";B
310 GOTO 230
320 PRINT
330 PRINT "THE FINAL BALANCE IS";B
999 END
```

```
RUNNH

CHECKING ACCOUNT PROGRAM

WHAT IS THE STARTING BALANCE? 1000

ENTER TRANSACTION CODE
C=CHECK, D=DEPOSIT, E=END? C
ENTER TRANSACTION AMOUNT? 100
THE NEW BALANCE IS 900

ENTER TRANSACTION CODE
C=CHECK, D=DEPOSIT, E=END? D
ENTER TRANSACTION AMOUNT? 200
THE NEW BALANCE IS 1100

ENTER TRANSACTION CODE
C=CHECK, D=DEPOSIT, E=END? C
ENTER TRANSACTION AMOUNT? 300
THE NEW BALANCE IS 800

ENTER TRANSACTION CODE
C=CHECK, D=DEPOSIT, E=END? E

THE FINAL BALANCE IS 800

Ready
```

PROGRAM 6-4 TAX LIABILITY

This program combines input, processing, decisions, and output to demonstrate a simplified tax liability computation. The program computes total income, dependent allowance, and adjusted income. Then it computes an adjustment to income as the greater of the two values deductible expenses or 10% of adjusted income. The tax liability is equal to 20% of the second adjusted income value.

The following four values are input to start the program: salary (into variable S), miscellaneous income (M), number of dependents (N), and deductible expenses (D).

The processing section begins with the computing of total income (variable T) equal to salary plus miscellaneous income. The first adjustment to income (A1) represents the dependent allowance and is equal to the number of dependents times 1000. Adjusted income 1 (I1) equals total income minus the first adjustment. The variable P equals 10% of the adjusted income figure. This 10% is compared to the deductible expenses and the greater of the two is assigned to the variable A2 representing adjustment 2. Adjustment 2 is subtracted from adjusted income 1 yielding adjusted income 2 (variable I2). Finally, the tax liability (L) is computed as 20% of adjusted income 2.

The program ends by outputting the following information: total income, dependent allowance, adjustment for deductible expenses of 10%, adjusted income, and tax liability.

```
100 REM TAX LIABLILITY                                    PROGRAM 6-4
110 REM
120 REM THIS PROGRAM COMPUTES TAX LIABILITY BASED ON SALARY, MISC.
130 REM INCOME, NUMBER OF DEPENDENTS, AND DEDUCTIBLE EXPENSES.
140 REM THE TOTAL INCOME IS ADJUSTED FIRST FOR THE DEPENDENT
150 REM ALLOWANCE ($1000 PER). THEN 10% OF THE ADJUSTED INCOME IS
160 REM COMPARED TO DEDUCTIBLE EXPENSES AND THE GREATER BECOMES
170 REM THE SECOND ADJUSTMENT TO INCOME. FINALLY, THE NEWLY
180 REM ADJUSTED INCOME IS USED AS THE BASIS OF A 20% TAX
240 REM                  VARIABLES
250 REM   S=SALARY                          A1=ADJUSTMENT 1
260 REM   M=MISC. INCOME                    I1=ADJUSTED INCOME 1
270 REM   N=NO. OF DEDUCTIONS               P=10%OF ADJUSTED INCOME 1
280 REM   D=DEDUCTIBLE EXPENSES             A2=ADJUSTMENT 2
290 REM   T=TOTAL INCOME                    I2=ADJUSTED INCOME 2
300 PRINT "TAX LIABILITY PROGRAM" : PRINT
310 INPUT "WHAT IS YOUR SALARY";S
320 INPUT "MISC. INCOME";M
330 INPUT "NUMBER OF DEPENDENTS";N
340 INPUT "DEDUCTIBLE EXPENSES";D
400 T=S+M                    ! COMPUTE TOTAL INCOME
410 A1=N*1000                ! COMPUTE DEPENDENT ALLOWANCE
420 I1=T-A1                  ! COMPUTE ADJUSTED INCOME 1
430 P=.1*I1                  ! COMPUTE 10% OF ADJUSTED INCOME 1
440 REM DETERMINE ADJUSTMENT 2
450 IF P>D THEN A2=P ELSE A2=D
460 I2=I1-A2             ! COMPUTE ADJUSTED INCOME 2
470 L=.2*I2              ! COMPUTE TAX LIABILITY
500 PRINT
510 PRINT "TOTAL INCOME";T
520 PRINT "DEPENDENT ALLOWANCE";A1
530 PRINT "ADJUSTED INCOME";I1
540 PRINT "ADJUSTMENT FOR DEDUCTIBLE EXPENSES OR 10%";A2
550 PRINT "ADJUSTED INCOME";I2
560 PRINT "TAX LIABILITY";L
999 END
RUNNH

TAX LIABILITY PROGRAM

WHAT IS YOUR SALARY? 25000
MISC. INCOME? 1000
NUMBER OF DEPENDENTS? 2
DEDUCTIBLE EXPENSES? 10000

TOTAL INCOME 26000
DEPENDENT ALLOWANCE 2000
ADJUSTED INCOME 24000
ADJUSTMENT FOR DEDUCTIBLE EXPENSES OR 10% 10000
ADJUSTED INCOME 14000
TAX LIABILITY 2800

Ready
```

PROGRAM 6-5 PAYROLL

This program computes and prints basic payroll information including straight pay, overtime pay (if it applies) and gross pay. The computations are based on the employee's pay type (hourly or salaried) and the number of hours worked during a given week.

The program starts by printing a heading above all the columns of information and by defining a variable (B$) as the output format to be used to print the information.

The main processing begins by reading, from DATA statements, pay type (T$; valid values H=hourly, S=salaried), hours worked (H), and rate of pay (R). The program then checks for sentinel value (when N$ equals "LAST RECORD") signalling the end of the data. Next, the pay type values are validated (as "H" or "S") and the the program transfers to further processing sections. If both pay type validations fail, a message is printed telling the user that there is an invalid pay type code for a particular employee.

For salaried employees:
```
Straight pay = 40  X  rate of pay
Overtime     = 0
Gross pay    = straight pay
```

For hourly employees (up to 40 hours per week):
```
Straight pay = number of hours   X  rate of pay
Overtime     = 0
Gross pay    = straight pay
```

For hourly employees (over 40 hours per wee):
```
Straight pay = 40  X  rate of pay
Overtime     = (number of hours - 40) X rate of pay X 1.5
Gross pay    = straight pay + overtime
```

The formatted output is accomplished with a PRINT USING B$ statement. Output includes employee name, pay type, number of hours worked, rate of pay, straight pay, overtime, and gross pay. The program then transfers back to read the next data record. This repetition continues until all of the data is used.

```
100 REM PAYROLL                                          PROGRAM 6-5
110 REM
120 REM PAYROLL AMOUNT IS COMPUTED, INCLUDING STRAIGHT, OVERTIME,
130 REM AND GROSS PAY. SALARIED EMPLOYEES DO NOT QUALIFY FOR OVER-
140 REM TIME PAY. HOURLY EMPLOYEES WORKING MORE THAN 40 HOURS ARE
150 REM PAID 1.5 THEIR REGULAR PAY RATE FOR HOURS BEYOND 40
160 REM
170 REM                    VARIABLES
180 REM   N$=NAME              S=STRAIGHT PAY
190 REM   T$=PAY TYPE          X=OVERTIME PAY
200 REM   H=HOURS WORKED       G=GROSS PAY
210 REM   R=RATE OF PAY        B$=OUTPUT FORMAT STRING
```

(continued)

```
220 PRINT "                    PAYROLL PROGRAM"
230 PRINT
240 PRINT "EMPLOYEE PAY  HOURS PAY    STRAIGHT  OVERTIME   GROSS"
250 PRINT "NAME     TYPE       RATE     PAY       PAY        PAY"
260 PRINT
270 B$ =  "\       \ !    ##  ##.##    $###.##   $###.##  $###.##"
300 READ N$,T$,H,R                     ! READ DATA ITEMS
310 IF N$="LAST RECORD" THEN 999
320 IF T$="H" THEN 500 ! BRANCH TO HOURLY ROUTINE
330 IF T$="S" THEN 400 ! BRANCH TO SALARIED ROUTINE
340 PRINT "INVALID PAY TYPE CODE (";T$;") FOR ";N$
350 GOTO 300                           ! READ NEXT RECORD
400 S=40*R                             ! SALARIED: STRAIGHT PAY
410 X=0                                !   NO OVERTIME
420 G=S                                !   COMPUTE GROSS PAY
430 GOTO 630
500 IF H>40 THEN 600                   ! BRANCH TO OVERTIME ROUTINE
510 S=H*R                              ! HOURLY(40): STRAIGHT PAY
520 X=0                                !   NO OVERTIME
530 G=S
540 GOTO 630
600 S=40*R                             ! HOURLY(>40): STRAIGHT PAY
610 X=(H-40)*R*1.5                     !   COMPUTE OVERTIME PAY
620 G=S+X                              !   COMPUTE GROSS PAY
630 PRINT USING B$,N$,T$,H,R,S,X,G
640 GOTO 300
700 DATA "SMITH","H",35,6.25
710 DATA "JONES","H",44,7.00
720 DATA "JOHNSON","S",40,5.95
730 DATA "MILLER","H",45,11.50
740 DATA "ANDERSON","S",41,3.50
750 DATA "O'MALLEY","H",38,7.50
760 DATA "WEST","S",35,4.85
770 DATA "LAST RECORD","0",0,0
999 END
RUNNH
```

PAYROLL PROGRAM

EMPLOYEE NAME	PAY TYPE	HOURS	PAY RATE	STRAIGHT PAY	OVERTIME PAY	GROSS PAY
SMITH	H	35	6.25	$218.75	$ 0.00	$218.75
JONES	H	44	7.00	$280.00	$ 42.00	$322.00
JOHNSON	S	40	5.95	$238.00	$ 0.00	$238.00
MILLER	H	45	11.50	$460.00	$ 86.25	$546.25
ANDERSON	S	41	3.50	$140.00	$ 0.00	$140.00
O'MALLEY	H	38	7.50	$285.00	$ 0.00	$285.00
WEST	S	35	4.85	$194.00	$ 0.00	$194.00

Ready

CHAPTER 7

LOOPING

A. OVERVIEW

1. Establishing a Loop

Many programming problems involve a set of instructions which must be repeated several times. This type of procedure is commonly called a loop. Each pass through the loop affect one or more of the variables in the loop. A good example is a simple totaling procedure. The following program will add any five numbers that are put in the DATA list:

```
100 N = 0                          ! INITIALIZE COUNTER
110 T = 0                          ! INITIALIZE
120 N = N + 1                      ! INCREMENT COUNTER
130 IF N > 5 THEN 170              ! EXIT AFTER 5TH NUMBER
140 READ X                         ! READ NEXT VALUE
150 T = T + X                      ! ACCUMULATE TOTAL
160 GOTO 120                       ! REPEAT LOOP
170 PRINT "THE TOTAL =";T
180 DATA 12,5,2,14,3
190 END
RUNNH

THE TOTAL = 36

Ready
```

The variable N as used in statement 120 is called a counter. It has the purpose of keeping track of how many times the loop has been repeated. The loop can be considered to be between statements 120 and 160. These statements are repeated until N has exceeded the value of 5 (statement 130). Note that only 5 numbers are read. When N reaches a value of 6 the relation in statement 130 will be true and program control will be transferred to statement 170, where a total will be printed.

To accomplish the loop procedure, several things are needed:

- a variable to be used as a counter (N)
- the counter is initialized to some starting value (stmt 100)
- the counter is incremented each time through the loop (stmt 120)
- a check to determine if more repetitions are needed (stmt 130)
- an instruction to return to the start of the loop (stmt 160)

In short, three essential statement types are present: an assignment (stmt 120) a decision (stmt 130) and a transfer (stmt 160).

2. General Format of the FOR-NEXT Process

Looping procedures are so common in programming that a special pair of statements is included in the BASIC language to make it easier to write a loop. The FOR-NEXT process has the general form:

```
line #  FOR  Index = Start Value TO End Value STEP Increment

(BASIC statements)

line # NEXT Index
```

The keywords of this process have the following meaning:

FOR	Mandatory word. It indicates the start of the loop.
Index	A numeric variable which will be incremented each time through the loop
=	Mandatory character.
Start Value	The starting value of the Index. This can be a constant or variable. If it is the latter its value must be established in a previous statement.
End Value	The upper limit of the loop. This can be a constant or variable. When the Index exceeds this value, the loop will not repeat again.
STEP Increment	This part of the statement is optional. It is needed only when the Index is to be incremented with a value other than 1.
NEXT Index	A statement that indicates the end of the set of statements that are to be repeated. The Index variable must be the same as the one in the FOR statement.

One "pass" through the loop includes the execution of the BASIC statements between the FOR statement and the NEXT statement. Within these statements the ordinary order of execution applies, along with any transfers (GOTOs) that may apply. Each time the loop is repeated the Index is incremented. If the STEP option is omitted, the increment value will be considered 1. When the Index exceeds the upper limit, the program will continue with the statement following the NEXT Index statement.

EXAMPLE. The program for totaling five numbers, given above, will be rewritten using a FOR-NEXT loop.

```
100 T = 0
110 FOR I = 1 TO 5
120 READ X
130 T = T + X
140 NEXT I
150 PRINT "THE TOTAL =";T
160 DATA 12,5,2,14,3
170 END
RUNNH

THE TOTAL = 36

Ready
```

In this case the three functions of assignment, decision and transfer needed to repeat the loop are all done automatically. In statement 110, the Index variable I is given a starting value of 1 and an upper limit of 5. Since the STEP option is omitted, the increment is 1. This means that the statements between FOR and NEXT (in this case statements 120 and 130) will be repeated 5 times, since the variable I will be incremented that many times. Note the Index in the NEXT statement (I) must be the same as that in the FOR statement. Thus five numbers are read (stmt 120) and their total is accumulated (stmt 130). When the loop repetitions are complete, program control is transferred to the statement after NEXT I, and the total is printed. Unlike the previous program, the programmer does not have to worry about incrementing the counter and checking if any more repetitions are needed.

EXAMPLE. The program can be made more versatile by making the end value of the loop a variable whose value is read as a data item:

```
100 READ N
110 T = 0
120 FOR I = 1 TO N
130 READ X
140 T = T + X
150 NEXT I
160 PRINT "THE TOTAL=";T
170 DATA 5
180 DATA 12,5,2,14,3
190 END
RUNNH

THE TOTAL = 36

Ready
```

Now by simply changing the DATA statements the program will total any set of numbers. No executable statements have to be modified.

B. USING THE LOOP INDEX

1. The Index as a Value

In the above FOR-NEXT example, the Index had only one purpose, to count the number of times through the loop. The variable I did not appear inside the loop. Since the Index is a legitimate variable its value can be used inside the loop like any other variable. Note that its value can be used but not changed inside the loop. That is to say, the Index variable cannot appear on the left side of an equal sign. Such an assignment statement would change its value. The FOR statement is the only place that the Index variable is changed. The next two examples will put the Index value to use inside the loop.

EXAMPLE. A program that finds the sum of all the integers between 1 and 10:

```
100 S = 0
110 FOR N = 1 TO 10
120 S = S + N
130 NEXT N
140 PRINT "THE SUM = ";S
150 END
RUNNH

THE SUM = 55

Ready
```

The accumulating total procedure is the same, but here the data values used are being produced by the changing value of the Index as it repeats the loop ten times. Notice there is no DATA statement, since the data is generated internally by program instructions.

EXAMPLE. Using the STEP option, the sum of all the even numbers between 2 and 100 can be found.

```
100 S = 0
110 FOR N = 2 TO 100 STEP 2
120 S = S + N
130 NEXT N
140 PRINT "THE SUM = ";S
150 END
RUNNH

THE SUM = 2550

Ready
```

The loop will be repeated 50 times. The first time the Index N will equal 2, the second time N will equal 4, etc. Statement 120 will accumulate the sum of all the even integers.

2. Fractional and Negative Indexes

The Index can be any real number and thus can be a decimal or negative value. The next two examples will demonstrate these uses.

EXAMPLE. This program will convert a series of numbers which represent length in inches to their equivalent value in centimeters (1 in = 2.54 cm). The lengths are given in one-half inch increments from 1 to 3 inches.

```
100 FOR I = 1 TO 3 STEP .5
110 C = I * 2.54
120 PRINT I;"INCHES=";C;"CENTIMETERS"
130 NEXT I
RUNNH

1 INCHES = 2.54 CENTIMETERS
1.5 INCHES = 3.81 CENTIMETERS
2 INCHES = 5.08 CENTIMETERS
2.5 INCHES = 6.35 CENTIMETERS
3 INCHES = 7.62 CENTIMETERS

Ready
```

EXAMPLE. A negative increment is valid, although not used very often. The program above which added all the even numbers between 2 and 100 can be written as:

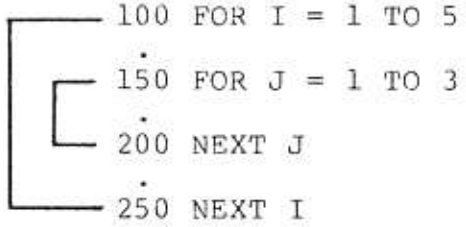
```
100 S = 0
110 FOR N = 100 TO 2 STEP -2
120 S = S + N
130 NEXT N
140 PRINT "THE SUM = ";S
150 END
RUNNH

THE SUM = 2550

Ready
```

Normally a loop will be repeated until the Index reaches the indicated upper limit. It is possible to exit from a loop before this time by simply transferring outside the range of the loop. The program however cannot jump back into the loop and continue. For example, the following program will normally read and total 10 numbers. If one of the numbers is zero, however, it will print out the total up to that point and continue on with the next instruction after the loop.

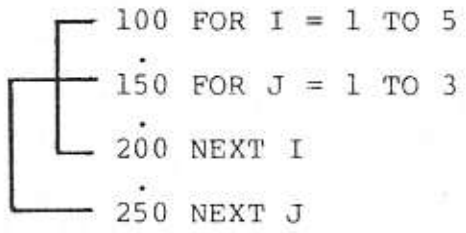
```
100 S = 0
110 FOR I = 1 TO 10
120 READ X
130 S = S + X
140 IF X <> 0 THEN 170
150 PRINT "THE SUM OF";I;"NUMBERS =";S
160 GOTO 200
170 NEXT I
180 DATA 23,6,14,3,2,8,0,34,15,4
190 PRINT "THE SUM =";S
200 END
RUNNH

THE SUM OF 6 NUMBERS IS 56

Ready
```

C. NESTED LOOPS

The repetitive procedure which forms a loop can have another loop contained within its statements. When one loop is inside another the loops are said to be nested. One must be careful to set up the loops properly, however. Take the following program segment:



The loop with index J is completely contained within the loop with index I. This means that the inner loop must have both its FOR and NEXT statements within the outer loop's FOR and NEXT range. Note that lines have been drawn by the side of the program connecting the respective FOR and NEXT statements. If the loops are nested properly, these lines will not cross. Also, a different index variable must be used for each loop. The program segment below shows incorrectly nested loops:



This will produce unpredictable results because statement 200 will instruct the computer to repeat the outer loop (I) before the inner loop is complete.

When loops are nested the index of the outer loop will be incremented and succeeding statements will be executed in order until the inner loop is reached. The inner loop then "takes precedence", that is, the computer will complete the inner loop before it returns to the outer loop. While it is performing the inner loop, the computer stores the value of the outer loop index and must retrieve that same value if the loop is to be performed properly. Thus it is important not to change the values of the indexes anywhere accept in the FOR statement.

EXAMPLE. Nested loops.

```
100 PRINT "  LOOP INDEX VALUES"
110 PRINT "OUTER LOOP  INNER LOOP"
120 FOR I = 1 TO 2
130 FOR J = 1 TO 3
140 PRINT I,J
150 NEXT J
160 NEXT I
170 END
RUNNH

  LOOP INDEX VALUES
OUTER LOOP   INNER LOOP
1              1
1              2
1              3
2              1
2              2
2              3

Ready
```

The outer loop is between FOR I and NEXT I (statements 120-160) and the inner loop is between FOR J and NEXT J (statements 130-150). The first time through the outer loop the index I is set to 1. At statement 130 the inner loop is started and the index J is set to 1. Statements 130-150 are repeated three times, with I remaining at 1 and the value of J changing from 1 to 3. The outer loop then continues at statement 160 which transfers control back to statement 120. The index I is set equal to 2 and the process begins again. When the outer loop is complete the program continues with the next statement after NEXT I, which happens to be the END statement in this case. Any number of statements could have appeared between the FOR I and FOR J statements as well as between the NEXT J and NEXT I, as long as the loops were nested properly.

PROGRAM 7-1 BRANCH OFFICE SUBTOTALS

A simple nested loop structure is used to accumulate Accounts/ Receivable (A/R) figures for two branch offices of a retail goods company. The outer loop, between statements 230-350 is repeated for each branch office. The inner loop, between statements 280-320 adds up the A/R figures for three product categories. Each product and A/R figure is printed as well as the A/R subtotal for each company branch and a grand total for the company.

```
100 REM BRANCH OFFICE SUBTOTALS                          PROGRAM 7-1
110 REM B$=BRANCH OFFICE NAME   A= A/R FOR EACH LINE OF GOODS
120 REM N$=LINE OF GOODS DESCR  T= BRANCH SUBTOTAL  G=COMPANY TOTL
200 PRINT "BRANCH OFFICE SUBTOTALS PROGRAM"
210 PRINT
220 G = 0
230 FOR B = 1 TO 2          ! FOR EACH BRANCH
240 T = 0                   ! INITIALIZE TOTAL
250 READ B$                 ! READ BRANCH NAME
260 PRINT
270 PRINT "BRANCH OFFICE: ";B$
280 FOR L = 1 TO 3          ! FOR EACH LINE OF GOODS
290 READ N$,A               ! READ DESCRIPTION AND A/R
300 PRINT N$,A
310 T = T + A               ! ACCUMULATE TOTAL
320 NEXT L
330 PRINT "TOTAL A/R FOR BRANCH = ";T
340 G = G + T
350 NEXT B
360 PRINT
370 PRINT "GRAND TOTAL FOR ACCOUNTS RECEIVABLE =";G
500 DATA "NEW YORK","CLOTHING",400,"HOUSEHOLD",300,"GARDEN",150
510 DATA "CHICAGO" ,"CLOTHING",500,"HOUSEHOLD",400,"GARDEN",100
999 END
RUNNH
```

```
BRANCH OFFICE SUBTOTALS PROGRAM

BRANCH OFFICE: NEW YORK
CLOTHING       400
HOUSEHOLD      300
GARDEN         150
TOTAL A/R FOR BRANCH = 850

BRANCH OFFICE: CHICAGO
CLOTHING       500
HOUSEHOLD      400
GARDEN         100
TOTAL A/R FOR BRANCH = 1000

GRAND TOTAL ACCOUNTS RECEIVABLE = 1850

Ready
```

PROGRAM 7-2 AMORTIZATION SCHEDULE

An amortization schedule shows the full history of a loan repayment. For each payment made on the loan, the schedule prints the payment number, the payment amount, the amount of interest paid, the amount of principal paid, and the remaining balance. This repetitive operation is a perfect application for the FOR-NEXT statements, where the loop parameters determine the number of lines printed in the amortization schedule.

This program starts by printing column headings and by defining an output format (F$) for the schedule. The loan parameters are constants for the sake of simplicity (B=balance, N=number of payments, A=payment amount, R=interest rate). An expanded version of this program would input the parameters and calculate the payment amount (see example 5-3) before printing the amortization schedule.

The main processing section of this program is the FOR-NEXT loop. The variable X is used as the loop index, with valid values 1 through N. Within the loop, four things occur. First, the interest paid (I) is computed. Next, the principal paid (P) is calculated. Third, the remaining balance (B) is adjusted by the amount of principal paid. Finally, the information is output with the PRINT USING F$ statement. These four things happen repeatedly until the loop concludes and the loan is completely repaid.

```
100 REM AMORTIZATION SCHEDULE                                PROGRAM 7-2
110 REM
120 REM THIS PROGRAM PRINTS AN AMORTIZATION SCHEDULE FOR A LOAN.
140 REM THE SCHEDULE SHOWS PAYMENT NUMBER, PAYMENT AMOUNT,
150 REM INTEREST PAID, PRINCIPAL PAID, AND REMAINING BALANCE
160 REM                      VARIABLES
170 REM B=LOAN BALANCE          X=LOOP INDEX     R=INTEREST RATE
180 REM N=NUMBER OF PAYMENTS    I=INTEREST PAID  A=PAYMENT AMOUNT
190 REM P=PRINCIPAL PAID        F$=OUTPUT FORMAT STRING
200 REM
220 PRINT ,"AMORTIZATION SCHEDULE PROGRAM"
230 PRINT "PAYMENT  PAYMENT  INTEREST  PRINCIPAL  REMAINING"
240 PRINT "NUMBER   AMOUNT     PAID      PAID      BALANCE"
250 F$=    " ###     #,###.## #,###.##  #,###.##   #,###.##"
300 B=1000                  ! BALANCE
310 N=10                    ! NUMBER OF PAYMENTS
320 A=149.03                ! PAYMENT AMOUNT
330 R=.08                   ! INTEREST RATE
340 FOR X=1 TO N
350   I=B*R                 ! COMPUTE INTEREST PAID
360   P=A-I                 ! COMPUTE PRINCIPAL PAID
370   B=B-P                 ! COMPUTE NEW BALANCE
380   PRINT USING F$,X,A,I,P,B
390 NEXT X
999 END
```

```
RUNNH

          AMORTIZATION SCHEDULE PROGRAM

PAYMENT   PAYMENT   INTEREST   PRINCIPAL   REMAINING
NUMBER    AMOUNT      PAID       PAID       BALANCE

   1        149.03     80.00      69.03      930.97
   2        149.03     74.48      74.55      856.42
   3        149.03     68.51      80.52      775.90
   4        149.03     62.07      86.96      688.94
   5        149.03     55.12      93.91      595.03
   6        149.03     47.60     101.43      493.60
   7        149.03     39.49     109.54      384.06
   8        149.03     30.72     118.31      265.75
   9        149.03     21.26     127.77      137.98
  10        149.03     11.04     137.99       -0.01

Ready
```

CHAPTER 8

LISTS AND TABLES

A. OVERVIEW

The programs that have been presented so far have used relatively few variable names. However, most business applications involve a great number of data values, each of which must be stored in a separate variable. The method of naming variables that was introduced in Chapter 3 becomes impractical in these situations. For example, if we wished to read five numeric values into separate variables we could write:

```
10 READ A,B,C,D,E
```

However, if there were one hundred data values to process, we would need one hundred variables! There are enough variables names available (A1, A2...B1, etc) but manipulating all these variables separately becomes unmanageable. There is no question that a separate variable name is needed for each value, but some method must be used to avoid writing these long lists of variables one by one.

The solution comes in the fact that whenever there is a great quantity of data, it can usually be thought of as a list of items that have some common characteristic, for example a list of part numbers, employees, automobile licence plates, etc. In these cases we are not so much concerned with giving a unique context to each entry in the list as we are with processing the list as a whole. Quite often each entry in a list has several characteristics that are related in some way, yet maintain an overall context with the other entries. In this case we call the list a table. For example, a personal income tax table may have a different row for each salary range, and a different column for each exemption category. Each data value in the table is unique and would need a separate variable name, but somehow must be connected in a convenient way to a specific salary and exemption. Data that can be put into a list or table is represented in BASIC by subscripted variables.

B. SUBSCRIPTED VARIABLES

The computerized list can be explained by borrowing a term from mathematics, the set. A set is a group of objects that share some common characteristic. A set is represented in a general way by the notation

$$X = (x_1 \quad x_2 \quad x_3 \quad . \quad . \quad . \quad x_n)$$

where X is the name of the set and x_1 x_2 x_3 ,etc are members or elements of the set. The number attached to each element is called the subscript. For example x_{27} is named "x sub 27". The term x_n indicates that there are n elements in the set.

This type of notation cannot be entered directly on the keyboard since there are no small size numbers that can be used for the subscripts. This problem is overcome by allowing the subscript to be placed inside parentheses, beside the variable name:

X(1) X(2) X(3) . . . .etc

These are distinct variables, just like A, B, C, etc., but the subscript notation now allows a great number of variable names while retaining a common context. The general term for a computerized list is an array. In the X array above X(1) could be the first part number in an inventory list, X(2) the second part number, and so on. Any traditional variable can be made into a subscripted variable, for example: X(9) D(3) G(2) etc.

The computerized table is constructed by adding a second dimension to the variable subscript. In this case one subscript is used to indicate what row the array element belongs to and a second subscript to indicate the column. For example an element from the tax table that was mentioned might be associated with the subscripted variable X(2,3). This would indicate the tax amount for someone who was in the second salary range (row 2) and had 3 exemptions (column 3).

C. DIMENSION SPECIFICATION

The BASIC language allows subscripted variables to be put to the same uses as regular variables, but a special statement must appear in a program before any subscripted variables appear. This is the DIM statement. The general format is:

line # DIM list of arrays

The "list of arrays" is a listing of each array name used in the program. A number is placed in parentheses beside the array name which specifies the maximum number of elements in the array. For example:

```
100 DIM X(5),B(3)
```

Here the term X(5) directs the computer to create five separate variables: X(1), X(2), X(3), X(4), X(5) and the term B(3) will create three more variables: B(1), B(2), B(3). Note that the meaning of the subscript in the DIM statement is quite different than when individual elements are specified in the remainder of the program.

A short program will demonstrate the use of subscripts and the DIM statement. The program below creates a five-element array (X) then reads a number into each one of the elements. Statement 130 displays the values that have been read:

```
100 DIM X(5)
110 READ X(1),X(2),X(3),X(4),X(5)
120 DATA 7,2,6,8,4
130 PRINT X(1);X(2);X(3);X(4);X(5)
140 END
RUNNH

7  2  6  8  4

Ready
```

To demonstrate a two-dimension subscript, a table (matrix) of numbers is manually written below:

		Column 1	Column 2
Row	1	62	53
	2	71	24
	3	81	39

The following program will create a table array and reserve memory storage locations for six variables: three rows and two columns. An individual value will then be read into each variable so that its subscript reflects the table above.

```
100 DIM X(3,2)
110 READ X(1,1),X(1,2)
120 READ X(2,1),X(2,2)
130 READ X(3,1),X(3,2)
140 DATA 62,53,71,24,81,39
150 PRINT X(1,1);X(1,2)
160 PRINT X(2,1);X(2,2)
170 PRINT X(3,1);X(3,2)
180 END
RUNNH

62  53
71  24
81  39

Ready
```

D. VARIABLE SUBSCRIPTS

Providing enough variables for many data values is no longer difficult when subscripted variables are used. But one major problem remains. How do we avoid writing out all the individual variables? Referring back to the first example in the chapter, if the five numbers were put into an array, we could wirte

```
100 READ X(1),X(2),X(3),X(4),X(5)
```

But if there were one hundred numbers, would it be practical to list one hundred variables in this manner? Certainly not.

The solution comes from the most powerful feature of subscripted variables - the fact that the subscript can be a variable itself. X(J) can refer to the Jth element in the array. Of course the variable J must have a value beforehand. The problem of writing many variables can now be eliminated by combining the variable subscript feature and the FOR-NEXT loop. The following programs are put side by side for comparison. They both read five values into array elements:

```
100 DIM X(5)
110 READ X(1),X(2),X(3),X(4),X(5)
120 DATA 5,7,6,2,3
130 END
```

```
100 DIM X(5)
110 FOR I = 1 TO 5
120 READ X(I)
130 NEXT I
150 DATA 5,7,6,2,3
140 END
```

These programs accomplish <u>exactly</u> the same thing, so what is the difference? The benefit of variable subscripts becomes evident if we had to re-write both programs to read one hundred variables. In the program on the left, statement 110 would have to be changed so that one hundred array elements were listed, whereas in the program on the right, statement 110 would only have to be changed to 110 FOR I = 1 TO 100.

But what about the data values? If one hundred values were to be read, then the DATA statements must be expanded to include the hundred numbers. There is no way we can get around this situation, but we have conveniently eliminated the need to write long list of variable names. The topic of <u>source data automation</u> deals with methods being developed in business and industry to aid the human being in writing down individual data values. This involves the use of specialized equipment which can sense the occurance of certain events and automatically record all relevent data. Examples of such devices are supermarket bar-code readers and automatic teller machines.

PROGRAM 8-1 BRANCH OFFICE SUBTOTALS USING ARRAYS

In Program 7-1 (accounts/receivable) the same variables were used for each branch office, thus there was no way of processing the data more than once. If another report were required, say totals by product type, the data would have to be read again. Using arrays eliminates the need to read data twice since each value is stored in a separate variable. The following program performs the same function as Program 7-1 except that array variables are used.

```
100 REM BRANCH OFFICE SUBTOTALS USING ARRAYS          PROGRAM 8-1
120 REM B$(I)=BRANCH OFFICE NAME     N$(I,J)=LINE OF GOODS DESCR
140 REM A(I,J)=A/R AMOUNT   T(B)=BRANCH SUBTOTAL  G=COMPANY TOTAL
200 PRINT "BRANCH OFFICE SUBTOTALS PROGRAM"
210 PRINT
220 FOR B = 1 TO 2              ! FOR EACH BRANCH
230 READ B$(B)                  ! BRANCH NAME
240   FOR L = 1 TO 3              ! FOR EACH PRODUCT LINE
250   READ N$(B,L),A(B,L)         ! PRODUCT TYPE, A/R AMOUNT
260   NEXT L
270 NEXT B
300 G = 0                       ! ACCUMULATE TOTALS
310 FOR B = 1 TO 2              ! FOR EACH BRANCH
320 PRINT "BRANCH OFFICE: ";B$(B)
330   FOR L = 1 TO 3              ! FOR EACH PRODUCT LINE
335   PRINT N$(B,L),A(B,L)        ! PRINT DETAIL
340   T(B)=T(B)+A(B,L)            ! ACCUMULATE BRANCH A/R
350   NEXT L
360 PRINT "TOTAL A/R FOR BRANCH =";T(B)
370 G = G + T(B)                ! ACCUMULATE COMPANY TOTAL
380 NEXT B
390 PRINT
400 PRINT "GRAND TOTAL FOR ACCOUNTS RECEIVABLE =";G
500 DATA "NEW YORK","CLOTHING",400,"HOUSEHOLD",300,"GARDEN",150
510 DATA "CHICAGO" ,"CLOTHING",500,"HOUSEHOLD",400,"GARDEN",100
999 END
RUNNH

BRANCH OFFICE SUBTOTALS PROGRAM

BRANCH OFFICE: NEW YORK
CLOTHING       400
HOUSEHOLD      300
GARDEN         150
TOTAL A/R FOR BRANCH = 850

BRANCH OFFICE: CHICAGO
CLOTHING       500
HOUSEHOLD      400
GARDEN         100
TOTAL A/R FOR BRANCH = 1000

GRAND TOTAL ACCOUNTS RECEIVABLE = 1850

Ready
```

PROGRAM 8-2 AVERAGE AND STANDARD DEVIATION

A common programming application is the calculation of statistics. This program computes the average, variance and standard deviation of a list of numbers, in this case student test scores. An array variable (S) is used for the test scores. First, the variable is "dimensioned." Then, separate values are input to each element of the array (with READ and DATA statements). The average value is equal to the total of all test scores divided by the number of scores. The variable T is used to total the scores and the variable A represents the average score.

The variance is a measurement of the dispersion of the test scores around the average. It is equal to the "average" of the square of the difference between the computed average score and each individual score. Mathematically, it is represented as:

$$\text{Variance} = \frac{(\text{testscore} - \text{average})^2}{\text{number of scores}}$$

The standard deviation is equal to the square root of the variance. The square root of a value is the same as the value raised to the one-half power. The latter method is used in this program. The program concludes by printing the average, variance, and standard deviation of the 20 test scores.

```
100 REM AVERAGE AND STANDARD DEVIATION                PROGRAM 8-2
110 REM
120 REM THIS PROGRAM COMPUTES THE AVERAGE, VARIANCE, AND STANDARD
130 REM DEVIATION FOR A LIST OF NUMBERS, IN THIS CASE TEST SCORES.
145 REM
150 REM   VARIABLES
160 REM  S=ARRAY OF TEST SCORES        A=AVERAGE TEST SCORE
170 REM  N=NUMBER OF SCORES            X=TOTAL OF VARIANCES
180 REM  I=LOOP INDEX                  V=VARIANCE
190 REM  T=TOTAL OF TEST SCORES        D=STANDARD DEVIATION
200 REM
210 DIM S(50)             ! DIMENSION TEST SCORE ARRAY
220 PRINT "AVERAGE AND STANDARD DEVIATION PROGRAM"
230 PRINT
240 PRINT "TEST SCORES:"
250 PRINT
260 READ N                ! READ NUMBER OF SCORES
270 FOR I=1 TO N
280   READ S(I)           ! READ SCORES
290   PRINT S(I)
300 NEXT I
```

(continued)

```
310 T=0  ! INITIALIZE TOTAL
320 FOR I=1 TO N
330   T=T+S(I)            ! TOTAL ALL SCORES
340 NEXT I
350 A=T/N                 ! COMPUTE AVERAGE SCORE
360 X=0                   ! INITIALIZE VARIANCE TOTAL
370 FOR I=1 TO N
380   X=X+(S(I)-A)^2      ! TOTAL SQUARES OF THE VARIANCES
390 NEXT I
400 V=X/N                 ! COMPUTE VARIANCE
410 D=V^.5                ! COMPUTE STANDARD DEVIATION
420 PRINT
430 PRINT "AVERAGE TEST SCORE";A
440 PRINT "VARIANCE OF SCORES";V
450 PRINT "STANDARD DEVIATION";D
460 DATA 20
470 DATA 86,90,76,82,88,68,99,84,86,66
480 DATA 35,68,80,74,72,71,48,82,79,57
999 END
RUNNH

AVERAGE AND STANDARD DEVIATION PROGRAM

TEST SCORES:
 86
 90
 76
 82
 88
 68
 99
 84
 86
 66
 35
 68
 80
 74
 72
 71
 48
 82
 79
 57

AVERAGE TEST SCORE 74.55
VARIANCE OF SCORES 214.248
STANDARD DEVIATION 14.6406

Ready
```

PROGRAM 8-3 EXPENSE REPORT

This program computes and prints the daily, category and grand totals for a weekly expense report matrix. The two-dimensional matrix (variable E) consists of seven columns by five rows of data. The seven columns represent days of the week. The five rows represent expense categories (e.g. food, lodging, transportation).

A one-dimensional array (C), containing seven elements, is used for the column (daily) totals. A five-element, one-dimensional array (R) is used for the row (category) totals. A simple variable (G) is used for the grand total. FOR-NEXT loops are used to initialize all of the totals. A nested loop is used to read the expense matrix and simple loops are used to read the days of the week array (D$) and the expense category array (C$). A two-level nested loop is used to compute the column, row and grand totals.

The first part of the output section prints the days of the week and column totals arrays. The next part prints the expense category and row total arrays. The final output is the grand total of the expense matrix.

```
100 REM EXPENSE REPORT                                    PROGRAM 8-3
110 REM
120 REM THIS PROGRAM COMPUTES DAILY AND CATEGORY TOTALS FOR AN
130 REM EXPENSE MATRIX. A GRAND TOTAL IS ALSO COMPUTED AND PRINTED
140 REM   VARIABLES
150 REM  E=EXPENSE MATRIX 7X5          D$=DAYS OF THE WEEK ARRAY
160 REM  C=COLUMN TOTAL ARRAY          C$=EXPENSE CATEGORY ARRAY
170 REM  R=ROW TOTAL ARRAY             I=LOOP INDEX
180 PRINT "EXPENSE REPORT PROGRAM"
190 DIM E(7,5), R(5), D$(7), C$(5)
200 FOR I=1 TO 7          ! INITIALIZE ALL TOTALS
210   C(I)=0              ! COLUMN TOTALS
220 NEXT I
230 FOR I=1 TO 5
240   R(I)=0              ! ROW TOTALS
250 NEXT I
260 G=0  ! GRAND TOTAL
300 FOR J=1 TO 5          ! ** READ DATA ITEMS **
310   FOR I=1 TO 7
320     READ E(I,J)       ! READ EXPENSE MATRIX
330   NEXT I
340 NEXT J
350 FOR I=1 TO 7
360   READ D$(I)          ! READ DAYS OF THE WEEK ARRAY
370 NEXT I
380 FOR I=1 TO 5
390   READ C$(I)          ! READ EXPENSE CATEGORY ARRAY
400 NEXT I
```

(continued)

```
500 FOR J=1 TO 5            ! ** COMPUTE TOTALS **
510   FOR I=1 TO 7
520     R(J)=R(J)+E(I,J) ! COMPUTE ROW TOTALS
530     C(I)=C(I)+E(I,J) ! COMPUTE COLUMN TOTALS
540     G=G+E(I,J)        ! COMPUTE  GRAND TOTAL
550   NEXT I
560 NEXT J
610 PRINT "DAY","TOTAL"                  ! ** OUTPUT RESULTS **
620 FOR I=1 TO 7
630   PRINT D$(I),C(I)                   ! PRINT DAY AND TOTAL EXP
640 NEXT I
650 PRINT
660 PRINT "ITEM","TOTAL"
670 FOR I=1 TO 5
680   PRINT C$(I),R(I)                   ! PRINT CTGRY AND TOTAL EXP
690 NEXT I
700 PRINT
710 PRINT "GRAND TOTAL",G
800 DATA 5,2,4,5,3,6,5
810 DATA 8,6,5,4,3,4,5
820 DATA 1,5,4,5,6,7,0
830 DATA 6,4,6,7,8,5,4
840 DATA 7,6,5,4,3,2,1
850 DATA "MONDAY","TUESDAY","WEDNESDAY","THURSDAY","FRIDAY"
860 DATA "SATURDAY","SUNDAY"
870 DATA "FOOD","LODGING","PHONE","AIR FAIRE","CAR RENTAL"
999 END
RUNNH

EXPENSE REPORT PROGRAM

DAY             TOTAL

MONDAY           27
TUESDAY          23
WEDNESDAY        24
THURSDAY         25
FRIDAY           23
SATURDAY         24
SUNDAY           15

ITEM            TOTAL

FOOD             30
LODGING          35
PHONE            28
AIR FARE         40
CAR RENTAL       28

GRAND TOTAL     161

Ready
```

PROGRAM 8-4 MONTHLY PROFIT REPORT

This program uses a number of array variables to produce a monthly revenue, expense, profit and average daily profit report. All of the arrays contain twelve elements representing twelve months of the year. The four input arrays, M$ for month names, D for number of days per month, R for monthly revenue, and E for monthly expenses are read from DATA statements using a FOR-NEXT loop.

The processing section starts by calculating the profit array P (revenue minus expenses) and the average daily profit array A (profit divided by number of days in the month).

Next, the program computes yearly totals for revenue (variable X), expenses (Y) and profit (Z).

The output section first prints column headings in the five standard print zones. Then, the following arrays are printed: month name, revenue, expenses, profit and average daily profit. Finally, the program outputs the yearly totals.

```
100 REM MONTHLY PROFIT REPORT                                  PROGRAM 8-4
110 REM
120 REM THIS PROGRAM COMPUTES MONTHLY PROFIT AND AVERAGE DAILY
130 REM PROFIT FOR TWELVE MONTHS OF REVENUE AND EXPENSE DATA.
140 REM THE AVERAGE DAILY PROFIT EQUALS THE MONTHLY PROFIT DIVIDED
150 REM BY THE NUMBER OF DAYS IN THE MONTH
160 REM
200 REM VARIABLES         M$=MONTH NAMES ARRAY
210 REM  D=NUMBER OF DAYS IN EACH MONTH ARRAY
220 REM  R=REVENUE ARRAY
230 REM  E=EXPENSE ARRAY
235 REM  P=PROFIT ARRAY
240 REM  A=AVERAGE DAILY PROFIT ARRAY
250 REM  I=LOOP INDEX
260 REM  X=TOTAL REVENUE FOR THE YEAR
270 REM  Y=TOTAL EXPENSES FOR THE YEAR
280 REM  Z=TOTAL PROFIT FOR THE YEAR
290 REM
300 PRINT ,"      MONTHLY PROFIT REPORT"
305 PRINT
310 DIM M$(12),D(12),R(12),E(12),P(12),A(12)       ! DIMENSIONS
320 FOR I=1 TO 12
330   READ M$(I),D(I),R(I),E(I)       ! READ DATA ITEMS
340 NEXT I
350 REM PROCESSING SECTION
360 FOR I=1 TO 12
370   P(I)=R(I)-E(I)                  ! COMPUTE PROFIT ARRAY
380   A(I)=P(I)/D(I)                  ! COMPUTE AVG DLY PROFIT
390 NEXT I
```

(continued)

```
400 X=0                                          ! INITIALIZE TOTALS
410 Y=0
420 Z=0
430 FOR I=1 TO 12
440   X=X+R(I)                                   ! COMPUTE TOTAL REVENUE
450   Y=Y+E(I)                                   ! COMPUTE TOTAL EXPENSES
460   Z=Z+P(I)                                   ! COMPUTE TOTAL PROFIT
470 NEXT I
480 REM OUTPUT SECTION
490 PRINT "MONTH","REVENUE","EXPENSES","PROFIT","AVG DAILY PROFIT"
495 PRINT
500 FOR I=1 TO 12
510   PRINT M$(I),R(I),E(I),P(I),A(I)
520 NEXT I
530 PRINT
540 PRINT "TOTALS",X,Y,Z
600 DATA "JANUARY",31,10000,8000
610 DATA "FEBRUARY",28,7800,6000
620 DATA "MARCH",31,9500,8000
630 DATA "APRIL",30,8800,4000
640 DATA "MAY",31,9000,6000
650 DATA "JUNE",30,8000,6700
660 DATA "JULY",31,12000,8000
670 DATA "AUGUST",31,14000,12000
680 DATA "SEPTEMBER",30,10000,6000
690 DATA "OCTOBER",31,9000,6700
700 DATA "NOVEMBER",30,8000,5000
710 DATA "DECEMBER",31,7000,4000
999 END
RUNNH
```

MONTHLY PROFIT REPORT

MONTH	REVENUE	EXPENSES	PROFIT	AVG DAILY PROFIT
JANUARY	100000	8000	2000	64.5161
FEBRUARY	7800	6000	18 00	64.2857
MARCH	9500	8000	15 00	48.3871
APRIL	8800	4000	4800	160
MAY	9000	6000	3000	96.7742
JUNE	8000	6700	1300	43.3333
JULY	12000	8000	4000	129.032
AUGUST	14000	12000	2000	64.5161
SEPTEMBER	10000	6000	4000	133.333
OCTOBER	9000	6700	2300	74.1936
NOVEMBER	8000	5000	3000	100
DECEMBER	7000	4000	3000	96.7742
TOTALS	113100	80400	32700	

Ready

CHAPTER 9

SUBROUTINES

A. OVERVIEW

By this point, you are well aware of the many complex possibilities of programs in the BASIC language. Even with a relatively small example, it is possible to "get lost" in the logic of a program. We have shown, by example, various ways to keep a program organized so that major sections are easy to find, individual statements are easy to read, and the program generally is easy to follow and understand. This chapter presents an additional programming tool to help the reader improve the organization of his or her programs.

As programs increase in size, they usually increase in complexity. This increase is likely to be exponential so that large programs are many times more complicated than smaller ones. An objective of programming then should be to keep programs as small as possible. Of course, this is practically impossible, so the next best thing is to divide programs into modules that are easily managed. These modules should deal with a limited number of processes and have very specific tasks to perform. When all the modules are complete, they can be assembled into the complete program. An analogy is the construction of an automobile. The engine, the drive train, the body, and other modules are built individually and then assembled to form a complete automobile. The "building blocks" of computer programs are called subroutines.

Subroutines are useful for program development and maintenance. Even the most simple of programs could contain three subroutines; one for data input, another for proccessing, and a third for output. As a program grows in function and size, this modularity helps the programmer manage the complexity in a simple and organized manner.

In fact, very large programming projects often exceed the ability of a single programmer. Here a team of programmers is used. The project is broken into modules and each team member is responsible for writing and testing one or more modules. They work in parallel to complete all of the pieces. The team leader is in charge of assembling all of the modules into the complete program.

After a large program is implemented, it sometimes needs to be updated or "maintained." An example is a payroll program that needs to change in order to reflect changes in tax laws. When such a program needs to be updated, just the module or modules involved with the new procedures need to be changed without affecting the rest of the program. This is an important point since program maintenance represents the single largest portion of typical data processing budgets. Efficient program maintenance can lower this cost substantially.

In addition to the program development and maintenance advantages gained with program modularity, subroutines can also decrease the size of computer programs. A subroutine is a section of program lines that can be used many times within a program. Thus, if a certain function is to be performed multiple times, it makes sense to write it as a subroutine and simply refer to the subroutine each time it is needed. This results in a smaller program than would exist if the program lines were repeated each time the function was performed.

An extension to this is the development of subroutine "libraries." Certain programming functions may appear throughout a system of programs. For example, the sorting of data (into alphabetical or numerical order) may need to be performed many times in a given application. Rather than rewrite the sorting program lines each time they are needed, a general-purpose sort subroutine could be written and referred to in all of the programs that require it. (For more information on sorting, see Chapter 11.)

The programming examples in this chapter demonstrate subroutines as they are used to make programs modular. Examples in following chapters show how subroutines can be used to decrease the size and complexity of larger programs.

B. GOSUB AND RETURN STATEMENTS

In BASIC, a subroutine is defined as any group of lines that begin at a certain point and end with a RETURN statement. The subroutine is referenced with the following statement:

```
line # GOSUB subroutine line #
```

which means "transfer control to the subroutine line #." In a way, the GOSUB statement is similar to the GOTO statement discussed earlier. There is an important difference, however. The GOTO statement transferred to another line much the same way that a GOSUB statement transfers to another line. But, with a GOSUB statement, it is easy to return to the point in the program immediately following the GOSUB statement.

The subroutine ends with a RETURN statement, which means "transfer back to the line immediately following the GOSUB statement that was used to transfer control to this subroutine." This way, a subroutine can be "called" from anywhere in the program and, after it is used, the subroutine will transfer control back to the correct place in the program.

The general form of the RETURN statement is:

```
line # RETURN
```

A RETURN statement is the final line in a subroutine. When the RETURN is executed, control is transferred to the line following the GOSUB statement which was used to transfer to the subroutine in the first place. The following example shows this:

```
100 ......
110 GOSUB 500        ! Branch to subroutine
120 ......
130 GOSUB 500        ! Branch to subroutine
140 ......
 .
 .
 .
500 ......
510 ......
520 ......
530 RETURN           ! Return to main program
```

The program starts at line 100. At line 110, the program transfers to the subroutine starting at line 500. Lines 500 through 520 are executed and the RETURN at line 530 transfers control back to the "main" routine, line 120. After line 120 is executed, line 130 calls subroutine 500 again. The subroutine is executed and control is returned to the main program, line 140.

In general, programs that use subroutines will be divided into two major sections: the "main" routine and the subroutines. The main routine frequently does little more than "call" the subroutines and control the overall processing.

Subroutines are usually placed near the end of a program following any DATA statements and preceding the END statement. A subroutine can have multiple entry points (line numbers) and multiple exits (RETURN statements) as long as the correct logic flow is maintained.

C. NESTED SUBROUTINES

One of the reasons for using subroutines is to reduce the overall size and complexity of a program. But what happens if a subroutine gets large and complex? Can a subroutine use other subroutines? The answer is yes and, in fact, these types of subroutines are given a special name: nested subroutines.

The thing to remember here is that the subroutines are on different "levels." The main program is the top level. If it calls a subroutine, that is considered one level lower. If that subroutine, in turn, calls another subroutine (the nested one), it is considered to be two levels below the top. It's as if we were talking about an underground parking garage. The street is the top level and the floors below street level are numbered from the top down. We drive from the street to level 1, then down to level 2 etc. until we find a place to park. When it's time to leave the garage, we drive back up past the different levels and end up back on the street.

Nested subroutines work the same way. The program gets to a certain nesting level with multiple GOSUB statements. Each subroutine ends with a RETURN statement which transfers control to the next higher level until the program returns to the top.

For example, study the following program segment:

```
 10 GOSUB 100              ! Branch to first subroutine
 20 PRINT "DONE"
 30 GOTO 999               ! Branch to end of program
  .
  .
100 INPUT A,B              ! Subroutine 1
110 GOSUB 200              ! Branch to second subroutine
120 PRINT C
130 RETURN                 ! Return to main program
  .
  .
200 C = A + B              ! Subroutine 2
210 RETURN                 ! Return to second subroutine
  .
  .
999 END
```

In this example, the main program consists of lines 10-90. the first level subroutine is lines 100-130. The nested subroutine (second level) is lines 200-210.

The program executes as follows:

1. line 10 transfers control to subroutine 100.

2. statement 100 is executed (two numeric variables, A and B, are input).

3. line 110 transfers control to the nested subroutine 200.

4. statement 200 is executed (the two variables are added together and the result is placed in variable C).

5. line 210 returns control to the first subroutine, line 120.

6. line 120 is executed (the variable C is output).

7. line 130 transfers control back to the main program, line 20.

8. line 20 is executed (the message "DONE" is printed).

9. statement 30 transfers control to the end of the program, line 999.

Subroutines can be nested to any depth necessary. The maximum level depends on the program size and the amount of memory available.

D. MULTI-BRANCH SUBROUTINE OPERATIONS

A multi-branch subroutine can be set up in one statement so that a program can call one of a list of specified subroutines based on the result of an expression. This is accomplished with the ON-GOSUB statement, which has the general form:

```
line # ON variable GOSUB list of line numbers
```

This statement is similar in function to the ON GOTO statement described in Chapter 6.

For example, these five statements:

```
10 IF X = 1 THEN GOSUB 100
20 IF X = 2 THEN GOSUB 200
30 IF X = 3 THEN GOSUB 300
40 IF X = 4 THEN GOSUB 400
50 IF X = 5 THEN GOSUB 500
```

could be written in only one statement as follows:

```
10 ON X GOSUB 100,200,300,400,500
```

PROGRAM 9-1 COMPOUND INTEREST

A modular programming approach is used to find the future value of an amount with compound interest. Three subroutines are used for input, calculation and output modules. The main program "calls" the three subroutines and then transfers to the end.

The input subroutine (starting at line 1000) prints the program title; inputs the principal (P), interest rate (I), and number of compounding periods (N); and then returns to the main program. The calculation subroutine (starting at 2000) computes the future value (F) using the following formula:

$$\text{future value} = \text{principal} * (1 + \text{interest rate})^{N}$$

After the calculation, the subroutine returns to the main program. The output subroutine (starting at 3000) prints the future value and returns to the main program.

```
100 REM COMPOUND INTEREST PROGRAM                        PROGRAM 9-1
110 REM
120 REM THIS PROGRAM COMPUTES THE FUTURE VALUE OF AN AMOUNT USING
130 REM THE STANDARD COMPOUND INTEREST FORMULA.
140 REM                    VARIABLES
150 REM P=STARTING PRINCIPAL              I=INTEREST RATE
160 REM N=NUMBER OF COMPOUNDING PERIODS   F=FUTURE VALUE
200 REM MAIN ROUTINE
210 GOSUB 1000            ! CALL THE INPUT SUBROUTINE
220 GOSUB 2000            ! CALL THE PROCESSING SUBROUTINE
230 GOSUB 3000            ! CALL THE OUTPUT SUBROUTINE
240 GOTO  9999            ! GOTO END OF PROGRAM
1000 PRINT "COMPOUND INTEREST PROGRAM":PRINT  ! INPUT SUBROUTINE
1030 INPUT "WHAT IS THE STARTING PRINCIPAL";P
1040 INPUT "WHAT IS THE INTEREST RATE PER PERIOD";I
1050 INPUT "HOW MANY COMPOUNDING PERIODS";N
1060 RETURN
2000 REM                                      ! PROCESSING SUBROUTINE
2010 F=P*(1+I)^N  ! COMPUTE FUTURE VALUE
2020 RETURN
3000 PRINT:PRINT "THE FUTURE VALUE IS $";F ! OUTPUT SUBROUTINE
3010 RETURN
9999 END
RUNNH

COMPOUND INTEREST PROGRAM

WHAT IS THE STARTING PRINCIPAL? 1000
WHAT IS THE INTEREST RATE PER PERIOD? .01
HOW MANY COMPOUNDING PERIODS? 24

THE FUTURE VALUE IS $ 1269.73

Ready
```

PROGRAM 9-2 AMORTIZATION SCHEDULE PROGRAM

This is the same as program 7-2, except that the calculations of interest paid, principal paid, and remaining balance are done in a subroutine (starting at line 400).

```
100 REM AMORTIZATION SCHEDULE PROGRAM                PROGRAM 9-2
110 REM
120 REM THIS PROGRAM PRINTS AN AMORTIZATION SCHEDULE FOR A LOAN.
140 REM THE SCHEDULE SHOWS PAYMENT NUMBER, PAYMENT AMOUNT, INTEREST
150 REM PAID, PRINCIPAL PAID, AND REMAINING BALANCE.
160 REM                      VARIABLES:
170 REM B=LOAN BALANCE         X=LOOP INDEX     R=INTEREST RATE
180 REM N=NUMBER OF PAYMENTS   I=INTEREST PAID  A=PAYMENT AMOUNT
190 REM P=PRINCIPAL PAID       F$=OUTPUT FORMAT STRING
200 REM
220 PRINT,"AMORTIZATION SCHEDULE PROGRAM":PRINT
230 PRINT "PAYMENT  PAYMENT   INTEREST  PRINCIPAL  REMAINING"
240 PRINT "NUMBER   AMOUNT      PAID      PAID      BALANCE
250 F$=    " ###     #,###.##  #,###.##  #,###.##   #,###.##"
300 B=1000                  ! BALANCE
310 N=10                    ! NUMBER OF PAYMENTS
320 A=149.03                ! PAYMENT AMOUNT
330 R=.08                   ! INTEREST RATE
340 FOR X=1 TO N
350    GOSUB 400            ! CALL THE PROCESSING SUBROUTINE
360    PRINT USING F$,X,A,I,P,B
370 NEXT X
380 GOTO 500                ! GOTO END OF PROGRAM
400 REM PROCESSING SUBROUTINE
410 I=B*R                   ! COMPUTE INTEREST PAID
420 P=A-I                   ! COMPUTE PRINCIPAL PAID
430 B=B-P                   ! COMPUTE REMAINING BALANCE
440 RETURN
500 END
RUNNH
```

```
           AMORTIZATION SCHEDULE PROGRAM

PAYMENT   PAYMENT    INTEREST  PRINCIPAL  REMAINING
NUMBER    AMOUNT       PAID      PAID      BALANCE
   1        149.03      80.00     69.03     930.97
   2        149.03      74.48     74.55     856.42
   3        149.03      68.51     80.52     775.90
   4        149.03      62.07     86.96     688.94
   5        149.03      55.12     93.91     595.03
   6        149.03      47.60    101.43     493.60
   7        149.03      39.49    109.54     384.06
   8        149.03      30.72    118.31     265.75
   9        149.03      21.26    127.77     137.98
  10        149.03      11.04    137.99      -0.01
Ready
```

CHAPTER 10

USING DATA FILES

A. OVERVIEW

The processing power of the computer would be severely restricted without a way to store and retrieve data in a convenient manner. Business transactions usually affect data that has been previously processed. For example, inventory records must be changed when stock items are ordered or sold; payroll records must be changed when an employee's salary changes; credit records must be changed when customers charge items or pay bills. It would be nice to store all the company's data in the processor memory, but the memory's size is relatively small compared to the amount of data available and using it for storage is relatively expensive compared to storing data on some external device.

The file is the main unit of external storage. Files can contain either programs or data. The most common media used to store files outside the processor are magnetic tapes and magnetic disks. We will limit our discussion to disk files since they are the primary means of external storage in today's computer systems. Their great advantage over tapes is the speed of data access that they provide. Tapes, however, are less expensive per unit of data stored and are very useful for purposes of backup and archival storage.

The files that have been discussed so far all contain computer programs. The system commands mentioned in Chapter 1 (SAVE, OLD, REPLACE, etc) enable the programmer to transfer a computer program between the disk and the processor work area. The data used by the programs was either entered interactively by the INPUT statement or was obtained from DATA statements used in conjunction with the READ statement. These methods are satisfactory for small applications, but become quite impractical when large amounts of data are to be processed. The INPUT statement makes data input very slow, and data contained in DATA statements are difficult to change since actual program statements must be altered.

Quick and efficient access to data can be obtained by storing it separately from the program statements in disk files. When the program is executed it will specify, by means of a special statement, which file in the disk catalog contains the data to be processed. Each subsequent input operation will then retrieve one data record from the disk file and the various data values in the record will be associated with appropriate program variables. This enables the program to read in only as much data as is needed.

The way a program accesses data in a file depends on how the file is structured. The two main file structures are sequential and direct. Each structure is best suited to a particular type of application. Students who are just beginning to learn programming will use a relatively small amount of data and, for this purpose, sequential files are most convenient. Thus we will limit our discussion to this type of file.

B. SEQUENTIAL FILES

Records contained in sequential files appear as one continuous stream of data. The records are retrieved in a manner similar to the READ and DATA statements. Each time a READ statement is executed, the next values not yet read are taken from the DATA list. Thus data is retrieved from the DATA "storage area" in a sequential manner, starting with the first value and proceeding forward through the last value.

The data in a sequential file is essentially one long record, but the programmer wants to process only "parts" of that record separately. The entire record in the file is called a physical record and the "parts" are called logical records. For example the following list represents the records in a personnel file. Each record has two items, an employee number and a name:

EMPLOYEE #	EMPLOYEE NAME
1243	SMITH
6452	LEE
3721	JACKSON

If these records were placed in a sequential file, the resulting physical record might look like this:

1243SMITH6452LEE3721JACKSON

This is a very compact storage scheme, but it is difficult to determine where one logical record (an employee number and name) ends and the next begins. If all the names were of the same length then the programmer would know that logical records are found at specific character positions within the physical record. But this is not usually the case since most alphanumeric data (names, addresses, etc) vary in length. One way to overcome this difficulty is to force each logical record to be a specific length by adding a number of blanks to certain data items. Thus the physical record above might be changed to:

1243SMITH 6452LEE 3721JACKSON

Here each record is exactly eleven characters long. The length chosen is that needed by the longest name. Shorter names are "blank filled" at the end. The relative starting point of each logical record can now be calculated. However, the items within each record still cannot be distinguished, and now valuable file space is being wasted by the added blanks.

A second approach is to separate each item with a special character, for instance:

```
1243*SMITH*6452*LEE*3721*JACKSON
```

An asterisk (*) is being used as a delimiter. It signals the end of each data item. Thus knowing the number of items in a record and their proper order, each item can be read into an appropriate program variable.

There is still one problem that has not been addressed. How does the programmer actually break apart the physical record into logical records? Extracting a group of characters or searching for a specific character (the delimiter) both involve the use of text processing functions (Chapter 12) and can be very time consuming. To aid the programmer in this matter BASIC provides two special delimiters for separating data items and logical records. Using these delimiters makes storing and retrieving data from the disk file an easy task. Refer to section D of this chapter for details.

C. OPENING AND CLOSING FILES

Since many files can exist in a disk catalog, the programmer must have some way of specifying which file or files are to be used in a particular program. This is accomplished by the OPEN statement. The same statement is used to create a new file. The general form of the OPEN statement is:

```
line #  OPEN "filespec" FOR {INPUT  } AS FILE # filenumber
                         {OUTPUT }
```

The words in capitals do not change. The other fields are:

"filespec" - A BASIC file specification. This consists of two groups of alphanumeric characters separated by a period. The first group is called the filename and the second group is called the file extension. The filename can be up to six characters long and the extension up to three characters long. Numbers may be included, but both the filename and extension must start with an alphabetic character. Since these files contain data, common extention names are DAT and TXT. One example program in this chapter contains credit account data, so a filespec of CREDIT.DAT was chosen. Note that the period separating the filename and extension is part of the complete file specification. The file specification must be enclosed in quotes.

Either the word INPUT or the word OUTPUT is entered. This signifies that the file is to be read from, or written to, respectively.

filenumber - A positive integer used in the program statements which perform the actual reading and writing. This number is also referred to as the internal or relative file number, or as the channel number. There must be a separate file number for each file being used in the program. An example of OPENing a file for input is:

```
100 OPEN "CREDIT.DAT" FOR INPUT AS FILE #1
```

This statement assumes that the file "CREDIT.DAT" already exists.

OPENing a file for output would be:

```
100 OPEN "CREDIT.DAT" FOR OUTPUT AS FILE #1
```

If the file named in the OPEN statement does not exist in the catalog, a new file will be created by the operating system. In most versions of BASIC, OPENing a file for sequential output will erase any records previously stored in the file. Some versions will allow records to be added on the the end of an existing file, thus saving all previous records.

When the user is finished reading from or writing to a file, the file should be closed. This is accomplished by the statement:

```
line # CLOSE # filenumber
```

where filenumber is the same filenumber used in the OPEN statement. Closing the file mentioned above could be done by:

```
200 CLOSE #1
```

D. STORING AND RETRIEVING DATA

1. Writing Data to a File

Since a file must obviously be created before it can be read, storing data in files is discussed first. The common phrase that indicates data being placed in a file is "writing to" the file. This is accomplished in BASIC by the PRINT # statement. The general form of the statement is:

```
line # PRINT # filenumber, list of variables
```

filenumber - The same file number specified in the OPEN statement.
list of variables - A list of the data items that are to be written to the file as one logical record.

The delimiters mentioned earlier that separate data items and records can now be introduced. The comma (,) separates data items and must be explicitly written to the file as a character placed between quotes. For example:

```
150 PRINT #1,N$","A
```

This statement places the data values of N$ and A into file #1. The actual name of file #1 is known by the system because of the OPEN statement. If file #1 was "CREDIT.DAT" then these data items would be written to that file. PRINTing to a file that is not OPEN will produce an error message. The end-of-record delimiter is automatically written to the file after the PRINT # statement is executed. Thus the programmer does not have to be concerned with including this in the PRINT # statement. Let's indicate an end-of-record character with the symbol <EOR>. The following program will write to records to a file named "ACCT.DAT":

```
100 OPEN "ACCT.DAT" FOR OUTPUT AS FILE# 1
110 A$ = "SMITH"              ! PUT VALUES INTO VARIABLES
120 X = 12.32
130 B$ = "JONES"
140 Y = 27.23
150 PRINT #1, A$","X          ! WRITE 1ST RECORD
160 PRINT #1, B$","Y          ! WRITE 2ND RECORD
170 CLOSE #1
180 END
RUNNH

Ready
```

The physical record in the file "ACCT.DAT" would look like

```
SMITH,12.32<EOR>JONES,27.23<EOR>
```

Note that there is no output displayed on the screen. The "output" in this case is this disk file.

2. Reading Data from a File

Data is retrieved from, or "read from" a sequential file by the INPUT # statement. The general form of this statement is:

```
line# INPUT # filenumber, list of variables
```

The file number and list of variables have the same meaning as before. The special delimiters written to the file in the PRINT # statement now make the retrieval of individual items an easy process. A comma (not in quotes) separates each variable in the list. For example:

```
200 INPUT #1,N$,A
```

This statement reads two data items from file #1. The values of these items are assigned to the variables N$ and A respectively. Of course the number and type of variables in the INPUT # statement must match those that were used in the PRINT # statement when the record was written to the file. If this rule is not followed, then unpredictable results will occur. The following program will read and print the data that was stored in file "ACCT.DAT" by the previous program:

```
100 OPEN "ACCT.DAT" FOR INPUT AS FILE #1
110 INPUT #1,A$,X              ! READ 1ST RECORD
120 INPUT #1,B$,Y              ! READ 2ND RECORD
130 PRINT A$,X
140 PRINT B$,Y
150 CLOSE #1
160 END
RUNNH

SMITH          12.32
JONES          27.23

Ready
```

PROGRAM 10-1A CREDIT ACCOUNT DATA FILE/CREATION

Business transactions will normally affect data stored in existing files. Existing records can be changed or deleted, or new records can be added. But how do "existing" records get into the file? There is usually an initialization program, such as this one, which creates a new file and writes a few records to the file. Subsequent transactions (adds, deletes, changes) can then access the existing file.

This program is run only once. It creates the file CREDIT.DAT and writes credit account records which it has obtained from DATA statements. Each record contains an account number, customer name, credit limit and beginning balance due. Since this is a sequential file, a delimiter (the comma) must be written to the file after each data item that is written (see statement 240).

In data processing, performing business transactions is called updating a file. Program 10-1B will perform updates (add, delete, change) to the records in CREDIT.DAT. Program 10-1C will read and print out the records in a report.

Note that this program has no output that is displayed. Its "output" is sent to the data file. The record with account number "LAST RECORD" is a sentinel that is written to the file and used by the next two programs to signal the end of data.

```
100 REM CREDIT ACCOUNT DATA FILE/CREATION          PROGRAM 10-1A
110 REM
120 REM CREDIT ACCOUNT DATA IS READ FROM DATA STATEMENTS AND
130 REM THEN WRITTEN TO A FILE CALLED "CREDIT.DAT"
140 REM                    VARIABLES
140 REM A$=ACCOUNT NUMBER              L=CREDIT LIMIT
160 REM N$=CUSTOMER NAME               B=BEGINNING BALANCE
170 REM
200 REM
210 OPEN "CREDIT.DAT" FOR OUTPUT AS FILE #1        ! OPEN THE FILE
220 READ A$,N$,L,B                     ! READ DATA ITEMS
230 IF A$="END" THEN 260 ! CHECK FOR SENTINEL
240 PRINT #1,A$","N$","L","B          ! PRINT DATA RECORD
250 GOTO 220
260 CLOSE #1                           ! CLOSE THE FILE
300 DATA "A-100","SMITH",500,250
310 DATA "G-300","JONES",500,125
320 DATA "D-325","MILLER",600,257
330 DATA "D-199","WEST",750,125
340 DATA "R-341","O'MALLEY",650,100
350 DATA "LAST RECORD","Z",0,0
360 DATA "END","00000",0,0
370 END
RUNNH

Ready
```

PROGRAM 10-1B CREDIT ACCOUNT DATA FILE/UPDATE

The records in the credit account file that were created in program 10-1A will have to be changed whenever a customer makes a purchase. Also, new customers will have to be added to the file and inactive accounts must be deleted. The following program can be used for these update functions.

INPUT SECTION (statements 320-390)

Records are read from file CREDIT.DAT after it is OPENed for input. The sentinel record has a customer name of "LAST RECORD". The data items in each record are read into appropriate arrays. A counter is incremented after each INPUT statement. The counter becomes the subscript of the next array element to be filled. When the sentinel record is encountered, the previous value of the subscript is a count of how many records are in the file. The file is then CLOSEd and the program proceeds to the update section.

UPDATE SECTION (statements 400-870)

A menu of choices is presented to the user: Add (A), Change (C), Delete (D), or End the program (E). The following actions are taken for each choice:

Add new records

A check is made to see that the file still has room for more records. Since the data arrays have been DIMensioned to 100, this is the maximum number of records which can be processed by the program. This limit was arbitrarily set for this example. Most processors will have enough memory to allow a much greater number of records.

The program prompts for the new account number and compares it to the current list of accounts. The account number Z-999 is not allowed since this is used in the record deletion process (see Deleting Records). If the number entered is already on file, a message notifies the user of this fact and the program returns to the update menu. If no duplicate account number exists the user is prompted for customer name, credit limit and beginning balance. The variable N, which stores a count of the number of records in the file, is incremented by one and the new value is used as the subscript of the array elements which receive the new data.

Changing Records

The user is prompted for account number and a check is made that the account is on file. If not, a message is displayed that the account does not exist and the program returns to the update menu. If the account number is valid, the currrent credit limit and account balance are displayed and the user is prompted to input new values for these items.

Deleting Records

The user is prompted for account number and a check is made that the account is on file. If so, the customer name, credit limit and current balance are displayed and the user is prompted to verify (Y/N) that this is the account to be deleted. If the transaction is verified, the account number is changed to Z-999, a special value that is used to signal the output portion of the program that the record is not to be written to the output file.

OUTPUT SECTION (statements 900-990)

File CREDIT.DAT is OPENed for sequential output. A loop is started from 1 to N to output the data in the arrays. If a delete flag is encountered (account Z-999) the record is not output. When the sentinel record is reached it is output to the file; CREDIT.DAT is then CLOSEd and the program ends.

```
100 REM CREDIT ACCOUNT DATA FILE/UPDATE              PROGRAM 10-1B
110 REM
120 REM UPDATE OF CREDIT ACCOUNT DATA CONTAINED IN THE FILE
103 REM CREDIT.DAT. ADDS, CHANGES, DELETIONS ALLOWED.
140 REM SEQUENTIAL INPUT AND OUTPUT
150 REM
160 REM                     VARIABLES
170 REM A$(I)=ACCOUNT NUMBER        T$=TRANSACTION CODE (A,C,D)
180 REM N$(I)=CUSTOMER NAME         N=# RECORDS IN FILE
190 REM L(I)=CREDIT LIMIT           T=# RECORDS WRITTEN TO FILE
200 REM B(I)=CURRENT BALANCE
210 REM
220 DIM A$(100),N$(100),L(100),B(100)
230 PRINT "CREDIT ACCOUNT FILE UPDATE PROGRAM"
300 REM ------------------------------------------------- INPUT
310 PRINT
320 OPEN "CREDIT.DAT" FOR INPUT AS FILE #1
330 I = 1
340 INPUT #1,A$(I),N$(I),L(I),B(I)
350 IF A$(I)="LAST RECORD" THEN 370
360 I = I + 1: GOTO 340
370 N = I - 1: CLOSE #1
380 PRINT "THERE ARE CURRENTLY";N;"RECORDS IN THE CREDIT FILE"
400 REM ------------------------------------------------- UPDATE
500 PRINT: PRINT "ENTER TRANSACTION CODE:
520 INPUT "ADD (A), CHANGE (C), DELETE (D), END (E)";T$
530 IF T$="E" THEN 900
540 IF T$="A" THEN 600
550 IF T$="C" THEN 700
560 IF T$="D" THEN 800
570 PRINT "ILLEGAL TRANSACTION CODE. TRY AGAIN": GOTO 500
```

(continued)

```
600 PRINT: PRINT "ADD NEW CREDIT CUSTOMER"  ! ---------------- Add
605 IF N = 100 THEN PRINT "FILE FULL.": GOTO 500
610 INPUT "NEW ACCOUNT NUMBER";A1$
620 IF A1$="Z-999" THEN PRINT "ILLEGAL ACCOUNT NUMBER":GOTO 500
630 FOR I = 1 TO N
640 IF A1$=A$(I) THEN PRINT "ACCOUNT ";A1$;" ALREADY ON FILE":
    GOTO 500
650 NEXT I
660 N = N + 1
665 A$(N) = A1$
670 INPUT "CUSTOMER NAME";N$(N)
680 INPUT "CREDIT LIMIT";L(N)
690 INPUT "BEGINNING BALANCE";B(N)
695 GOTO 500
700 PRINT: PRINT "CHANGE CREDIT RECORD"  ! --------------- Change
710 INPUT "ACCOUNT NUMBER";A1$
715 IF A1$="Z-999" THEN PRINT "ILLEGAL ACCOUNT NUMBER": GOTO 500
720 FOR I = 1 TO N
725 IF A1$ = A$(I) THEN 740
730 NEXT I
735 PRINT "ACCOUNT ";A1$;" NOT ON FILE": GOTO 500
740 PRINT "CUSTOMER NAME: ";N$(I)
745 PRINT "CURRENT CREDIT LIMIT:";L(I)
750 PRINT "CURRENT ACCOUNT BALANCE:";B(I)
755 INPUT "CHANGE CREDIT LIMIT (Y/N)";X$
760 IF X$="N" THEN 780 ELSE IF X$ <> "Y" THEN 755
770 INPUT "NEW CREDIT LIMIT";L(I)
780 INPUT "NEW ACCOUNT BALANCE";B(I)
790 GOTO 500
800 PRINT: PRINT "DELETE CREDIT RECORD"  ! --------------- Delete
810 INPUT "ACCOUNT NUMBER";A1$
815 IF A1$="Z-999" THEN PRINT "ILLEGAL ACCOUNT NUMBER": GOTO 500
820 FOR I = 1 TO N
825 IF A1$ = A$(I) THEN 840
830 NEXT I
835 PRINT "ACCOUNT ";A1$;" NOT IN FILE": GOTO 500
840 PRINT "CUSTOMER NAME: ";N$(I)
845 PRINT "CREDIT LIMIT:";L(I)
850 PRINT "CURRENT BALANCE:";B(I)
855 INPUT "PLEASE VERIFY FOR DELETION (Y/N)";X$
860 IF X$ = "Y" THEN A$(I) = "Z-999": GOTO 500
870 IF X$ <> "N" THEN 855 ELSE 500
900 REM ------------------------------------------------ OUTPUT
910 OPEN "CREDIT.DAT" FOR OUTPUT AS FILE #1
920 FOR I = 1 TO N
930 IF A$(I) = "Z-999" THEN 960
940 PRINT #1, A$(I)","N$(I)","L(I)","B(I)
950 T = T + 1
960 NEXT I
970 PRINT #1, "LAST RECORD";",";"Z";0","0
980 PRINT T;"ACCOUNT RECORDS WRITTEN TO FILE "
990 CLOSE #1
999 END
```

```
RUNNH

CREDIT ACCOUNT FILE UPDATE PROGRAM

THERE ARE CURRENTLY 5 RECORDS IN THE CREDIT FILE

ENTER TRANSACTION CODE:
ADD (A), CHANGE (C), DELETE (D), END (E)? A

ADD NEW CREDIT CUSTOMER
NEW ACCOUNT NUMBER? G-400
CUSTOMER NAME? JOHNSON
CREDIT LIMIT? 600
BEGINNING BALANCE? 200

ENTER TRANSACTION CODE:
ADD (A), CHANGE (C), DELETE (D), END (E)? C

CHANGE CREDIT RECORD
ACCOUNT NUMBER? G-300
CUSTOMER NAME: JONES
CURRENT CREDIT LIMIT: 500
CURRENT ACCOUNT BALANCE: 125
CHANGE CREDIT LIMIT (Y/N)? N
NEW ACCOUNT BALANCE? 300

ENTER TRANSACTION CODE:
ADD (A), CHANGE (C), DELETE (D), END (E)? D

DELETE CREDIT RECORD
ACCOUNT NUMBER? A-100
CUSTOMER NAME: SMITH
CREDIT LIMIT: 500
CURRENT BALANCE: 250
PLEASE VERIFY FOR DELETION (Y/N)? Y

ENTER TRANSACTION CODE:
ADD (A), CHANGE (C), DELETE (D), END (E)? E

5 ACCOUNTS WRITTEN TO FILE

Ready
```

PROGRAM 10-1C CREDIT ACCOUNT DATA FILE/REPORT

The credit account records which were created by Program 10-1A and updated by Program 10-1B are input and printed in a report format.

The program starts by printing column headings. The data file CREDIT.DAT is then OPENed for input.

Data records are read one at a time (statement 260) and the data items are printed as a report detail line. Before the print statement, a check is made to see if the sentinel account "LAST RECORD" has been encountered (statement 270). If so, the data file is CLOSEd and the program ends.

```
100 REM CREDIT ACCOUNT DATA FILE/REPORT            PROGRAM 10-1C
110 REM
120 REM THIS PROGRAM INPUTS DATA RECORDS FROM A FILE CALLED
130 REM "CREDIT.DAT" AND THEN PRINTS THE DATA FIELDS.
140 REM
150 REM VARIABLES        A$=ACCOUNT NUMBER
160 REM                  N$=CUSTOMER NAME
170 REM                  L=CREDIT LIMIT
180 REM                  B=CURRENT BALANCE DUE
190 REM
200 REM
210 PRINT "        CREDIT ACCOUNT REPORT PROGRAM"
220 PRINT
230 PRINT "ACCOUNT","CUSTOMER","CURRENT","CREDIT"
240 PRINT "NUMBER","NAME","BALANCE","LIMIT"
250 OPEN "CREDIT.DAT" FOR INPUT AS FILE #1          ! OPEN THE FILE
260 INPUT #1,A$,N$,L,B                  ! INPUT DATA RECORD
270 IF A$="LAST RECORD" THEN 300        ! CHECK FOR SENTINEL
280 PRINT A$,N$,B,L                     ! PRINT DATA FIELDS
290 GOTO 250
300 CLOSE #1                            ! CLOSE THE FILE
999 END
RUNNH
```

```
        CREDIT ACCOUNT REPORT PROGRAM

ACCOUNT         CUSTOMER        CURRENT         CREDIT
NUMBER          NAME            BALANCE         LIMIT

G-300           JONES           300             500
D-325           MILLER          257             600
D-199           WEST            125             750
R-341           O'MALLEY        100             650
G-400           JOHNSON         200             600

Ready
```

PROGRAM 10-2A PAYROLL DATA FILE/CREATION

A sequential file called PAYROL.DAT is created by this program. It will be used as an input file for the next sample program (10-2B). The data used is the same as the previous payroll example (Program 6-4). Each record contains an employee name, pay type (hourly or salaried), number of hours worked, and rate of pay.

```
100 REM PAYROLL DATA FILE/CREATION                    PROGRAM 10-2A
110 REM
120 REM PAYROLL DATA IS READ FROM DATA STATEMENTS AND THEN
130 REM WRITTEN TO A FILE CALLED "PAYROL.DAT".
140 REM
150 REM VARIABLES         N$=EMPLOYEE NAME
160 REM                   T$=PAY TYPE (H=HOURLY S=SALARIED)
170 REM                   H=HOURS WORKED
180 REM                   R=RATE OF PAY
190 REM
200 OPEN "PAYROL.DAT" FOR OUTPUT AS FILE #1
210 READ N$,T$,H,R
220 IF N$="END" THEN 250
230 PRINT #1, N$","T$","L","R
240 GOTO 210
250 CLOSE #1
300 DATA "SMITH","H",35,6.25
310 DATA "JONES","H",44,7.00
320 DATA "JOHNSON","S",40,5.95
330 DATA "MILLER","S",45,11.50
340 DATA "ANDERSON","S",41,3.50
350 DATA "O'MALLEY","H",38,7.50
360 DATA "WEST","S",35,4.85
370 DATA "LAST RECORD","Z",0,0
380 DATA "END","0",0,0
999 END
RUNNH

Ready
```

PROGRAM 10-2B PAYROLL DATA FILE/REPORT

This is the same as Program 6-5 except that the input data comes from a data file rather than from DATA statements.

The file, called "PAYROL.DAT" is opened prior to input. The program inputs the data records (one at a time), checks for the sentinel of "LAST RECORD", and performs the payroll calculation. When the last record is input, the program transfers to a section that closes the file and ends.

```
100 REM PAYROLL DATA FILE REPORT                          PROGRAM 10-2B
110 REM
120 REM THIS PROGRAM COMPUTES NET PAY, OVERTIME, AND GROSS PAY
130 REM FOR A GROUP OF EMPLOYEES. SALARIED EMPLOYEES DO NOT
140 REM QUALIFY FOR OVERTIME PAY. HOURLY EMPLOYEES WORKING MORE
150 REM THAN 40 HOURS ARE PAID 1.5 THEIR REGULAR PAY RATE FOR
160 REM THE HOURS BEYOND 40. THE DATA FOR THIS PROGRAM IS INPUT
170 REM FROM A DATA FILE CALLED "PAYROL.DAT".
180 REM
190 REM                     VARIABLES
200 REM      N$=NAME              T$=PAY TYPE (H=HOURLY, S=SALARIED)
210 REM      H=HOURS WORKED       S=STRAIGHT PAY
220 REM      R=RATE OF PAY        X=OVERTIME PAY
230 REM      G=GROSS PAY          B$=OUTPUT FORMAT STRING
240 REM
250 PRINT "                       PAYROLL PROGRAM"
260 PRINT
270 PRINT "EMPLOYEE   PAY   HOURS   PAY     STRAIGHT   OVERTIME     GROSS"
280 PRINT "NAME       TYPE          RATE      PAY         PAY         PAY"
290 B$ =  "\        \   !    ##    ##.##     $###.##    $###.##    $###.##"
300 OPEN "PAYROL.DAT" FOR INPUT AS FILE #1          ! OPEN THE FILE
310 INPUT #1,N$,T$,H,R                    ! INPUT DATA RECORD
320 IF N$="LAST RECORD" THEN 998
330 IF T$="H" THEN 500                    ! BRANCH TO HOURLY ROUTINE
340 IF T$="S" THEN 400                    ! BRANCH TO SALARIED ROUTINE
350 PRINT "INVALID PAY TYPE CODE (";T$;") FOR ";N$
360 GOTO 310                              ! READ NEXT RECORD
```

(continued)

```
400 REM SALARIED EMPLOYEE
410 S=40*R                                  ! COMPUTE STRAIGHT PAY
420 X=0                                     ! NO OVERTIME
430 G=S                                     ! COMPUTE GROSS PAY
440 GOTO 610
500 REM HOURLY EMPLOYEE
510 IF H>40 THEN 580                        ! BRANCH TO OVERTIME ROUTINE
520 REM NO OVERTIME
530 S=H*R                                   ! COMPUTE STRAIGHT PAY
540 X=0                                     ! NO OVERTIME
550 G=S                                     ! COMPUTE GROSS PAY
560 GOTO 610
570 REM OVERTIME
580 S=40*R                                  ! COMPUTE STRAIGHT PAY
590 X=(H-40)*R*1.5                          ! COMPUTE OVERTIME PAY
600 G=S+X                                   ! COMPUTE GROSS PAY
610 PRINT USING B$,N$,T$,H,R,S,X,G
620 GOTO 310
998 CLOSE #1
999 END

RUNNH

                    PAYROLL PROGRAM

EMPLOYEE PAY  HOURS  PAY    STRAIGHT  OVERTIME    GROSS
NAME     TYPE        RATE     PAY       PAY        PAY

SMITH      H     35    6.25   $218.75   $  0.00   $218.75
JONES      H     44    7.00   $280.00   $ 42.00   $322.00
JOHNSON    S     40    5.95   $238.00   $  0.00   $238.00
MILLER     H     45   11.50   $460.00   $ 86.25   $546.25
ANDERSON   S     41    3.50   $140.00   $  0.00   $140.00
O'MALLEY   H     38    7.50   $285.00   $  0.00   $285.00
WEST       S     35    4.85   $194.00   $  0.00   $194.00

Ready
```

CHAPTER 11

SORTING AND SEARCHING

A. OVERVIEW

We have covered many ways to manipulate and store data. In this chapter, we show how the previously discussed statements can be arranged to perform two common data processing functions -- sorting and searching.

Sorting is the process of arranging a list of data items into some predefined sequence (e.g., alphabetical order, ascending or descending numerical order). A relatively simple program can be written to perform sorting operations quickly.

We're all accustomed to using a phone book. The information is arranged (sorted) in alphabetical order by last name, then first name. For the most common application (i.e., looking up someone's phone number) this order is appropriate. But suppose you were analyzing your long-distance phone bill and couldn't remember who you had called. You would want the phone book to be sorted by phone number (so you could match the numbers from your bill to the names in the phone book). Or, suppose you wanted to know the names of the people living on your street. In this case, you would want the phone book to be sorted by address. (Note: both types of "reverse" phone books do exist, but are available only by subscription.)

Searching is the process of looking through a list of data until a specific value is found. There are different ways to look through a data list, just like there are different ways to look up names and numbers in a phone book. The method used depends on how much data exists and how the data items are sorted. For example, if the phone book contained only ten names, it would be very easy to find any specific name, address, or phone number regardless of how the listings were sorted. In contrast, a phone book containing 50,000 names would have to be pre-arranged and searched through with a precise method (to minimize the overall search time).

This book presents two common programming methods for sorting and searching. The sample programs demonstrate these techniques with data described in previous chapters.

B. SORTING METHODS

In order to sort a list of data, we must know a couple of things. First, how many items are to be sorted? In what order is the current list? Into what predefined sequence must the list be sorted?

For example, the following list contains ten numbers in random order:

```
87  18  72  41  56  25  33  64  91  34
```

It is an easy task to sort the list into increasing order. Look for the smallest number (18 in this case), the next smallest (25), etc in order to create a second (sorted) list, as follows:

```
18  25  33  34  41  56  64  72  87  91
```

A BASIC program could be written to perform this type of sorting without too much difficulty. The program would require two data lists: the unsorted one and the sorted one.

Let's consider another approach, this time with a single data list.

Unsorted list:

```
87  18  72  41  56  25  33  64  91  34
```

If we compare adjacent values (e.g., 87 and 18) and exchange them if the first one is larger than the second one, and repeat this procedure enough times, we will eventually end up with a sorted list.

To describe this method in detail, the 87 and 18 are compared and exchanged. Now, 87 is the second value in the list. It is compared to the third value (72). They too are exchanged and the 87 becomes the third value. The 87 is then compared to the 41 and exchanged.

The following diagram shows the full compare and exchange process. Each successive line of the diagram moves one data value to the right and then does the compare and exchange (if necessary).

```
87  18  72  41  56  25  33  64  91  34
18  87
    72  87
        41  87
            56  87
                25  87
                    33  87
                        64  87  91
                                34  91
```

So, after one pass through the list, it looks like this:

```
18  72  41  56  25  33  64  87  34  91
```

This is still unsorted, but notice that the largest value (91) got "pushed" all the way to the right (end) of the list. Now if we go back to the beginning of the list and do the compare and exchange routine again, we get closer to the sorted list. Here again are the details:

```
18  72  41  56  25  33  64  87  34  91
18  72
    41  72
        56  72
            25  72
                33  72
                    64  72  87
                            34  87  91
```

The procedure can be repeated one more time as follows:

```
18  41  56  25  33  64  72  34  87  91
18  41  56
        25  56
            33  56  64  72
                    34  72  87  91
```

One more time:

```
18  41  25  33  56  64  34  72  87  91
18  41
    25  41
        33  41  56  64
                34  64  72  87  91
```

Once again:

```
18  25  33  41  56  34  64  72  87  91
18  25  33  41  56
                34  56  64  72  87  91
```

And finally:

```
18  25  33  41  34  56  64  72  87  91
18  25  33  41
            34  41  56  64  72  87  91
```

The sorted list is:

```
18  25  33  34  41  56  64  72  87  91
```

The final list shows the values sorted in ascending order. This was accomplished by comparing and exchanging adjacent values in the list. Each pass through the data list "pushed" the next highest data value to the right (end) of the list. This common compare and exchange sorting technique has a special name. It is called the "bubble sort" method because the largest values "float" to the end of the list like underwater bubbles float to the surface.

To write a bubble sort program, the data items must be contained in a array variable. The program is then able to compare adjacent values easily by way of the subscripts. If two values need to be exchanged, a temporary variable must be used to hold one of the values while the other one is being moved.

The pseudocode for the bubble sort method is shown below. The data items are represented by the array ITEM, where ITEM(i) is any given value and ITEM(i+1) is the adjacent value. The variable n is used to represent the number of items to be sorted.

There are two loops in the algorithm. The outer loop (from steps 2 through 7) controls the number of times that the compare and exchange routine is to be performed. Even the worst case of unsorted data could be sorted with n repetitions of the outer loop. Step 3 decreases n by one for each repetition of the inner loop (steps 4 to 6). This decrease is due to the fact that the highest data value is pushed to the end of the list each time through the inner loop (so the compare and exchange routine does not need to be performed all the way to the end of the list each time). Step 5 compares adjacent data values and exchanges them if necessary.

Bubble sort pseudocode:

```
1  ASSIGN n = number of data items to be sorted
2  REPEAT FOR n TIMES
3     ASSIGN n = n - 1
4     REPEAT FOR n - 1 TIMES (i.e., for i = 1 to n-1)
5        IF ITEM(i) > ITEM(i+1)
            THEN ASSIGN temporary = ITEM(i)
                 ASSIGN ITEM(i) = ITEM(i+1)
                 ASSIGN ITEM(i+1) = temporary
6     END REPEAT
7  END REPEAT
```

The sample programs in this chapter use this procedure to sort data values. Notice that the algorithm sorts data into increasing order. If we wanted to sort into decreasing order, we could simply change the ">" symbol to a "<".

C. SEARCHING METHODS

The easiest way to search through a list of data for some specific value is to start at the beginning of the list. Now compare the first data value to the value you're looking for (i.e., the search value). If they match, you've found what you're looking for. If not, go to the next value on the list. Continue this sequential search through the list until either a match is found or until you exhaust the entire list.

Sequential search pseudocode:

```
1  ASSIGN n = number of data items to be searched
2  INPUT search value
3  REPEAT FOR n TIMES (i.e., for i = 1 to n)
4     IF ITEM(i) = search value
         THEN GOTO step 8
5  END REPEAT
6  OUTPUT "No match found"
7  GOTO step 2
8  continue processing
```

A sequential search is easy to understand and to program. It works well for small amounts of data without any regard to how the items are sorted.

A disadvantage is that sequential searching is time consuming. On the average, half of the list must be searched to match any given search value. For a large amount of data, this time would be intolerable. In these cases, a different searching method is used.

The most efficient searching method requires that the data list be sorted into some predefined sequence (e.g., alphabetical order). The first comparison between the list and the search value occurs at the exact middle of the list. If the search value doesn't match, the remaining list is split in half and the mid-point of this list becomes the next comparison value. This splitting in half procedure continues until either a match is found or the entire list is exhausted.

This is reminiscent of the following number guessing game:

```
Player 1:  "I'm thinking of a number between 1 and 100."
Player 2:  "Is it 50?"
Player 1:  "No. That's too low. Guess again."
Player 2:  "How about 75?"
Player 1:  "Nope. That's too high. Try again."
Player 2:  "62?"
Player 1:  "No. Too low."
Player 2:  "68?"
Player 1:  "Yes. You got it in only four guesses."
```

In the previous number guessing game, player 2 splits the original list in half and chooses the mid-point (50 in this case). After finding out that the guess is too low, player 2 divides the upper half of the list in half and chooses that mid-point (75). This process continues until player 2 chooses the right value.

This search method is called a "binary search" or the "method of bisection." It requires that the data list be sorted prior to the search. The search starts in the middle of the list. A comparison is made between the search value and the mid-point value. If the values are equal, the search stops (and subsequent processing continues). If the values do not match, half of the list (the half containing the search value) is divided in half and the mid-point comparison is repeated. This process is repeated until either a match occurs or the entire list is exahusted.

It turns out that the method of bisection is the most efficient way to search through a list of data. On the average, the search value will be matched to a value from the list in the least amount of steps (and time). For large lists of data, this method is the only practical solution to the searching problem.

Since the following sample programs deal with small amounts of data, the sequential search is the only searching method shown.

PROGRAM 11-1 ALPHABETICAL SORT PROGRAM

This program sorts a list of names into alphabetical order using the bubble sort technique. The names, stored in a 1-dimensional array (variable N$), are read from DATA statements and printed in unsorted order.

The sort routine uses two FOR-NEXT loops to place the names in alphabetical order. The major decision occurs at statement 440, where two names are compared. If they are already in alphabetical order, the program transfers to the end of the nested loop. If not, the names are exchanged. The temporary field (T$) is used in this exchange process. After the sort routine, the names are printed in alphabetical order.

```
100 REM ALPHABETICAL SORT                                  PROGRAM 11-1
110 REM
120 REM THIS PROGRAM SORTS A LIST OF NAMES INTO ALPHABETICAL ORDER
130 REM USING THE BUBBLE SORT TECHNIQUE. THE DATA IS READ FROM
140 REM DATA STATEMENTS AND STORED IN AN ARRAY.
150 REM    VARIABLES:
160 REM    N$=NAME ARRAY     N=NUMBER OF NAMES
170 REM    I=LOOP INDEX      L=LOOP LIMIT FOR SORT ROUTINE
180 REM    X=LOOP INDEX      T$=TEMPORARY STRING USED IN SORTING
190 REM    Y=LOOP INDEX
200 REM
300 PRINT "ALPHABETICAL SORT PROGRAM"
305 PRINT
310 DIM N$(50)                ! DIMENSION THE NAME ARRAY
320 READ N                    ! READ THE NUMBER OF NAMES
330 PRINT "UNSORTED DATA:"
340 FOR I= 1 TO N
350   READ N$(I)              ! READ THE NAME ARRAY
360   PRINT N$(I)             ! PRINT THE UNSORTED NAME ARRAY
370 NEXT I
380 PRINT
400 L=N                       ! SET LOOP LIMIT FOR SORT ROUTINE
410 FOR X= 1 TO L
420   L= L - 1                ! ADJUST LOOP LIMIT
430   FOR Y=1 TO L            ! START INNER LOOP
440     IF N$(Y) < N$(Y+1) THEN 480      ! ALREADY IN ORDER?
450     T$=N$(Y)              ! PLACE 1ST NAME INTO TEMP. FIELD
460     N$(Y)=N$(Y+1)         ! MOVE SECOND NAME INTO 1ST FIELD
470     N$(Y+1)=T$            ! MOVE TEMP. FIELD INTO 2ND FIELD
480   NEXT Y
490 NEXT X
500 PRINT "SORTED DATA:"
510 FOR I= 1 TO N
520   PRINT N$(I)             ! PRINT THE SORTED NAME ARRAY
530 NEXT I
590 DATA 20
```

(continued)

```
600 DATA "SMITH","JONES","MILLER","ANDERSON","WEST"
610 DATA "JOHNSON","O MALLEY","SIMS","WALKER","BROWN"
620 DATA "SMYTHE","COOPER","CLARK","DAVIS","LEE"
630 DATA "GARCIA","PARDUCCI","SALGO","BOYNTON","THUNES"
999 END
RUNNH

ALPHABETICAL SORT PROGRAM

UNSORTED DATA:
SMITH
JONES
MILLER
ANDERSON
WEST
JOHNSON
O'MALLEY
SIMS
WALKER
BROWN
SMYTHE
COOPER
CLARK
DAVIS
LEE
GARCIA
PARDUCCI
SALGO
BOYNTON
THUNES

SORTED DATA:
ANDERSON
BOYNTON
BROWN
CLARK
COOPER
DAVIS
GARCIA
JOHNSON
JONES
LEE
MILLER
O'MALLEY
PARDUCCI
SALGO
SIMS
SMITH
SMYTHE
THUNES
WALKER
WEST

Ready
```

PROGRAM 11-2 CREDIT FILE DATA RECORD SORT

This program sorts data records from the "CREDIT.DAT" data file. The user is given the choice of sorting by any of three data fields from the record. A "menu" displays the program options. There are seven subroutines in this modular progam. The functional outline of this program is shown below.

Main routine:
 a. print program title
 b. call input subroutine to input all data records from file
 c. print menu
 d. input choice
 e. call appropriate sort and/or print subroutine or end
 f. goto step c

Input subroutine (statement 1000)
 a. open "CREDIT.DAT" data file
 b. read data record (checking for sentinel)
 c. count number of records
 d. put data into arrays
 e. close data file
 f. return to main routine

Sort subroutines:
 All of the sort subroutines are identical with the exception of one statement; the comparison of data fields to be exchanged (the sort field). Each subroutine calls a nested subroutine to exchange the data items. Each subroutine also calls the print subroutine to output the data items.

 Subroutine 2000 - sort by customer name
 Subroutine 3000 - sort by account number
 Subroutine 4000 - sort by balance due (ascending)
 Subroutine 5000 - sort by balance due (descending)

Exchange subroutine (statement 6000)
 a. exchange adjacent names in name array
 b. exchange adjacent account numbers in account number array
 c. exchange adjacent balances in balance array
 d. return to appropriate sort subroutine

Print subroutine (statement 7000)
 a. print column headings
 b. print data arrays
 c. return to main program or calling subroutine

```
100 REM CREDIT FILE DATA RECORD SORT                    PROGRAM 11-2
110 REM
120 REM THIS PROGRAM SORTS DATA RECORDS FROM A DATA FILE. THE USER
130 REM IS GIVEN THE CHOICE OF SORTING BY ANY OF THREE FIELDS FROM
140 REM THE RECORD. A MENU DISPLAYS THE PROGRAM OPTIONS.
150 REM SUBROUTINES ARE USED TO KEEP THE PROGRAM MODULAR.
160 REM
170 REM   VARIABLES:
180 REM   N$=NAME ARRAY                  N=NUMBER OF RECORDS IN FILE
190 REM   A$=ACCOUNT NUMBER ARRAY        L=LIMIT FOR SORT ROUTINES
200 REM   B=BALANCE DUE ARRAY            I=LOOP INDEX
210 REM   C=MENU CHOICE                  J=LOOP INDEX
220 REM   T$=TEMPORARY FIELD FOR EXCHANGE ROUTINE
230 REM   T=TEMPORARY FIELD FOR EXCHANGE ROUTINE
240 REM   X$=TEMPORARY FIELD FOR ACCOUNT NUMBER
250 REM   Y$=TEMPORARY FIELD FOR NAME
260 REM   Z=TEMPORARY FIELD FOR BALANCE DUE
270 REM   X=TEMPORARY FIELD FOR CREDIT LIMIT
280 REM
300 REM MAIN ROUTINE
320 PRINT "DATA RECORD SORT PROGRAM"
330 PRINT
340 GOSUB 1000        ! READ DATA RECORDS FROM FILE
350 PRINT "PROGRAM OPTIONS:"
360 PRINT "1-SORT AND PRINT BY CUSTOMER NAME"
370 PRINT "2-SORT AND PRINT BY ACCOUNT NUMBER"
380 PRINT "3-SORT AND PRINT BY BALANCE DUE (ASCENDING)"
390 PRINT "4-SORT AND PRINT BY BALANCE DUE (DESCENDING)"
400 PRINT "5-PRINT IN CURRENT ORDER"
410 PRINT "6-END"
420 PRINT
430 INPUT "YOUR CHOICE";C
440 IF C<1 OR C>6 THEN 350                  ! CHECK FOR VALID RANGE
450 IF C=6 THEN 9999                        ! GOTO END OF PROGRAM
460 ON C GOSUB 2000,3000,4000,5000,7000     ! CALL SUBROUTINE
470 GOTO 350                                ! GOTO MENU

1000 REM DATA FILE INPUT SUBROUTINE
1020 OPEN "CREDIT.DAT" FOR INPUT AS FILE #1        ! OPEN THE FILE
1030 N=0              ! INITIALIZE COUNTER
1040 INPUT #1,X$,Y$,X,Z                ! INPUT DATA RECORD
1050 IF X$="LAST RECORD" THEN 1110     ! CHECK FOR SENTINEL
1060 N=N+1            ! INCREMENT COUNTER
1070 A$(N)=X$         ! MOVE ACCT NO TO ACCT NO ARRAY
1080 N$(N)=Y$         ! MOVE NAME TO NAME ARRAY
1090 B(N)=Z           ! MOVE BALANCE TO BALANCE ARRAY
1100 GOTO 1040
1110 CLOSE #1
1120 RETURN
```

(continued)

```
2000 REM NAME SORT SUBROUTINE
2020 L=N                ! SET LOOP LIMIT
2030 FOR J= 1 TO L-1    ! START OUTER LOOP
2040   L = L-1          ! ADJUST LOOP LIMIT
2050   FOR I=1 TO L     ! START INNER LOOP
2060   IF N$(I)<=N$(I+1) THEN 2080     ! ALREADY IN ORDER?
3070   GOSUB 6000       ! CALL EXCHANGE SUBROUTINE
2080   NEXT I
2090 NEXT J
2100 GOSUB 7000            ! CALL PRINT SUBROUTINE
2110 RETURN

3000 REM ACCOUNT NUMBER SORT SUBROUTINE
3020 L=N                ! SET LOOP LIMIT
3030 FOR J=1 TO L-1     ! START OUTER LOOP
3040   L=L-1            ! ADJUST LOOP LIMIT
3050   FOR I=1 TO L     ! START INNER LOOP
3060   IF A$(I)<=A$(I+1) THEN 3080     ! ALREADY IN ORDER?
3070   GOSUB 6000       ! CALL EXCHANGE SUBROUTINE
3080   NEXT I
3090 NEXT J
3100 GOSUB 7000            ! CALL PRINT SUBROUTINE
3110 RETURN

4000 REM BALANCE DUE SORT (ASCENDING) SUBROUTINE
4020 L=N                ! SET LOOP LIMIT
4030 FOR J=1 TO L-1     ! START OUTER LOOP
4040   L=L-1            ! ADJUST LOOP LIMIT
4050   FOR I=1 TO L     ! START INNER LOOP
4060   IF B(I)<=B(I+1) THEN 4080       ! ALREADY IN ORDER?
4070   GOSUB 6000       ! CALL EXCHANGE SUBROUTINE
4080   NEXT I
4090 NEXT J
4100 GOSUB 7000            ! CALL PRINT SUBROUTINE
4110 RETURN

5000 REM BALANCE DUE SORT (DESCENDING) SUBROUTINE
5020 L=N                ! SET LOOP LIMIT
5030 FOR J=1 TO L-1     ! START OUTER LOOP
5040   L=L-1            ! ADJUST LOOP LIMIT
5050   FOR I=1 TO L     ! START INNER LOOP
5060   IF B(I)>=B(I+1) THEN 5080       ! ALREADY IN ORDER?
5070   GOSUB 6000       ! CALL EXCHANGE SUBROUTINE
5080   NEXT I
5090 NEXT J
5100 GOSUB 7000            ! CALL PRINT SUBROUTINE
5110 RETURN
```

(continued)

```
6000 REM EXCHANGE SUBROUTINE
6020 T$=N$(I)          ! MOVE FIRST NAME TO TEMPORARY FIELD
6030 N$(I)=N$(I+1)     ! MOVE SECOND NAME TO FIRST NAME FIELD
6040 N$(I+1)=T$        ! MOVE TEMP. FIELD TO SECOND NAME FIELD
6060 T$=A$(I)          ! MOVE FIRST ACCT NO TO TEMPORARY FIELD
6070 A$(I)=A$(I+1)     ! MOVE SECOND ACCT NO TO FIRST ACCT NO
6080 A$(I+1)=T$        ! MOVE TEMP. FIELD TO SECOND ACCT NO
6100 T=B(I)            ! MOVE FIRST BALANCE TO TEMPORARY FIELD
6110 B(I)=B(I+1)       ! MOVE SECOND BALANCE TO FIRST BALANCE
6120 B(I+1)=T          ! MOVE TEMP. FIELD TO SECOND BALANCE
6140 RETURN

7000 REM PRINT SUBROUTINE
7020 PRINT "ACCOUNT","CUSTOMER","BALANCE"
7030 PRINT "NUMBER","NAME","DUE"
7040 FOR I= 1 TO N
7050   PRINT A$(I),N$(I),B(I)          ! PRINT DATA FIELDS
7060 NEXT I
7070 PRINT
7080 RETURN
9999 END
```

```
RUNNH

DATA RECORD SORT PROGRAM

PROGRAM OPTIONS:
1-SORT AND PRINT BY CUSTOMER NAME
2-SORT AND PRINT BY ACCOUNT NUMBER
3-SORT AND PRINT BY BALANCE DUE (ASCENDING ORDER)
4-SORT AND PRINT BY BALANCE DUE (DESCENDING ORDER)
5-PRINT IN CURRENT ORDER
6-END

YOUR CHOICE? 1
ACCOUNT        CUSTOMER       BALANCE
NUMBER         NAME           DUE
G-400          JOHNSON         200
G-300          JONES           300
D-325          MILLER          257
R-341          O'MALLEY        100
D-199          WEST            125

PROGRAM OPTIONS:
1-SORT AND PRINT BY CUSTOMER NAME
2-SORT AND PRINT BY ACCOUNT NUMBER
3-SORT AND PRINT BY BALANCE DUE (ASCENDING ORDER)
4-SORT AND PRINT BY BALANCE DUE (DESCENDING ORDER)
5-PRINT IN CURRENT ORDER
6-END

YOUR CHOICE? 4
ACCOUNT        CUSTOMER       BALANCE
NUMBER         NAME           DUE
G-300          JONES           300
D-325          MILLER          257
G-400          JOHNSON         200
D-199          WEST            125
R-341          O'MALLEY        100

PROGRAM OPTIONS:
1-SORT AND PRINT BY CUSTOMER NAME
2-SORT AND PRINT BY ACCOUNT NUMBER
3-SORT AND PRINT BY BALANCE DUE (ASCENDING ORDER)
4-SORT AND PRINT BY BALANCE DUE (DESCENDING ORDER)
5-PRINT IN CURRENT ORDER
6-END

YOUR CHOICE? 6

Ready
```

PROGRAM 11-3 INVENTORY TABLE SEARCH

This program demonstrates a sequential table search for inventory item numbers. Three 1-dimensional arrays are used; I$ for item numbers, D$ for descriptions, and P for prices. The program starts by reading data into these arrays (from DATA statements). Next, the user is asked to input a specific item number (into variable N$). After checking for the sentinel value "END", the program proceeds to search for a matching item number in the I$ array (using a FOR-NEXT loop to scan through the array). If and when a match is found, the program jumps out of the loop, displays the description and price associated with the item number, inputs a certain quantity, calculates the extended cost, and outputs the extended cost. If no matching value is found in the item number array, the program displays a message and transfers back to the item number input section.

```
100 REM INVENTORY TABLE SEARCH                          PROGRAM 11-3
110 REM
120 REM THIS PROGRAM SEARCHES A TABLE OF INVENTORY PART NUMBERS
130 REM IN ORDER TO MATCH AN INPUT VALUE. IT THEN PRINTS THE
140 REM ASSOCIATED ITEM DESCRIPTION AND PRICE. NEXT, THE USER
150 REM ENTERS THE QUANTITY ORDERED. THE PROGRAM CALCULATES AND
160 REM PRINTS THE EXTENDED COST.
170 REM  VARIABLES:
180 REM  I$=ITEM NUMBER ARRAY          I=LOOP INDEX
190 REM  D$=ITEM DESCRIPTION ARRAY     N$=ITEM NUMBER
200 REM  P=PRICE ARRAY                 Q=QUANTITY ORDERED
210 REM  N=NUMBER OF ITEMS             E=EXTENDED COST
220 REM
300 PRINT "INVENTORY TABLE SEARCH PROGRAM"
305 PRINT
310 DIM I$(20),D$(20),P(20)
320 READ N                             ! INPUT NO OF DATA ITEMS
330 FOR I = 1 TO N
340   READ I$(I),D$(I),P(I)            ! INPUT DATA ITEMS
350 NEXT I
360 INPUT "ITEM NUMBER (OR END)";N$    ! INPUT SEARCH VALUE
370 IF N$="END" THEN 999
380 FOR I= 1 TO N                      ! SEQUENTIAL SEARCH LOOP
390   IF N$ = I$(I) THEN 500           ! JUMP OUT IF MATCH IS FOUND
400 NEXT I
410 PRINT "ITEM NUMBER NOT FOUND"      ! NO MATCH FOUND
420 GOTO 360
500 PRINT "DESCRIPTION: ";D$(I)        ! OUTPUT DESCRIPTION
510 PRINT "PRICE:";P(I)                ! OUTPUT PRICE
520 INPUT "QUANTITY ORDERED";Q         ! INPUT QUANTITY
530 E = Q * P(I)                       ! COMPUTE EXTENDED COST
540 PRINT "EXTENDED COST:";E           ! OUTPUT EXTENDED COST
550 PRINT
560 GOTO 360
```

(continued)

```
600 DATA 10
610 DATA "A-100","LIGHT BULBS",1.50
620 DATA "A-200","EXTENSION CORD",5.95
630 DATA "A-300","3-WAY PLAY",.52
640 DATA "A-400","VOLTAGE CONVERTER",13.95
650 DATA "A-500","DIMMER SWITCH",6.49
660 DATA "B-100","PEN & PENCIL SET",3.95
670 DATA "B-200","STAPLER",5.25
680 DATA "B-300","PAPER WEIGHT",2.95
690 DATA "B-400","CALCULATOR",11.95
700 DATA "B-500","CLIP BOARD",2.75
999 END

RUNNH

INVENTORY TABLE SEARCH PROGRAM

ITEM NUMBER (OR END)? A-500
DESCRIPTION: DIMMER SWITCH
PRICE: 6.49
QUANTITY ORDERED? 2
EXTENDED COST: 12.98

ITEM NUMBER (OR END)? B-400
DESCRIPTION: CALCULATOR
PRICE: 11.95
QUANTITY ORDERED? 4
EXTENDED COST: 47.80

ITEM NUMBER (OR END)? C-100
ITEM NUMBER NOT FOUND
ITEM NUMBER (OR END)? A-100
DESCRIPTION: LIGHT BULBS
PRICE: 1.50
QUANTITY ORDERED? 5
EXTENDED COST: 7.50

ITEM NUMBER (OR END)? END

Ready
```

CHAPTER 12

FUNCTIONS

A. LIBRARY FUNCTIONS

Certain pre-written instructions, called library functions, are included as part of the BASIC language. These instructions automatically perform mathematical and string operations that are often needed by programmers. The functions are evoked, or called out, by simple keywords and associated parameters. The keywords are also reserved words, that is, they are an intrincate part of the BASIC language and can be used only for their stated purpose. Thus none of these words can be used for variable names. The general form of most library functions is

function(argument)

where "function" is one of the reserved words that specifies the operation to be performed and "argument" is the data value operated on. The "value of the function" is the result of the operation. This value can then be assigned to a variable or become part of a larger mathematical or string expression. The argument can be a numeric or string constant, a variable (with an assigned value), or an expression to be evaluated.

The following examples demonstrate the use of the function SQR(X). This function "returns a value of" (is evaluated as) the square root of the argument X.

EXAMPLE: Argument as a constant

```
10 Y = SQR(9)
20 PRINT "THE SQUARE ROOT = ";Y
30 END
RUNNH

THE SQUARE ROOT = 3

Ready
```

EXAMPLE: Argument as a variable

```
10 X = 9
20 Y = SQR(X)
30 PRINT "THE SQUARE ROOT = ";Y
40 END
RUNNH

THE SQUARE ROOT = 3

Ready
```

EXAMPLE: Argument as an expression

```
10 READ A,B,C
20 DATA 2,4,1
30 Y = SQR(A*B + C)
40 PRINT "THE SQUARE ROOT = ";Y
50 END
RUNNH

THE SQUARE ROOT = 3

Ready
```

EXAMPLE: Function as part of a larger expression

```
10 READ A,B,C
20 DATA 2,4,16
30 Y = A*B + SQR(C)
40 PRINT "CALCULATION RESULT = ";Y
50 END
RUNNH

CALCULATION RESULT = 12

Ready
```

1. Mathematical Functions

The most important mathematical library functions are listed in the table below:

Function	Value Returned	
SIN(X)	Sine of X	(X = angle in radians)
COS(X)	Cosine of X	(")
TAN(X)	Tangent of X	(")
ATN(X)	Arctangent of X	(Returns angle in radians)
EXP(X)	e^X	(e=Base of natural logarithms)
LOG(X)	Natural Log of X	
SGN(X)	-1 if X<0, 0 if X=0, 1 if X>0	
SQR(X)	Square root of X	
RND(X)	Random number between 0 and 1	(X=dummy argument)
ABS(X)	Absolute value of X	

The trigonometric functions (SIN,COS,TAN) require an angle measured in radians. One radian is approximately 57.3 degrees. If an angle is known in degrees it must be divided by 57.3 to be presented as a proper argument to these functions. The ATN function returns a value in radians. This must be multiplied by 57.3 to obtain a value in degrees.

The following program demonstrates the major mathematical library functions:

```
100 REM MATHEMATICAL LIBRARY FUNCTIONS PROGRAM
110 REM
200 PRINT,"MATHEMATICAL LIBRARY FUNCTIONS": PRINT
220 X=75                                  ! ANGLE IN DEGREES
230 R=X/57.3                              ! ANGLE IN RADIANS
240 PRINT "THE SINE OF";X;"DEGREES=";SIN(R)
250 PRINT "THE COSINE OF";X;"DEGREES=";COS(R)
260 PRINT "THE TANGENT OF";X;"DEGREES=";TAN(R)
270 Y=TAN(R):PRINT "THE ARCTANGENT OF";Y;"=";ATN(Y)*57.3;"DEGREES"
300 PRINT
310 PRINT "THE EXPONENTIAL E SQUARED =";EXP(2)
320 Y=EXP(2):PRINT "THE NATURAL LOGARITHM OF";Y;"=";LOG(Y)
330 PRINT
345 FOR I = 1 TO 3
350 READ X : S=SGN(X)
365 PRINT "THE SIGN OF";X;"IS";
370 IF S < 0 THEN PRINT "NEGATIVE" ELSE
    IF S = 0 THEN PRINT "ZERO" ELSE PRINT "POSITIVE"
380 DATA -5,0,10
390 NEXT I
400 PRINT
410 PRINT "THE SQUARE ROOT OF 9 =";SQR(9)
420 PRINT
430 PRINT "A RANDOM NUMBER BETWEEN 0 AND 1 =";RND(0)
440 PRINT
450 PRINT "THE ABSOLUTE VALUE OF -27 =";ABS(-27)
460 PRINT
470 PRINT "THE SQUARE ROOT OF 9 =";SQR(9)
999 END
RUNNH
```

```
              MATHEMATICAL LIBRARY FUNCTIONS

THE SINE OF 75 DEGREES = .965901
THE COSINE OF 75 DEGREES = .258912
THE TANGENT OF 75 DEGREES = 3.73061
THE ARCTANGENT OF 3.73061 = 75 DEGREES

THE EXPONENTIAL E SQUARED = 7.38906
THE NATURAL LOGARITHM OF 7.38906 = 2

THE SIGN OF -5 IS NEGATIVE
THE SIGN OF 0 IS ZERO
THE SIGN OF 10 IS POSITIVE

THE SQAURE ROOT OF 9 = 3

A RANDOM NUMBER BETWEEN 0 AND 1 IS .682034

THE ABSOLUTE VALUE OF -27 = 27

Ready
```

2. String Functions

The string library functions enable the programmer to easily extract a group of characters (a substring) from within a larger string or to search for certain characters with a string. Just as with the mathematical functions, a value is returned by the string functions and this value can be assigned to a new string variable. The most important string functions are listed below:

Function	Value Returned
LEN(X$)	The length of (# of characters in) string X$
LEFT(X$,n)	The leftmost n characters from string X$
RIGHT(X$,n)	The substring of X$ from the "nth" character to the right-most character
MID(X$,n1,n2)	The substring of X$ that begins at the n1 character position and is n2 characters in length
INSTR(n,X$,Y$)	The starting character position of substring Y$ within string X$ (0 returned if not found).

The following program demonstrates the major string functions:

```
100 REM STRING LIBRARY FUNCTION PROGRAM
110 REM
200 PRINT ,"STRING LIBRARY FUNCTIONS":PRINT
210 X$ = "GOLDEN GATE BRIDGE"
220 PRINT "THE STRING TO BE ANALYZED IS: ";X$
230 PRINT "THE LENGTH OF THE STRING = ";LEN(X$);"CHARACTERS"
240 PRINT "THE LEFT 4 CHARACTERS ARE: ";LEFT(X$,4)
250 PRINT "THE SUBSTRING FROM CHARACTER 14 TO THE END IS: ";
    RIGHT(X$,14)
260 PRINT "THE 4-CHARACTER SUBSTRING STARTING AT POSITION 8 IS: "
    MID(X$,8,4)
270 PRINT "THE SUBSTRING -GATE- STARTS AT POSITION NUMBER";
    INSTR(1,X$,"GATE")
999 END
RUNNH

              STRING LIBRARY FUNCTIONS

THE STRING TO BE ANALYZED IS: GOLDEN GATE BRIDGE
THE LENGTH OF THE STRING = 18 CHARACTERS
THE LEFT 4 CHARACTERS ARE: GOLD
THE SUBSTRING FROM CHARACTER 14 TO THE END IS: RIDGE
THE 4-CHARACTER SUBSTRING STARTING AT POSITION 8 IS: GATE
THE SUBSTRING -GATE- STARTS AT POSITION NUMBER 8

Ready
```

B. USER-DEFINED FUNCTIONS

Programmers can define their own functions for operations that are perfomed several times within a program and for which no library function exists. The function created only applies to the program in which it appears, but it still can save a great deal of space in the program and in the processor memory.

The DEF statement is used to define a user function. The general form is:

line # DEF FNa(X) = expression

line # = BASIC line number

DEF = mandatory reserved word

FNa = the function name. The letters FN are mandatory, and "a" is any BASIC variable name.

(X) = The argument in parenthesis is called a dummy variable, which can be any legal numeric or string variable name. This variable is used to define the "expression" part of the function.

expression= Any BASIC expression, including any legitimate mathematical or string operators.

Several examples will demonstrate user-defined functions:

EXAMPLE: Converting from centigrade to fahrenheit degrees

```
100 REM USER DEFINED FUNCTION PROGRAM
100 PRINT " TEMPERATURE CONVERSION"
110 DEF FNT(X) = (9/5)*X+32
120 PRINT:INPUT "ENTER CENTIGRADE DEGREES";C
130 IF C=-9999 THEN 999
150 PRINT "FAHRENHEIT DEGREES = ";FNT(C): GOTO 120
999 END
RUNNH

ENTER CENTIGRADE DEGREES? 0
FAHRENHEIT DEGREES = 32

ENTER CENTIGRADE DEGREES? 200
FAHRENHEIT DEGREES = 392

ENTER CENTIGRADE DEGREES? -9999

Ready
```

EXAMPLE: Converting a retail store cost to a selling price that has a 20% markup over cost.

```
100 REM RETAIL MARKUP
110 PRINT,"20% MARKUP OVER COST"
120 DEF FNS2(X) = X + .2*X
130 PRINT
140 INPUT "ENTER DOLLAR COST";C
150 IF C = 0 THEN 999
160 PRINT "SELLING PRICE = $";FNS2(C): GOTO 130
999 END
RUNNH
```

```
                20% MARKUP OVER COST

ENTER DOLLAR COST? 50
SELLING PRICE = $ 60

ENTER DOLLAR COST? 100
SELLING PRICE = $ 120

ENTER DOLLAR COST? 0

Ready
```

EXAMPLE: A social security number will be entered without hyphens The user-defined string function will insert hyphens between the 3rd and 4th numbers and between the 5th and 6th numbers.

```
100 REM INSERT HYPHENS IN SOCIAL SECURITY NUMBER
110 PRINT ,"STRING INSERTION FUNCTION"
120 DEF FNR$=LEFT(X$,3) + "-" + MID(X$,5,2) + "-" +
              RIGHT(X$,8)
130 PRINT
140 INPUT "ENTER SOCIAL SECURITY NUMBER";X$
150 IF X$="Z" THEN 999
160 PRINT "SSN WITH HYPENS = ";FNR$(X$)
170 GOTO 130
999 END
RUNNH
```

```
                STRING INSERTION FUNCTION

ENTER SOCIAL SECURITY NUMBER? 123456789
SSN WITH HYPHENS = 123-45-6789

ENTER SOCIAL SECURITY NUMBER? 222334444
SSN WITH HYPHENS = 222-33-4444

ENTER SOCIAL SECURITY NUMBER? Z

Ready
```

APPENDIX I BASIC STATEMENT SUMMARY

A. EXECUTABLE STATEMENTS

ASSIGNMENT

```
line # LET variable = <expression>
line # variable = <expression>
```

INPUT

```
line # INPUT list of variables
line # READ list of variables
line # INPUT #n list of variables
```

OUTPUT

```
line # PRINT list of variables
line # PRINT USING output format string, list of variables
line # PRINT #n, list of variables
```

DECISION

```
line # IF relation THEN instruction ELSE instruction
                        or transfer      or transfer
```

TRANSFER

```
line # GOTO line number
line # ON variables GOTO list of line numbers
line # ON variables GOSUB list of line numbers
```

PREDEFINED PROCESS

```
line # FOR Index = Start value TO End value STEP Increment
          .
line # NEXT Index

line # GOSUB subroutine line number
          .
line # RETURN
```

FILE CONTROL

```
                                 INPUT
line # OPEN "filespec" FOR       OUTPUT AS FILE #filenumber
line # CLOSE #filenumberd
```

PROGRAM TERMINATION

```
line # END
line # STOP
```

B. NONEXECUTABLE STATEMENTS

SPECIFICATION

```
line # DIM list of subscripted variables
```

DATA STORAGE

```
line # DATA list of constants
```

DOCUMENTATION

```
line # REM Comment
line # (BASIC Statement) ! Comment
```

APPENDIX II MICROCOMPUTER SYSTEM COMMAND SUMMARY

A. IBM PERSONAL COMPUTER

Naming and Saving Programs

NEW name - Assigns a label of "name" to the work area. Clears work area.

SAVE filename - A copy of the program in the work area is saved on the disk catalog.

LOAD filename - Moves a copy of program "filename" from the disk catalog into the work area

Listing and Executing Programs

LIST - Displays all program lines currently in work area.

LIST n-m - Displays program lines n through m.

RUN - Executes program in work area.

Deleting Programs

DEL filename - Deletes file "filename" from disk catalog.

Disk Catalog

DIR - Displays all filenames currently in the user's disk catalog.

Control Keys

CONTROL-BREAK Key - Terminates program listing-in-progress or program execution-in-progress. Work area is unaffected.

CONTROL-NUM LOCK Key - Halts program listing-in-progress or program execution-in-progress. Press any key to continue

B. APPLE II COMPUTER

Naming and Saving Programs

NEW name - Assigns a label of "name" to the work area. Clears work area.

SAVE filename - A copy of the program in the work area is saved on the disk catalog.

LOAD filename - Moves a copy of program "filename" from the disk catalog into the work area

Listing and Executing Programs

LIST - Displays all program lines currently in work area.

LIST n-m - Displays program lines n through m.

RUN - Executes program in work area.

Deleting Programs

DELETE filename- Deletes file "filename" from disk catalog.

Disk Catalog

CAT - Displays all filenames currently in the user's disk catalog.

Control Keys

CONTROL-C - Terminates program listing-in-progress or program execution-in-progress. Work area is unaffected.

CONTROL-RESET - Halts program listing-in-progress or program execution-in-progress. Press CONT to resume.

C. RADIO SHACK TRS-80 MODEL III

Naming and Saving Programs

NEW name - Assigns a label of "name" to the work area. Clears work area.

SAVE filename - A copy of the program in the work area is saved on the disk catalog.

LOAD filename - Moves a copy of program "filename" from the disk catalog into the work area

Listing and Executing Programs

LIST - Displays all program lines currently in work area.

LIST n-m - Displays program lines n through m.

RUN - Executes program in work area.

Deleting Programs

KILL filename - Deletes file "filename" from disk catalog.

Disk Catalog

DIR - Displays all filenames currently in the user's disk catalog.

Control Keys

BREAK - Terminates program listing-in-progress or program execution-in-progress. Work area is unaffected.

SHIFT-@ - Halts program listing-in-progress or program execution-in-progress. Press any key to continue.

D. COMMODORE VIC 64 COMPUTER

Naming and Saving Programs

NEW name - Assigns a label of "name" to the work area. Clears work area.

SAVE filename - A copy of the program in the work area is saved on the disk catalog.

LOAD filename - Moves a copy of program "filename" from the disk catalog into the work area

Listing and Executing Programs

LIST - Displays all program lines currently in work area.

LIST n-m - Displays program lines n through m.

RUN - Executes program in work area.

Deleting Programs

DELETE filename - Deletes file "filename" from disk catalog.

Disk Catalog

LOAD "$", disk no - Displays all filenames currently in the
LIST user's disk catalog.

Control Keys

RUN/STOP - Terminates program listing-in-progress or program execution-in-progress. Work area is unaffected.

BASIC PROGRAMMING

With Applications In Business

Bryce A. Martens
Ronald J. Tortorelli

Kendall/Hunt Publishing Company

B 403523 01
ISBN 0-8403-3523-7