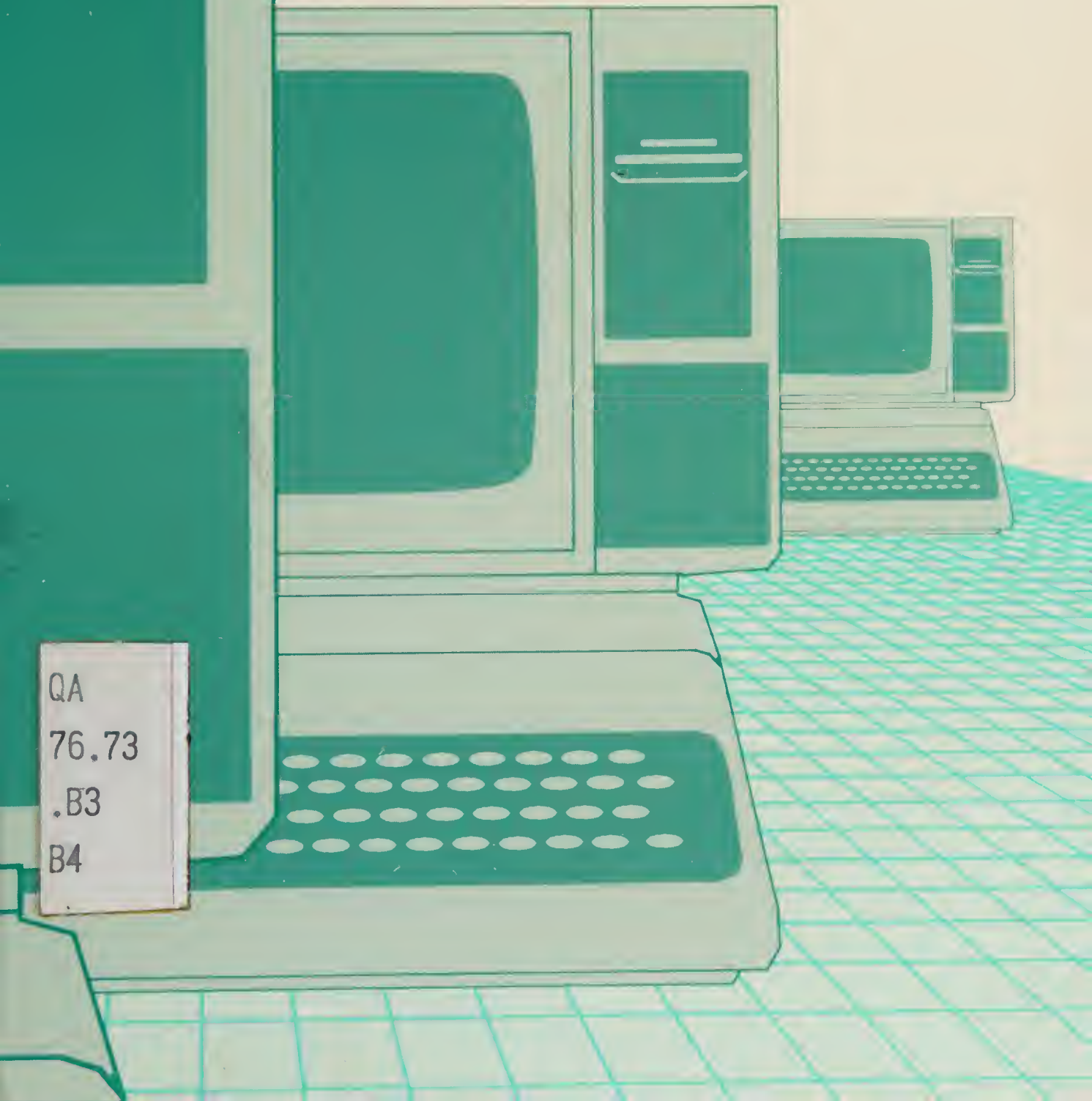
PROGRAMMING IN BASIC
the first steps

Robert G. Bell



PROGRAMMING IN BASIC
the first steps

Robert G. Bell

PROGRAMMING IN BASIC
the first steps

Robert G. Bell
Durham College

Prentice-Hall Canada Inc., Scarborough, Ontario

Canadian Cataloguing in Publication Data

Bell, Robert G. (Robert George), 1942-
Programming in BASIC: the first steps

Includes index.
ISBN 0-13-729830-7

1. Basic (Computer program language). 2. Programming (Electronic computers). I. Title.

QA76.73.B3B4 001.64'24 C83-098436-4

Prentice-Hall, Inc., Englewood Cliffs, New Jersey
Prentice-Hall International, Inc., London
Prentice-Hall of Australia, Pty., Ltd., Sydney
Prentice-Hall of India Pvt., Ltd., New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia (Pte.) Ltd., Singapore
Editora Prentice-Hall do Brasil Ltda., Rio de Janeiro

ISBN 0-13-729830-7

Production Editor: Charles Macli
Designer: Steven Boyle
Production: Monika Heike
Typesetting: Howarth and Smith Ltd.

Printed and bound in Canada by John Deyell Co.

1 2 3 4 5 JD 88 87 86 85 84

This book is dedicated to my wife Carole
and to my children Leanne and Rob.

Contents

PART II

Acknowledgements

This book is the end result of several years of "thinking about it". I've been mulling it over since about my third year of teaching.

It is also the result of several people's efforts and patience. My wife Carole has encouraged me over the years "to sit down and write the thing" and she rearranged our family schedule on more than one occasion to accommodate my writing. She also spent many, many hours proofreading the manuscript, testing the programs in it, and generally helping me to keep myself half organized.

Terry Woo of Prentice-Hall offered encouragement and assistance, particularly in the early stages of the book.

I would like to thank a number of my colleagues for their comments and ideas. Specifically Jim Anderson, Ron Neun, John Mather, Mary Perkins, Don Hargest, and Len Edwards.

I would also like to thank the staff of the college who helped me to master the intricacies of word-processing machines, which I used for a great deal of the preparation of this textbook.

Dale Jewett patiently put up with my countless questions and poor memory, and Linda Dillon, Lorrette Shermet, and Brenda Jackson also gave of their time and expertise in assisting me over the months that I've been involved in preparing this text.

Preface

Intent

The intent of this book is to introduce the beginner to the world of programming. As such it covers only the simplest concepts of the BASIC programming language. I have always felt that many texts cover too much too fast and have tried to avoid that error.

Format

The format of the book is such that it can be used as a self-study text or in conjunction with a course in programming.

It departs from the structure of most BASIC programming texts, since the same example is used in successive chapters. In each chapter, a new concept or two is introduced, which expands the example that has been discussed in the previous chapter. Only theory necessary for the understanding of the new concepts is presented. Assignments associated with a given chapter involve most of the concepts covered earlier in the text as well as those introduced in the latest chapter.

Do you have a computer?

The assumption is made that you have access to a computer or a computer terminal of some sort so that you will be able to apply the new concepts.

After all, programming is a skill and to be learned it must be practised. You cannot learn how to program a computer simply by reading about it. Indeed, you'll find that you will learn much more readily when you are using the computer.

My approach

I have intentionally simplified some aspects of the language and perhaps even implied limitations which are not in fact present. This is in order to concen-

trate on the basics of programming, in the hope of sparing the beginner some of the complexities involved.

As your proficiency in programming increases you will be capable of extending yourself beyond the scope of this beginner's text.

I have included the topic of structured programming, though not in great detail.

Wordiness

You may find some of my explanations wordy. I would ask the indulgence of those of you who feel this way since there are many novices who find detailed explanations helpful.

Depending upon your background and your aptitude for programming, you may require a more or less detailed explanation of a new principle than someone else. A key factor which prompted the writing of detailed explanations is the number of students who have remarked to me that reference texts in the library are "too vague" or "leave too much out" or that "they weren't much help".

It's rather ironic, I think, that it is only when students no longer need the explanation that they are able to understand many textbooks.

Many individuals require more detailed explanations than an instructor has time for. Others simply need more time to absorb a new principle or concept. If a detailed explanation is not available somewhere, the student is left to flounder and experiment, the result being utter confusion and frustration.

If you don't require detailed explanations, simply skim the material to the next example or section.

You will also find that later chapters are less wordy than earlier ones since the groundwork is by then firmly established, and more concise explanations suffice.

Business oriented examples

This text will not illustrate all of the features that your computer is capable of. Only the manufacturer's manual will do that. What it will hopefully do, however, is teach you the basics of what your computer can do, and provide you with enough knowledge to make it possible for you to pick up the manufacturer's manual and use it with a higher degree of understanding than you would otherwise be able to do.

The text uses simple business examples to illustrate concepts of BASIC. The principles can easily be extended to non-business applications, of course.

A blessing or a curse?

There are literally dozens of computer manufacturers in the marketplace today. Although there are some similarities among their computers, there are also many differences.

This tremendous variety of models and sub-models of computers is a blessing from the point of view of offering users a wide choice in machines; however, it is a curse for an author attempting to write a textbook that can be applied by as many computer users as possible.

In an attempt to address this problem, I have divided this book into two parts. Part I covers those aspects of BASIC which are very similar for all computers. Part II includes features which vary from one computer to the next.

I hope that you will find one or both parts useful for your purposes.

PART I

1 Setting the Stage

In this opening chapter we will examine the programming environment. We will look at what a computer is, what a program is, define a few terms, and mention some ideas that will assist you on your journey into the realm of programming.

New Vocabulary

BASIC	MICROCOMPUTER
COMPUTER	MINICOMPUTER
CPU	OUTPUT
CRT	PRINTER
DISK	PROCESSING
HARD/SOFT COPY	PROGRAM
INPUT	SYNTAX
INSTRUCTION	TAPE
MAINFRAME COMPUTERS	WORKSPACE

1-1. The Book Itself

This text is intended for self-study purposes or as a course book in introductory programming, and uses the BASIC programming language. BASIC is probably the most popular choice for teaching programming principles since it is one of the simplest languages and is employed on most of the microcomputers currently in the marketplace; e.g., Radio-Shack TRS-80, Commodore PET, Apple, etc.

The text is non-technical and non-mathematical to enable the reader to concentrate on learning the techniques of programming without having to review mathematical principles or theories.

Only the basic capabilities of the language are included and the extent of mathematics used is simple addition, subtraction, multiplication, and division. More advanced concepts and techniques can be found in other texts.

This text can be used in conjunction with virtually any computer that uses the BASIC language. Since only the fundamental characteristics of the language are studied, all programs, samples, etc. used in the text can be expected to work on the particular computer that you are using.

The text is written to allow the reader to learn the language in progressive steps. Each chapter introduces a new concept or two, building on the previous chapters. As such, the text should be studied from beginning to end, unless you are already somewhat familiar with BASIC or another computer language.

Embellishments are kept to a minimum to keep the text reasonably concise and hopefully less confusing.

An attempt has been made to include tables and summaries in such a manner as to allow the text to be helpful as a reference as well.

1-2. Opening Remarks

For most of us the concepts involved in programming are completely foreign to anything we have experienced elsewhere. The discipline is replete with new terminology, ideas, and machines. We can be forgiven, then, for a certain feeling of inadequacy or helplessness as we enter this new realm.

Experience has shown that this sense of frustration seems to afflict us all to a greater or lesser extent. Only previous experience in the computing field, or perhaps experience with another computer language, seems to be of any significant help in this regard.

It is also important to remember that no matter how well explained a new concept is or how many examples are given, a certain amount of trial and error, and hence time, seems to be a key factor in the learning process. For this reason, I recommend that you study the material in the text in relatively small portions. Attempting to cover too much too quickly often leads to confusion and frustration.

As well, you will find that you can save yourself hours of work and frustration if you have someone you can ask questions of: an instructor, fellow student, programmer friend, etc. Most of us cannot manage it strictly on our own - there are just too many things to sort out without a little help from someone.

1-3. The "Avalanche"

In studying programming we encounter an avalanche of new rules and terms, one which you may feel you will never get out from under. Those of you who persevere, however, will be rewarded by the acquisition of a new skill, which will surely be of benefit to you in the future.

Plan on reading much of this text over two or three times before you gain a high degree of understanding. To assist yourself further in this regard, you might highlight important passages, make notes in the margin, or on the inside front cover as you study BASIC.

1-4. What is a Computer?

In the broadest sense, a computer can be said to be anything that computes. This definition would include a calculator, an adding machine, an abacus, etc.

Normally, however, we are more restrictive in our definition, and limit it to a device which can store a program (see Section 1-7), and follow its instructions, which may be very complex indeed!

"Computers" in video games, televisions, microwave ovens, some toys, etc., are not computers by our definition; rather they are "microprocessors" - a term which is beyond the scope of this text.

1-5. Types of Computers

Computers may be classified by several means but the classification most often used is size. Generally there are four categories: MAINFRAME COMPUTERS, MINICOMPUTERS, MICROCOMPUTERS, and POCKET COMPUTERS.

MAINFRAME COMPUTERS have been in use commercially since the early 1950s. They come in various models with varying capabilities.

Mainframe computers consist of several devices which are attached together and housed in one or more locations. They may be contained in a single room or spread over a wide area geographically and connected via some communication link such as telephone lines or even satellites.

They range in cost from several thousand dollars for the smallest ones to several million dollars for the largest.

Virtually all large businesses use mainframes, but almost no small business could afford one, or would have reason to acquire one.

MINICOMPUTERS (or simply MINIS) arrived on the scene in the 1960s and, as their name implies, they can be considered as smaller versions of the mainframe computers.

Generally, and that's an important "generally", they are not as fast as the mainframe computers, nor can they store as much data, nor are they as capable

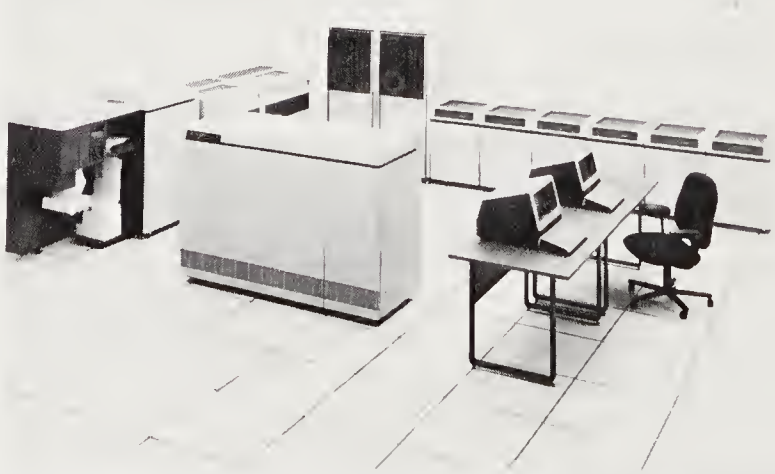


Fig. 1.1 A Hewlett-Packard 3000 Series 64 Computer System
(Photo: Courtesy of Hewlett-Packard (Canada) Ltd.)

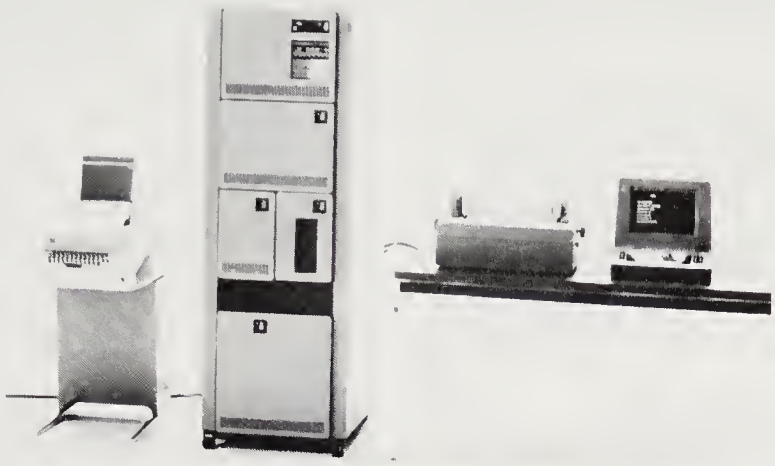


Fig. 1.2 An IBM Series 1 Minicomputer
(Photo: Courtesy of International Business Machines)

overall. However, minis are much less expensive than mainframes and they have opened the world of computing to areas which were previously unable to afford the high cost of the larger computers.

Minicomputers typically cost between $20 000 and $200 000 but some of them are classed as SUPERMINIS and these can cost considerably more.

Physically, minis are about the size of an office desk, though they may consist of several machines connected together. They are used by businesses of all sizes.

Two of these systems are illustrated in Figures 1-1 and 1-2.

MICROCOMPUTERS are a recent entry into the computer world. They were introduced in the mid-seventies and are offered in both the business and consumer markets. Those for sale to consumers are generally less capable than those offered to the business world, but they are also much less expensive. Microcomputers are also referred to as PERSONAL COMPUTERS, DESKTOP COMPUTERS or simply as MICROS. In the home market they are often called HOME COMPUTERS. Popular makes include Radio-Shack's TRS-80, Commodore's PET, and Apple's APPLE II.

Micros are about the size of a portable television set and may cost anywhere from a few hundred to several thousand dollars. Most of them consist of a single integrated unit or, at most, a "main unit" and one or two other devices. Very small businesses may use this size of computer as their only computer; larger firms would have some micros as well as some minis and/or mainframe computers.

Illustrations of microcomputers are shown in Figures 1-3, 1-4, and 1-5.

POCKET COMPUTERS have recently been added to the array of computers in the marketplace. They are compact enough to fit into a pocket (although not too small a one), and are sometimes called "hand-held computers". Pocket

computers do not have the capabilities of the larger ones of course, but they do allow programming and may have provision for attaching small printers or other devices to them. Some also include small, built-in display screens.

A number of computers are also referred to as "portable computers". These may be pocket computers or ones of a size somewhere between microcomputers and pocket computers. Classification is becoming more and more difficult it seems.

How long will it be until we have "wrist-computers"? How about "ring-computers"? What would come after that? "Pin-head" computers, perhaps. The actual limitation for size is our ability to manipulate the keyboard, controls, etc. Human eyes, ears, and hands become design limitations for engineers.

You could be using any size of computer to do your programming on. Almost all computers are capable of understanding the BASIC language.

An interesting point concerning sizes of computers is that as powerful as the large computers are, a given large computer may not be able to perform all the functions that a given mini or micro can. This is due to the fact that any given machine may or may not have certain options on it. This can be considered analogous to comparing a small car which might have cruise control and remote-controlled mirrors to a large car which does not have them.

It should be emphasized at this point too that, although some computers very definitely fit into one or the other of these categories, there are instances where it is not quite so obvious. For example, a certain large microcomputer could in fact be considered a small minicomputer. Or a powerful minicomputer might be every bit as capable as a smaller mainframe computer. At the "edges" of the categories things are definitely blurry.

Fig. 1.3 A Radio Shack TRS-80 Microcomputer
(Photo: Courtesy of Radio Shack, a division of Tandy Electronics Ltd.)

Fig. 1.4 A Commodore PET Microcomputer

(Photo: Courtesy of Commodore Business Machines)

Fig. 1.5 An Apple //e Microcomputer

(Photo: Courtesy of Apple Computer Inc.)

Incompatibility An annoying and frustrating fact with this dazzling array of computers is that, for the most part, they are not compatible with each other.

What this means is that if you type a program for one manufacturer's computer and try to run it on another manufacturer's machine, you will probably find that you can't. At least not without a bit of manipulation. It may be necessary to change a few (to several) lines of the program, especially if it is a complicated one.

The differences between your computer and another manufacturer's may be slight or considerable. Most often the differences will mean that a straightforward exchange of programs from one system to another is not possible.

The computer industry is so young and volatile that standardization has been a difficult goal. Although there are many accepted standards in the computer industry today, there is still a long way to go.

1-6. Can I Hurt the Computer?

Beginning programmers often feel that they may accidentally destroy valuable information or otherwise harm the workings of the computer. This is very unlikely. In fact, it is almost impossible for you to cause it any real harm, unless you decide to get violent. (You may at times feel like doing harm to the computer, but please contain yourself: after all, it's just a machine.)

If you are unsure which button or key to press and no one nearby seems to be able to offer any help, try something yourself. You won't harm the machine. At worst, you'll cause yourself some extra work by losing what you have already entered into the computer. Most often, you will simply receive an error message if you happen to have guessed incorrectly.

The key components of the computer system are well protected from our initial fumbling and exploring. The designers have allowed for us in their design of the computer system. So, be bold - "Press that key!"

1-7. Programming

It seems to be a fairly common misconception that in order to have a computer solve a problem for us, we can simply talk to it and tell it to solve our problem. As yet this is not possible. Maybe we'll be able to at some time in the future, but until that time arrives, we must write a series of simple instructions which the computer is able to decode and act upon. This set of instructions is called a PROGRAM.

Simply put, a program is a series of INSTRUCTIONS, written in a special language, through which we tell the computer what we want it to do; e.g., read in data for some employees (name, hours worked, and hourly rate of pay), calculate their pay using that data, and print out their names and pay.

Programming Languages A program is written in a PROGRAMMING LANGUAGE. You may have heard of some of the languages that are used with computers. They include COBOL, FORTRAN, RPG, Assembler, Pascal, PL/1, ALGOL, APL, and of course the one we'll be using, BASIC. There are literally hundreds of these languages but the nine just mentioned probably account for over 90% of all programs in existence. Most of the other languages are for specialized uses.

Different languages have different strengths or advantages. One may be powerful in its mathematical ability, another in its efficient use of the computer's memory, and a third may be good at handling files of information (e.g., a customer file).

BASIC, the one that we are studying, is a good introductory language. It is simpler than most of the others, and yet it is quite powerful as well. (You may only be convinced of it's simplicity after you've studied another language or two.)

Language Rules and Considerations Each programming language has a set of rules which govern its use. In a similar manner to English or any other human language, it has a certain vocabulary, special grammar rules (called SYNTAX), and some punctuation.

You will find, too, that the computer is every bit as demanding as the strictest grammar teacher you ever had. If you forget a comma that should be in-

cluded, or add one where there shouldn't be one, the computer will tell you of your error.

Incredibly Fast Idiot You must keep in mind that, in programming, we are giving instructions to a machine. It is very powerful and impressive but it's a machine, nonetheless. As such, we must be very specific and precise. It cannot read our minds as humans seem able to do at times, nor can it very accurately guess what we have in mind (though the designers of the computer have made it seem capable of this at times). The computer after all, as has often been said, can be considered to be an incredibly fast idiot. We, as humans, are still left with the burden of all the thinking. The computer will do our calculations for us, and it will do them correctly, as long as we give it the proper instructions.

There's a rather well-known acronym in the computer field that covers the above point well. The acronym is GIGO, which stands for the perhaps inelegant expression "Garbage In Garbage Out". In other words, if we type in the wrong data or the wrong instructions, then our results will be wrong as well.

The History of BASIC The original BASIC was developed in the mid-sixties by Doctors Kemeny and Kurtz at Dartmouth College. BASIC is an acronym for Beginner's All-Purpose Symbolic Instruction Code. Today, there are many versions of BASIC (one estimate states that there are more than 200 variations), including such names as Microsoft BASIC, Extended BASIC, TRS-80 BASIC, DEC BASIC-Plus, Tiny BASIC, Commercial BASIC, Integer BASIC, PET BASIC, TI BASIC, etc.

Although these variations differ from one another, you can think of them as dialects within the BASIC language. For most of them, there are more similarities than differences. They have unique names, however, because they were designed for specific computers by the manufacturers, and may offer some features that the others do not. The main differences tend to be in the more advanced features. A given version of BASIC may or may not have one of the more powerful functions. All will have the fundamental structure of the language, though.

I have attempted to limit this text to those features of BASIC which are common to all versions of the language.

1-8. The Basic Principle of Computing

Anytime we run a problem on a computer we are following the sequence shown in Figure 1-6. Something is fed into the computer (the INPUT), this is then PROCESSED (calculations performed, etc.), and finally something is produced as OUTPUT. In our case the output will always be something printed, either on a "TV screen" or on a printer.

In many business applications the input and/or output may be on magnetic tape, magnetic disk, or some other medium.

Fig. 1-6 The Basic Principle of Computing

1-9. What You'll be Using

The computer you are actually using may be a single device (self-contained), or it may be a terminal that is connected to a computer elsewhere. If it is a self-contained unit, it may include a built-in cassette tape or a floppy disk (see below). If it is a terminal, the computer it is attached to may be nearby or it could be many miles away.

Terminals If you are using a terminal, it will be one of two possible types:

a. CRT (Cathode Ray Tube). This device looks like a TV set except that it has a keyboard much like a typewriter's. The keyboard is used to send information to the computer and the screen is used to see what the computer sends us. (We are often called the USER, by the way, since we use the equipment.)

Almost all microcomputers use a CRT. The picture tube of your home television set is also a CRT and may be so employed by most home computers on the market.

Other names for this device include VISUAL DISPLAY UNIT (VDU), VIDEO DISPLAY TERMINAL (VDT), VIDEO UNIT, etc. CRT is probably the most commonly used term, however.

b. TELETYPEWRITER (usually just called a PRINTER). This device looks very similar to an electric typewriter. It has a keyboard much like a typewriter's and it also uses paper except that the paper is in one long ribbon and is called CONTINUOUS FORM.

Both of these devices operate in essentially the same manner. The only significant difference is that the printer provides us with a permanent record of what we do, while the CRT does not.

The fact that the printer output is on paper has led to the idea of referring to this kind of output as HARD COPY. The output from the CRT on the other hand cannot be saved, nor even touched really; hence, this form of output is referred to as SOFT COPY.

Often, as we are perfecting our work, we do not wish to keep any permanent record of it, in which case the CRT device is perfectly adequate. At other times, we do wish to have a record to take back to our desk or home (or maybe hand in to an instructor), in which case the output must be from a printer.

An advantage of using CRTs is that they are very quiet to operate and usually very easy to read. For these reasons, many people prefer to do as much of their work as is possible on a CRT. Later, it is possible to transfer your work to

a printer, assuming your computer has one. (By the way, this can be done without the need of retyping your program.)

Disk or Tape Storage Most computers have the capability of storing programs on MAGNETIC DISK or MAGNETIC TAPE for later use.

Larger computers very often use HARD DISKS for storage of programs. These are made of metal (usually aluminum) and coated with a magnetizable material. They may cost a few hundred dollars or as much as a couple of thousand dollars each. In appearance, they look like several large record albums stacked one above the other.

If a microcomputer uses disk storage, it will generally employ what are called FLOPPY DISKS (also called DISKETTES). They consist of a small round disk (the size of a 45-rpm record or smaller), held inside a square protective envelope, which is inserted into a "disk drive" for use by the computer. Floppy disks are made of a special plastic material and are flexible, hence the name. Do not bend them! It could easily result in the loss of data stored on them.

Disks of any type offer the advantage of being faster than tapes for retrieving programs or information, but they are also more expensive.

If tape is used it may be the "open reel" type on larger computers or tape cassettes for smaller computers. Magnetic tape cassettes for computers are very similar to those used for recording voice or music in the home. In fact, many people with home computers use their audio cassettes for their computer.

All tapes and/or disks must be handled with great care to protect the data that they contain.

If your computer does not have a tape or disk attached, when you turn off the computer you will lose your program. (This is the same thing that happens when you turn off most pocket calculators too.) If that happens, you must retype the program the next time you wish to use it: no problem for a short program, but a considerable amount of work if you are engaged on a long program.

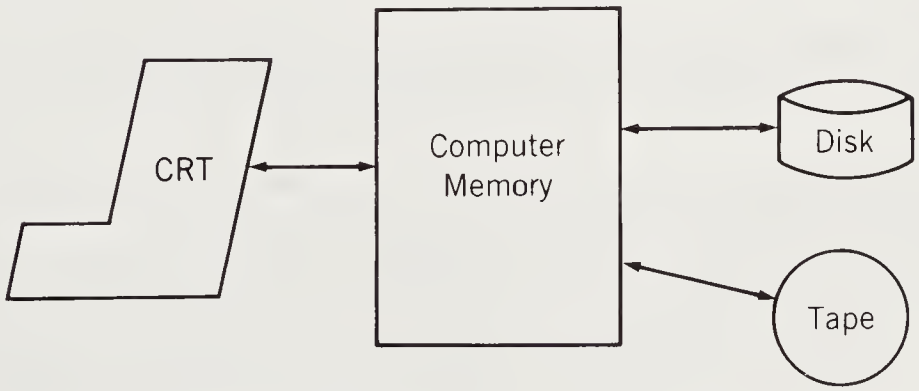


Fig. 1-7 What You'll be Using

1-10. Your WORKSPACE (or WORK AREA)

The above term is often used to describe that portion of a computer's memory that you are using to work in; that is, to do your programming in. The computer assigns a workspace to you automatically. On a larger computer your workspace will only be a portion of its total memory. On a microcomputer it may be the entire available memory.

The memory of a computer is part of what is called the CPU (Central Processing Unit). This is the key component of a computer. It contains the electronic circuits that are used to analyze and interpret your program, perform calculations, move data to the screen, etc. It also includes the memory and it controls the overall operation of the computer system.

Since someone else may have been using your workspace, or perhaps you yourself were using it for a different task, it is important that you be able to "clear the slate" before you start something new. Otherwise, one program may become mixed up with another one.

There is a special command in BASIC that will let us do just that. It is called the NEW command (or possibly CLEAR or SCRATCH on your computer). We will see how it is used soon.

Important Note There can only be one program in your workspace at a time. If you have one in it and you retrieve a second one (from a disk or tape for instance) you will destroy the first one.

If you start to type a new program while another program is in your workspace, the two will be mixed up with each other.

Now that we have seen what tools we will be using in our work, we are ready to examine BASIC itself. The next chapter is the beginning of our journey into the world of *programming*, which is the key we use to unlock the power of the computer.

Summary

The most important trait to bring along with you on this journey is perseverance. You will experience some frustrating and discouraging sessions at the keyboard, but you will also have some very productive and fascinating ones as well.

The computer that you are using may be capable of running several PROGRAMMING LANGUAGES, but we will only be using BASIC.

We write a PROGRAM to tell the computer what we want done. The program consists of a series of BASIC STATEMENTS or INSTRUCTIONS. In our program we will provide the computer with some INPUT DATA, we will tell it how to PROCESS that data, and also how to print the OUTPUT (our report).

We may write our programs on a TERMINAL or on the computer itself (a microcomputer). If the terminal or computer uses a "TV-like" screen, it is called a CRT (Cathode Ray Tube). It will produce SOFT COPY output on its screen. If the terminal has paper as its output medium, it is called a PRINTER. The printer produces output on paper which is referred to as HARD COPY.

Programs can be saved over a period of time by storing them on magnetic disk or magnetic tape.

We write our programs in a WORKSPACE (or WORK AREA) in the computer's memory that is automatically assigned to us by the computer.

Review Quiz

Reread the chapter and then try the following quiz.

1. What do the following acronyms (abbreviations) stand for? Explain what they mean.
   a. CRT
   b. GIGO
2. What is HARD COPY? What type of device produces it?
3. Computers are often classified by size into four groupings. What are they?
4. What is a PROGRAM?
5. What are the three basic operations involved in computing? Explain each one or give an example.
6. What is a FLOPPY DISK and what is it used for?
7. What is meant by WORKSPACE?

2 The Journey Begins

In this chapter we will begin to write programs. They will be extremely simple at first, becoming more complex as we introduce more concepts. They will, however, form the foundation upon which the remainder of the text will build. It is critical that you master each topic presented before proceeding to the next one.

New Vocabulary	COMMAND	PRINT statement
	CURSOR	PRINT ZONES
	END statement	RETURN key
	INCREMENT	RUN command
	KEYBOARD	SCRATCH command
	LIST command	SIGN-ON
	NUMERICS	STATEMENT

2-1. The Keyboard/Screen

Look over the keyboard of your computer or terminal as you study this portion of the chapter. Although it is very similar to a typewriter keyboard, there are a few differences:

a. The CURSOR and SCREEN. If you are using a CRT, there will be a CURSOR on the screen. It may be a short dash or a block of light, and it may flash as well. The cursor shows you where the next character will print on the screen, when you strike a key.

Depending on the computer that you are using, the screen size and the amount of data it is capable of displaying at one time may vary. The number of lines it can display at a time (top to bottom) may be 16 or 24 or some other number. The length of a line also varies with different computers/CRTs. Larger computers usually use CRTs that display 24 lines with 80 print positions on each line. Smaller computers typically use CRTs which display 16 lines or less and have a line length of 22, 32, or 40 print positions. However, other combinations are certainly possible.

If you are using a printer there is no cursor; the position of the type-head indicates where the next character will print. The line length for printers is

commonly 132 print positions. Smaller printers may have much shorter lines of course, while a few larger ones have line lengths exceeding 132 positions.

b. Upper/Lower Case. On some computers there are no lower-case (small) letters. Look for a CAPS LOCK key. If there is one, yours probably has both upper and lower case. Pressing this key once will lock your keyboard in upper case (capitals). Pressing it again will lock it in lower case. Most computers will function properly with this key in either the upper or lower-case setting, but there are exceptions.

You will notice that some keys (non-alphabetic ones) have two symbols on them. At times you will wish to type one or the other of these. To type a symbol that is on the top of a given key (e.g., a question mark), simply hold down the SHIFT key as you press the key with the desired symbol on it. (Most computer keyboards have two identical SHIFT keys, located on each side of the bottom row of the keyboard for your convenience.) Without pressing the SHIFT key, the symbol on the bottom of a given key is typed.

c. Letter O vs Zero. Be very careful when you wish to type a zero or the letter "O". Even though humans can easily determine what is meant from context, computers cannot. For example, if we see 4508 we know a zero is intended, while if we see a word such as TOP we know that a letter is meant. A computer, however, is unable to distinguish between the two. You will notice on your keyboard that the zero is next to the "9", and the letter "O" is below it in the top line of letters. Be careful with these two characters! One of the most common errors with beginning programmers is typing a zero when the letter "O" was intended or vice-versa.

To make it a little easier for you, most computers print the two slightly differently. One or the other may have a slash through it or be shaped slightly differently (a little wider or more oblong). Check yours by typing a few of each and noting any differences. Ignore any error messages that may be printed when you try.

d. l vs I vs 1. Also note that on a computer the number 1 is different from the letter "I", which is different from the lower case "l" (small L). Again, you must be careful to press the correct key so that the computer will interpret your input correctly. Many typists are accustomed to using a lower case "l" for the number "one". This does *not* hold true for a computer.

e. Period vs Decimal Point. These are in fact the same symbol on a computer keyboard (usually in the lower right corner of the keyboard).

f. Spaces. The SPACE BAR is still used to provide spaces, exactly as it is on a typewriter.

g. RETURN key. The RETURN key (ENTER key on some machines) is very important and must be pressed after each line that you wish to send to the computer.

Beginning programmers will often type some input to the computer but forget to press the RETURN (ENTER) key when finished. Just remember that nothing is sent to the computer until you press it.

2-2. Signing-on

Before we can even begin to write programs, we must have the computer turned on and operating.

If you are using a microcomputer the procedure could be as simple as turning it on. You might automatically be in the BASIC programming mode and "ready to go".

Larger computers almost always require that a more formal SIGN-ON procedure be followed. This procedure is not especially difficult, but it may seem involved to you at first.

The complexity of the sign-on procedure depends on the particular computer that you are using. It often involves identifying yourself or an account number for billing purposes, and it may also require that you give the computer a secret password. This password is for security reasons to prevent unauthorized use of the computer.

Since the procedure varies considerably from computer to computer, it is important to check the operating manual of your computer or have the instructor show you how to sign on to your particular computer, so that it is set up to use the BASIC language.

Once you are signed on and ready, we can begin to program.

By the way, you will know when the computer is ready to accept input because it will tell you by printing a special word or symbol known as the SYSTEM PROMPT.

If it's a word, it might be "READY" or "OK". If it is a symbol, it could be a number sign (#) or maybe a "greater than" sign (>).

As soon as you have the prompt, you can begin to program in BASIC.

2-3. Signing-off

It is important that we also be able to sign-off the computer when we have finished our work. In the simplest case this could mean just turning the computer off. For another computer it might mean typing the word BYE or maybe OFF.

Check with your computer manual or your instructor to see what the procedure is for your computer.

Record the methods for signing on and off on the inside front cover or in Appendix F.

2-4. The First Step

The first step on our journey is to pose a small problem and examine a program that will solve it for us. The problem is simpler than any practical one would be, but remember that we are simply trying to learn the rudiments of programming and must begin at the beginning.

Our Problem Let's say that we have two numbers, 3333 and 6.00, that we want the computer to print out on the screen (or paper) for us. (All numbers

used in calculations are referred to as NUMERIC DATA or simply NUMERICS. Even numbers not used in calculations are often called numerics as well.)

Let's assume that the first numeric is an employee number and the second one is the hourly wage rate. This is not important as far as the computer is concerned but it is important to us, because we know what the numbers represent.

Since we want the computer to print out the two numbers for us, we must write a program to tell it to do so. The following program will accomplish what we want.

```
10 PRINT 3333,6.00
20 END
```

Important Notes

1. The 10 and 20 are LINE NUMBERS. Each line of a BASIC program *must* begin with a line number. Some computers provide these for us automatically, but most require that we type them in. Usually we count by tens so that we can insert lines if we wish to.

2. We will not include the dollar sign or the number sign since computers cannot perform calculations using numbers with dollar signs or number signs in them, and we will want to do that as we develop our program.

3. Spaces are not critical. (An exception to this is when they are between quotation marks but we'll look at that later.) For example, line 10 could have been written as:

```
10PRINT3333,6.00
```
or
```
10 PRINT 3333 , 6.00
```

The computer doesn't care. We include spaces to make our programs more readable to us or to other humans. You may find that if you do leave out the spaces (or add extras), the computer will insert (or delete) them for you.

The above program consists of two lines or STATEMENTS. It is very short but it's a program, nonetheless. We will analyze it in detail later to discover some rules of BASIC.

The next step is to type this program into the computer, but before we do, we should be aware of how to correct any typing errors we might make.

Make a Mistake? The following are useful correction techniques:

a. Use the BACKSPACE key. It is usually in the upper right corner of the keyboard. Simply backspace to the position where you made an error and retype the correct version. If you cannot find a BACKSPACE key, look for a "backwards arrow" (←). It accomplishes the same thing for us.

b. If nothing else seems to work you can always press RETURN (ENTER) and rekey the entire line. This is a little more work but you *can* correct your errors this way.

c. If you accidentally type in an incorrect line that you do not want, you can delete it. To delete a line that is present in a program, type the line number and press RETURN (ENTER); e.g.,

20 (and press RETURN or ENTER)

will delete line 20 from your program. You will be using this procedure frequently. (See Section 2-7 for more details.)

Type the program into the computer now. Remember, after each line that you type you must press RETURN (ENTER).

2-5. BASIC Commands

We now want the computer to print our "results" for us. This is accomplished by typing in the command RUN - and pressing RETURN (ENTER). COMMANDS are special words that we use to tell the computer to perform certain functions for us. For now we will only use the following three commands:

a. NEW (or CLEAR or SCRATCH) to "clear the slate" before we begin. This command is used to ensure that there is nothing in our workspace when we begin. If we do not use it, we may end up getting our program mixed up with an earlier program. We did not require this the first time we signed on to the computer but we will from now on. I will remind you to use it for the first few times.

b. LIST to show us what our program looks like; that is, this command will simply print out our program as it is stored in the computer. It may, however, appear in a slightly different form, since some computers reformat (rearrange) programs somewhat - perhaps adding/deleting spaces as mentioned earlier.

c. RUN to produce our program results for us. That is, it will give us whatever OUTPUT we have instructed it to. Output is the term we use to describe what our program prints out when we run it. In this case, it will be our report.

KEY POINT — TO TYPE COMMANDS DO NOT INCLUDE A LINE NUMBER. COMMANDS ARE ENTERED SIMPLY AS THE WORD ITSELF; FOR EXAMPLE, RUN or LIST.

Important Note Because we use the commands LIST and RUN so often, some computers today have special keys on their keyboards that can be used instead of typing out the words. If yours does, you need only press the RUN key or the LIST key when you wish either of these actions. It may even have a key for the word PRINT. Other computers allow certain abbreviations to be used for these words. Rather handy.

Using BASIC Commands After you have typed the last line of the program:

```
20 END
```

type the command RUN and press RETURN (ENTER). (Remember that since RUN is a command it does not need a line number.)

Thus, you will have typed:

```
10 PRINT 3333, 6.00
20 END
RUN
```

and the output should appear as

```
3333          6
```

Note that the decimal point and the zeroes probably were not printed out. Most computers consider that zeroes in front of or behind a number do not add anything to it, so they ignore them. Of course these same computers would not ignore zeroes within a number; e.g., 600.04 would be stored and printed as exactly that but 600.040000 would be treated as 600.04, as would 000600.04.

Now try listing the program. Type LIST (and press RETURN or ENTER) to see what your program looks like. Only the corrected, "clean" version will be printed; that is, any typing errors that you made and corrected will not be printed.

It is never mandatory that you list a program, but since you will want to do so often, make sure that you know how to do it. Note that the computer will probably drop the decimal point and zeroes as it did before when we ran the program.

Important Note Some computers allow us to use some of the ordinary BASIC *statements* as *commands*. If your computer has this ability, and you type certain BASIC statements without a line number, it will act immediately on the "STATEMENT COMMAND". For instance, if you type PRINT 3333,6.00 the computer will immediately print the values 3333 and 6.00. We do not want the computer to act upon our individual lines immediately though - we want to write the entire program and then have the computer run it for us "all at once".

If your computer does have this feature, forget about it for now and be sure to include a line number before each line of a program.

2-6. Programming Rules and Notes

As simple as the above program is, it does illustrate several concepts and rules of BASIC. Let's analyze the program to learn these rules.

```
10 PRINT 3333,6.00
20 END
```

Rule 1. Each LINE of a program (also called a STATEMENT or INSTRUCTION) must be numbered. The only requirement of these numbers is that they increase with each succeeding line. The increase is called an increment. Commonly we start at 10 and increase by 10 each time so that lines can later be inserted in the program if we wish to do so. (We will see how this is done later.)

Rule 2. The only things that can be included in a data value (e.g., 3333 and 6.00) are a decimal point (.) and/or a sign (+ or -). We must not include a $ or a # or any other symbol if we wish to use the number in calculations. Thus, if we typed the first line of our program as 10 PRINT #3333, $6.00, the computer would not have printed the numbers for us. Rather it would have generated an error message stating that we had done something wrong.

By the way, if we do not include a sign with our numbers (+ or -), the computer will assume that the number is positive (+). Most often, that is exactly what we want it to assume.

Rule 3. PRINT is one of the words that is in the computer's vocabulary. Because it is a very limited vocabulary by human standards, we cannot use the word WRITE instead of PRINT. We must use PRINT, and only PRINT.

The word PRINT is used to have the computer print information onto the screen or paper. Note that we told it *what* to print but we did not tell it *where* to print, or *how*. The computer decides the "where" and the "how" (see next chapter).

For now, just keep in mind that the computer divides each line on the screen/page into invisible ZONES. Think of these zones as invisible columns. The number of these zones per line, and their width, varies with different computers. Yours may have only two zones, or it could have four, five, or more per line.

When we tell the computer to print something out, it will automatically print out the first value in the first zone, the second value in the second zone, and so on. Each comma in the PRINT statement tells the computer to advance to the next print zone.

In our case it will print the value 3333 in the first zone and the value 6 in the second.

It will begin printing at the left-hand side of each zone using as many spaces as required.

The comma is very important since it shifts the printing to the next zone (column). See what happens when you leave the comma out: type line 10 as

```
10 PRINT 3333 6
```

and RUN the program again. You will notice that the output will appear as

```
33336
```

With no comma included, the computer treats 3333 6 as a single number: 33336. (The computer ignores spaces, remember, unless they are between quotation marks.)

Rule 4. The computer automatically starts at the first line of our program and after it has done what that line tells it to do, it will automatically go to the following line or lines, one after the other.

KEY POINT — UNLESS IT IS TOLD TO DO OTHERWISE, A COMPUTER ALWAYS FALLS TO THE NEXT LINE IN A PROGRAM.

Rule 5. The last line of our program is simply the word END. This is another word that is in the computer's vocabulary. This signals the computer that the program is complete. On many computers this statement is optional; however, it is a good idea to use it anyway.

2-7. Adding, Deleting, and Changing Lines

Memorize these three items — you will be using them constantly. Also make a note of them on the inside front cover or in Appendix F.

Adding Lines To add a line to an existing program, simply type the line number to be added, followed by the appropriate BASIC statement. The line number must not already exist or the current version of it will be destroyed.

Usually, we want to add a line between two existing lines. To do this, we simply choose a line number between the two. It is usually best to choose a number that is close to the middle of the gap; e.g., to add a line between lines 40 and 50, it would be best to select number 45: that way, more lines may be inserted later if necessary.

Try this: (Change the instructions slightly if you are using a printer.) Add line 15 to the program we have been using.

To do this, you can simply type the line now, wherever the cursor happens to be on the screen (assuming that it is on the left-hand side of the screen somewhere). It is not necessary to move the cursor "up" the screen.

If you type the line "where the cursor is now", the screen may look something like this:

```
10 PRINT 3333,6.00
20 END
RUN
3333          6
LIST
   10 PRINT 3333,6
   20 END
15 PRINT 2000
```

This is correct. The fact that line 15 appears on the screen below the other two (10 and 20) does not matter. Inside the computer, line 15 will be stored between lines 10 and 20 as it should be. The screen is just our "writing slate".

You can quickly confirm that the program is as it should be inside the com-

puter. LIST and/or RUN the program again. When you list it, you will find that the program appears as follows:

```
10 PRINT 3333,6
15 PRINT 2000
20 END
```

which is what we wanted.

RUN and LIST the program now if you haven't already.

KEY POINT — TO ADD A LINE TO A PROGRAM, SIMPLY TYPE THE LINE NUMBER THAT YOU WISH TO ADD, WITH THE APPROPRIATE BASIC STATEMENT.

Deleting Lines What if you make a mistake and type in a line that you realize you do not need? Or you find that you have typed a line in the wrong place?

KEY POINT — TO DELETE A LINE FROM A PROGRAM SIMPLY TYPE THE LINE NUMBER OF THE LINE THAT YOU WANT TO DELETE, AND PRESS RETURN (ENTER).

Delete line 15 and LIST and RUN the program once more; i.e., to delete line 15, simply type 15 and press RETURN (ENTER).

Changing Lines To change or replace an existing line with another line, simply type the new version of the line with the same line number. The computer will automatically erase the old version of the line when you give it the new one. It is not necessary to delete the old one first. Deleting a line is only necessary when there is not going to be a replacement for it.

Hence to change line 10 we might type

```
10 PRINT 56, 678
```

From then on this would be line 10 as far as the computer is concerned. It is not necessary to retype the remaining correct lines.

KEY POINT — TO CHANGE A LINE, SIMPLY RETYPE THE LINE WITH THE SAME LINE NUMBER. IT WILL REPLACE THE OLD VERSION OF THE LINE INSIDE THE COMPUTER.

Some computers allow changes to be made directly to a listing on the screen by moving the cursor to the appropriate error and correcting it. If yours does not, you must retype the entire line.

Important Note One problem we all seem to run into at some time or other is ending up with a line number 0 or a huge line number like 32033. They creep into our programs like gremlins when we are learning to program. Don't be too concerned if you do not understand how they got there. You can delete them the same way you delete any other line, by typing the line number and pressing RETURN (ENTER).

2-8. Skipping Print Zones

For those computers that have more than two print zones, there's a simple trick to make our output print in specific zones.

For example, if we wanted our two values to print in the second and fourth zones, we print blanks in the first and third zones.

Our print line would be

```
10 PRINT " ", 3333, " ", 6
```

The " " tells the computer to print a blank space in that particular zone. Thus, the computer "skips" it.

There must be ONE BLANK SPACE between the quotation marks.

Some computers allow us to omit the blank within quotation marks and just insert commas to skip zones. If yours does, you could type line 10 as

```
10 PRINT , 3333, , 6
```

The commas still tell the computer to advance to the next print zone.

If your computer only has two print zones the only choice you might have is whether or not to skip the first zone.

```
10 PRINT " ", 3333
```

would thus print the number 3333 in the second print zone. We will study the PRINT statement in detail in the next chapter.

You will find yourself constantly referring to the text and/or your notes in the early stages of learning to program. There is simply too much to remember at first. You might highlight passages in the text or write notes in the margins. Also, write notes on the inside front cover of the text as you progress through the material. Anything that helps you locate appropriate references quickly will be of benefit to you.

2-9. Idiosyncrasies of Computers

It is worth noting here that as simple as our early programs are, you may experience difficulties when attempting to enter them into the computer. As if you didn't have enough to be concerned about already, computers tend to exhibit certain traits we normally associate with humans.

Each type of computer seems to possess a number of idiosyncrasies that would be impossible to describe in a text. They are minor points that make the

computer unique. These idiosyncrasies can be very irritating to a beginner, but they are definitely a part of the business of programming.

To deal with them may require advice from a classmate or instructor as to what you should do. Occasionally, there is no one around to help you and you are forced to try something on your own. It may help to look over the material in the text once more. If you cannot see the answer there, you may have to "just try something".

Sometimes, you may struggle with some problem for a considerable length of time (some people have spent hours) only to discover that someone else is able to solve the problem for you in a matter of seconds. Almost all of us go through similar experiences - it seems to be a part of the initiation rites that allow us to enter the world of programming. Take heart in the fact that eventually you will find that you're able to remember the rules and procedures of BASIC.

Review the above material until you understand it thoroughly. Since we will build on previous material, chapter by chapter, it is vital that you understand as much of each topic as possible as we go along.

2-10. Scientific Notation (or E-notation)

Before we end this chapter type in and RUN the following program. Remember to clear your workspace first.

```
10 PRINT 3 444 000 000
20 END
```

(Spaces in the number are for clarity only; the computer will ignore them.)

The output may appear as follows:

```
3.444E+09
```

Look a little strange? This form of output for numbers is called SCIENTIFIC NOTATION. This is how the computer prints out very large and very small numbers.

Think of it as being in two parts - two numbers separated by an E (for EXPONENT). The first number (in our case 3.444) represents the basic number itself. The second part (+09) tells us how many positions we must move the decimal point in the basic number (3.444) to obtain the actual number being represented.

A plus sign (+) before the exponent indicates that the decimal point must be moved the required number of positions to the right, a minus sign (-) means the decimal point must be moved to the left.

Thus, 3.444E+09 represents 3 billion, 444 million which is the number we started with. Only the form has changed.

Some computers use this notation for all numbers over a million (1 000 000), others for those over a billion (1 000 000 000).

If you are curious, experiment with this a little on your computer to discover when it switches to scientific notation. Change line 10 in the program. Try the other direction too, if you are curious. That is, try it with very small numbers; e.g.,

```
10 PRINT 0.000 000 111 (spaces are for clarity only)
```

If any of the programs in this text produce output in scientific notation, it means that you have made a mistake somewhere. Check your program closely.

If you do have occasion to use large numbers in your programming, you may enter data using scientific notation; e.g., instead of typing:

```
10 DATA 4000000, 120500000
```

you could type it as:

```
10 DATA 4E6, 1.205E8
```

2-11. Limits of Your Computer

Computers do have limits. Your computer will not be able to accept values above a certain number. This number is so huge that only scientists are apt to find it a limiting factor.

Try something like this on your computer if you are curious: (Remember to clear your workspace first. Spaces in the number are for clarity only.)

```
10 PRINT 111 222 333 444 555 666 777 888 999 000 999 888
20 END
RUN
```

If this number does not give you an error message indicating that you are trying to use too large a number, you will get one at some point. Make the number large enough so that you get the error message. This error message takes the form of an "overflow message" meaning that you have "filled the cup" — and then some.

Numeric precision Another limit that computers have is the number of digits they can keep track of, for a given value.

For example one computer might be able to keep the exact value 3456.887766554433 in its memory, but another one might only be able to hold a maximum of 9 digits per value, meaning that it would record the above value as 3456.88777 (rounding the original value to the maximum number of positions that it is capable of storing).

Many computers have two levels of accuracy which we will call "normal precision" and "extended precision". Normal precision might be accurate to 9 "significant digits" (non-zero digits), and extended precision to 16 digits, for example. If your computer has two levels of precision, you will have the option of calling the extended level if you need it.

The CLEAR key Something you might have noted by now is a CLEAR key on the keyboard (perhaps marked CLR). This key only affects the screen of your computer or CRT, and not the copy of your program that is inside the computer's memory. Thus, when you press the CLEAR key, it seems to destroy your program (if it is on the screen), but in fact it is only the image on the screen that is gone. Your program is still safe inside the computer. You can RUN it, LIST it, etc., as you please.

Summary

This chapter has introduced some extremely important concepts and rules of the BASIC language. They form the very foundation of our work and as such must be mastered before you continue on to the next chapter.

We wrote our first program.

We found that the KEYBOARD of the computer is similar to that of a typewriter, but it does include a number of keys which perform special functions that are unique to computers.

It is very important that we distinguish between the letter "O" and the number zero on the keyboard, as well as between the number "1" and the letters "I" and "l". Remember each of these five has its own key. The computer requires that the correct one be pressed.

Remember too, that the RETURN key (or ENTER) must be pressed after each line that we type.

We have also learned that BASIC programs consist of a number of lines called STATEMENTS. These statements must be numbered, using an INCREMENT of 5, 10, etc., to allow additional statements to be inserted later if required.

We found that to print something on the screen or paper we use the PRINT statement, one of the words that is in the computer's vocabulary.

The computer prints data values in ZONES. These zones vary in size and number per line depending on the computer that you are using (more about zones in the next chapter).

Numbers that are used in calculations are called NUMERICS and they may not contain a $ or a #. The only permissible symbols other than the digits themselves are a sign (+ or -) and/or a decimal point.

COMMANDS are used to tell the computer what you want done. The NEW command (or CLEAR or SCRATCH) will clear our workspace so that we can start with a "clean slate". The LIST command will show us what our program looks like, and the RUN command will produce the output.

All programs should end with an END statement.

Lines can easily be added to a program or changed or deleted.

Review Quiz

1. Each line of a BASIC program is called a _____.
2. Explain the following terms :

a. PRINT ZONE
b. NUMERIC
c. COMMAND
d. STATEMENT
e. CURSOR

3. What will the following program do?

```
10 PRINT 678,0
20 END
```

4. What is a SIGN-ON procedure? When is it performed?
5. Why is the RETURN (ENTER) key so important?
6. What is an INCREMENT? What should we keep in mind regarding it when writing our programs?
7. What is the difference between the LIST command and the RUN command? Does it matter what order we do these in?
8. What does it mean if the computer prints out a value for a number in the following form:

```
3.4567E+09
```

9. How is a line added to a program?
10. How is a line deleted?
11. How is a line changed?

Exercises

I suggest that you get in the habit of writing your exercises down on paper first, then when you have them figured out as best you can, key them into the computer. That way, you will have a record on paper - if you only have a CRT. It's also a fact that most of us think better on paper.

Remember to clear your workspace before you type each program into the computer.

2-1. The Invoice Exercise

This exercise will be an "on-going" one. We will begin to use it here but, as the textbook progresses, we will add to it. It will grow in complexity as we acquire more knowledge of BASIC.
Begin the INVOICE EXERCISE as follows:
Write a program that will print out the following data. Ignore headings: just print out the numbers.

Input data:	(Item #)	(Price $)
	6217	10.00
	3424	69.95

(Remember that the $ and # symbols are not permitted in a program.)

2-2. Write a program to print out your age and year of birth.

2-3. How would the program from exercise 2-2 have to be changed so that it would print out both your age and year of birth as well as those of a friend?

2-4. It was stated that it is not permissible to include a dollar sign or a number sign in our data. Write a short program which does include them. Then try to RUN it, and make note of the error message(s) that you receive from the computer.

For example, you could enter the simple program

```
10 PRINT # 3333, $6.00
20 END
```

3 The PRINT Statement

The PRINT statement is one of the most frequently used and, hence, most important of the statements in BASIC. For this reason, it should be studied carefully. There are also a number of variations to the PRINT statement which increase its flexibility. We will examine them in this chapter.

New Vocabulary	JUSTIFICATION	PRINT USING STATEMENT
	PRINTER SPACING CHART	TAB function

3-1. The "Ordinary" PRINT Statement

Recall that in the last chapter we used the following program :

```
10 PRINT 3333 , 6
20 END
```

Clear your workspace and type it in again.

The form of the PRINT statement used in this program can be referred to as the "ordinary" or "standard" PRINT statement. It uses the word PRINT, followed by the data values that we wish to print, separated by commas. These commas tell the computer to separate the page or screen into invisible PRINT ZONES. You can think of a print zone as a column on the screen.

Recall that the number of these print zones and their size depends on the particular computer that you are using. Microcomputers usually have fewer and smaller zones than larger computers do, though not always. The determining factor is how many print positions there are on a line of your screen or paper. Commonly, there are 32, 40, or 80 print positions on a line of a CRT and 120 or 132 positions on a hard copy terminal (printer). Zone sizes will vary accordingly as will the number of zones per line.

For example, one microcomputer may have four zones of ten positions each, another may have only two zones of 16 positions each. Larger computers commonly have five zones. Some have five zones of which the first four are 15 positions and the last one is only 12. Ask your instructor, read the manual for

your computer or experiment a little (try the exercise below) to determine the number of zones on your computer. The programs in this text will use four zones at most.

Test Your Computer To determine how many zones your computer has, add the following line to your program temporarily:

```
1 PRINT 1,2,3,4,5
```

Now RUN your program. Each of the five numbers will print in a separate zone. If they all print on one line it means that your computer uses at least five zones per line. If the numbers 1 and 2 print on the first line, 3 and 4 print on the second line, and the 5 prints on the third line, it means that your computer only has two zones per line.

Of course, if the numbers 1 through 4 print on the first line and the number 5 prints on the second line, then your computer uses four zones per line.

Delete the line you added by typing the line number (1) and pressing RETURN (ENTER).

3-2. Printing of Names, Headings, Words

It is easy to print a person's name or a heading on the screen. We simply enclose the name or heading in *double* quotation marks; e.g.,

```
10 PRINT "ELLIOTT"
```

will print the name ELLIOTT on the screen or paper, starting at the left-hand side.

We could also easily combine a name and a number on a single line. The statement would then be

```
10 PRINT 3333, "ELLIOTT"
```

To print a pair of headings we could use

```
10 PRINT "NUMBER", "PAY RATE"
```

We will be looking at adding names and headings to our programs later. I mention it at this point mainly for those of you who are a little more adventurous than most or perhaps a little less patient.

Adding headings can lead to complications, however, and if you wish to try to enter one or the other of the above headings, please refer back to Section 2-7 if you cannot remember how lines are added to a program.

3-3. Alternatives to the Ordinary PRINT Statement

It is important that you remember how many zones your computer uses when you type your PRINT statements.

If your computer only has two zones to a line, it would be awkward at times reading the output from a program. For example, if the program was to print four items per line, it would have to print two of them on the first line and two on the next line. This would be very confusing to sort out, since different information (such as wage rate and hours) might be printing out in the same column of a report.

There is a solution to this dilemma. In fact, we have a choice of a few other forms of the PRINT statement which prove extremely valuable on a computer that only has two zones, but which are also invaluable at times on computers that have four or more zones.

Let's look at these alternate forms. The first one makes use of the TAB function.

The TAB Function The TAB function allows us to ignore the zones and treat the screen or paper as one continuous line. It is similar to the TAB key on a typewriter in that you are able to "jump" to specified positions on the line. Since we are only using two zones to this point, we have not really needed this function. In the next few chapters, though, we will be using up to four zones and, if your computer has only two, it will be much easier for you to do these programs if you understand and use the TAB function.

If we were to use the TAB function in our short program, it would appear something like this:

```
10 PRINT TAB(2); 3333; TAB(10); 6
20 END
```

When you RUN the program the output should be

3333bbbb6 (Each b represents one blank position on the screen/paper.)

The word TAB is another word that is in the computer's vocabulary. It tells the computer to space over to the *print position* indicated in parentheses after the word TAB, and print out the number that follows the TAB entry.

Thus, the example above tells the computer to move over to print position 2 and type 3333, *starting* in *print position 2*; and to print position 10, to print the value 6.

Try changing the TAB locations and rerunning the program; e.g., you might try

```
10 PRINT TAB(7); 3333; TAB(20); 6
```

Important Notes

1. Many computers leave the first position blank if a number is positive. (The numbers will appear to be starting to print one position late.)

2. Some computers number the print positions on the line beginning at zero, others start at one. To start at zero may seem confusing, I know, but computers are notorious

for starting to count at zero in other instances, too. Accept it as a necessary evil.

To sort out what yours does, you could enter and run this program: (Remember to clear your workspace first.)

```
10 PRINT "1234567890"
20 PRINT TAB (2); 1000
30 PRINT TAB (2); -2000
40 PRINT TAB (2); "HELLO"
50 END
```

The first line of the program labels the first ten print positions for you. Then line 20 shows you how your computer prints a positive number, line 30 shows how a negative number prints and line 40 illustrates how a piece of alphanumeric data prints (more about the term "alphanumeric" later).

If the minus sign of the -2000 prints under the 3, your computer starts counting at 0. If it prints under the 2, the computer starts counting at 1.

Another word of caution. If you are using a large computer, the TAB function may not work beyond print position 71. Check yours to see.

Several Values on a Line An example that prints four values on a line would be:

```
10 PRINT 88; TAB(8); 999; TAB(16); 2468; TAB(24); 678.9
```

Enter this line into your program and RUN it. The output should be something like this:

```
88      999     2468     678.9
```

Note that there is no TAB before the 88 so it begins printing in print position 1.

Printer Spacing Charts To make it easier to plan your output when using the TAB function, there are printed forms available which show individual print positions for a line of printed output. They are called PRINTER SPACING CHARTS or printer output sheets. If you do not have them, you can improvise with a sheet of ordinary paper. Simply rule it off in as many columns as there are print positions on a line of output for your computer. (Every 5 or 10 print positions may be enough.) Then, decide where on the line you want to print each value that you are including in your PRINT statement.

You will then be able to include the necessary TABs in your PRINT statement to accomplish this.

The Use of Semi-Colons A second means of ignoring print zones in order to have more than the usual number of values print on one line is through the use of semi-colons in the PRINT statement. These simply replace the commas between data values in the PRINT statement. Their effect is to print the values closer together on the line; e.g.,

```
10 PRINT 1;2;3;4;5
```

will print all five values on one line even if your computer only has two print zones.

```
10 PRINT 1,2,3,4,5
```

would space the output much further apart you will recall.

Although the semi-colon permits us to print several values on a line, it is not without it's own complexities.

The main one is that when more than one line of information is being printed, and values are of unequal length, they may not line up very well. For example, output may appear as follows:

```
1111      222.2          .333            4
1.11          .222            33333      44.4
1111      22222       .33333          4444.4
```

Although the output is readable, it is also rather confusing and can easily lead to errors when people read the results.

Nonetheless, the semi-colon is a valuable tool to have at our disposal and we will make use of it in the future.

It is possible to combine the TAB function with commas and/or semi-colons as well; e.g.,

```
10 PRINT 16, TAB(15); 26; 36
```

3-4. The "Trailing Semi-Colon or Comma"

If we inadvertently tack a semi-colon on the end of a PRINT statement, the effect is that the next PRINT statement will not start printing on a new line as we normally want it to. Instead, it will be "tacked on" to the last line that was printed. Sometimes, you may want to do just that but probably not very often.

Try the following example to see how this works: (Remember to clear your workspace first.)

```
10 PRINT 11 ; 12 ; 13
20 PRINT 21 ; 22 ; 23
30 END
```

RUN the program. The output might look like this:

```
11    12    13
21    22    23
```

Some computers will print the results as:

```
111213
212223
```

Now add a "trailing semi-colon" to line 10 so that it appears as:

```
10 PRINT 11; 12; 13;
```

RUN the program again. The output might now look like this:

```
11   12   13   21   22   23
```

or it may even appear as:

```
111213212223
```

(Again, the output varies, depending upon the particular computer that you are using.)

Another option available for spacing the above values would be as follows:

```
10 PRINT 11 ; " " ; 12 ; " " ; 13
```

The blanks between the quotation marks will separate the values more than they would otherwise be.

The above comments apply to trailing commas as well.

A common error of beginning programmers is adding a trailing comma or semi-colon where there should not be one.

3-5. The PRINT USING Statement

This is a third alternative to using the standard PRINT statement; however, it is only available on certain computers.

The PRINT USING statement is more complicated than any of the other methods but that is the price we pay for power - it will do more than the other forms, as well.

It will accomplish the following for us:

- **a.** Allow us to use the full width of the print line without regard to print zones.
- **b.** Line up columns of numeric data, automatically aligning it on the decimal point - just as humans do manually.
- **c.** Automatically "round" numeric values to whatever decimal position we wish.
- **d.** Print out dollar signs if we want them.

Many computers do not allow the use of the PRINT USING statement at all. Of those that do, the form of the statement varies from one computer to the next. For this reason I will not discuss it further here. If you wish more detail on it, see Chapter 17.

3-6. Justification of Output

(This only applies to the "ordinary" PRINT statement.)

A point worth mentioning now perhaps is that of JUSTIFICATION. Our output has been printing in what is known as a LEFT-JUSTIFIED format. This means

that for any given column, the data is aligned on the left-most digit. Since our numbers to date have been of equal length in a given column, this has not been of any concern to us. (The invoice exercise is an exception.) If the numbers 6666 and 23 were to be printed in the *same* column, however, they would appear like this:

```
6666
23
```

This is not as we are used to seeing them in everyday life. Normally figures are RIGHT-JUSTIFIED:

```
6666
  23
```

Left justification is something we must accept for now. (The PRINT USING statement mentioned earlier will right-justify output for us but this is not necessary. Let's keep things simple for now and stick to the ordinary PRINT statement.)

Summary

The computer divides the screen or page into invisible ZONES that it uses for printing. The number and length of these zones varies with different computers. There could be from two to five zones or more per line, and typically, they will each be from ten to twenty print positions wide.

To print words we enclose them within quotation marks.

If we wish to print more than the usual number of columns of data per line we can use the TAB function; e.g.,

```
40 PRINT TAB(5); 22; TAB(15); 33; TAB(22); 44
```

We can also use semi-colons in the PRINT statement:

```
40 PRINT 22; 33; 44
```

Remember that the TAB function is usually the better choice of the two for most programs.

Printed output is left-justified automatically; that is, it will be lined up on the left-hand side position of a column. This applies to both numeric and alphanumeric data.

Exercises

3-1. Write a program to print the following values on a single line. You will have to use the TAB function or semi-colons in the PRINT statement (for most computers).

```
2   4   5   6   10   11   22
```

3-2. As a diversion you might like to draw a picture; e.g., to draw a chair you could use:

```
10 PRINT "XX"
20 PRINT "XX"
30 PRINT "XX"
40 PRINT "XX"
50 PRINT "XXXXXXXXXX"
60 PRINT "XX        XX"
70 PRINT "XX        XX"
80 PRINT "XX        XX"
90 END
```

You can be much more elaborate of course if you are artistic. Try using different letters and symbols.

You have probably seen much more complicated drawings than this of anything from Snoopy to the Mona Lisa. They were programmed by someone with a great deal of time and patience.

3-3. The Invoice Exercise.
The input data is now as follows:

(item #)	(price)	(quantity sold)
6217	10.00	5
3424	69.95	100

Write a program that will print the above data on the screen (paper). If your computer uses only two print zones per line, it will be necessary to use the TAB function or semi-colons in the PRINT line.

4 The LET Statement
(The Assignment Statement)

This chapter introduces us to the extremely important concepts of variables and the assignment of values to them. These ideas are at the very heart of all programming and you will find that they offer us a great deal more flexibility in our work.

New Vocabulary LET statement VARIABLES

4-1. The Second Step

Although the method of Chapter 3 is an acceptable one to begin with, it is not very efficient if we wish to print out details of hundreds or thousands of employees.

Because of this we will begin to use VARIABLES in our programs at this point. The variables are the single letters in the program below (E and R).

Though this method is superior to the previous one, you may find it somewhat confusing at first. Persevere, remember.

Our sample program now appears as follows:

```
10 LET E = 3333
20 LET R = 6
30 PRINT E,R
40 END
```

Important Note In most versions of BASIC the word LET may be omitted. In some, the word LET must be omitted. Your results will be incorrect if you use it in these versions. I will omit it in future chapters.

Clear your workspace and type the above program in.

LIST it if you wish. By the way, it is never necessary to list a program. We do so simply to have a look at it when we wish to.

If we run this program on the computer (by typing the BASIC command RUN), it would print out the same result as before

```
3333      6
```

More work — same result. Why bother? Read on.

Notice that the words LET, PRINT, and END are used in this program. We have already used PRINT and END. LET is another of the words in the computer's vocabulary. There are several others as well but keep in mind that the vocabulary is quite limited and you must stick to the words that are in it.

BASIC statements are named by the key word contained in them. Thus, the above sample program includes two LET statements, a PRINT statement and an END statement.

Functions of the LET Statement The word LET can be used either to assign a value or perform a calculation. For now, we will only look at the first task.

The statement

```
10 LET E = 3333
```

assigns the value of 3333 to E. Think of E as the name of a "box" inside the computer's memory. The "boxes" will hold our values for us temporarily. These boxes are more properly called VARIABLES.

The computer stores values in these variables until we tell it what we want done with them. Using variables gives us more flexibility in our programming, as will become more apparent to you later.

If we examine our program above we see that the first line of it:

```
10 LET E = 3333
```

sets a variable called E to a value of 3333. The computer will set up a variable named E inside its memory to hold the data. Nothing appears on the screen as yet.

The computer will then read the next line of our program:

```
20 LET R = 6
```

and set up a second variable by the name of R and place a value of 6 in it.

Thus, inside the computer we have the following situation :

Variables:

E	R
3333	6

The PRINT Statement (with Variables) The computer will now go to the next line of our program.

```
30 PRINT E,R
```

This line tells the computer that we want it to print out the values that it has stored in the variables (boxes) E and R.

KEY POINT	THE COMPUTER HAS BEEN DESIGNED IN SUCH A WAY THAT IT WILL PRINT OUT THE CONTENTS OF THE VARIABLES, AND NOT THEIR NAMES.

Thus, when we say PRINT E,R it prints what is *in* those "boxes" and not the names of them.

As we mentioned before, it will always print the values starting in the left-most position of each zone.

Our output will look like this:

```
3333      6
```

By the way, a common error of beginning programmers is to type line 30 as:

```
30 PRINT "E", "R"
```

If you do, the output will appear as:

```
E     R
```

which is obviously not what we want.

Remember too that the number of values that can be printed on a line is limited to the number of zones that there are on a line. Their size and number vary from computer to computer. It could be as few as two or as many as five or more.

So far, so good?

Now, let's have a look at those variables we have been using.

VARIABLE (or "Box") Names Notice that we used single letters of the alphabet for variable names in our program — namely E and R. The rules for naming these variables are:

a. they can be a single letter of the alphabet; or

b. they can be a single letter followed by a single digit;
e.g., A3, T6, Q4, E9, etc.

Nothing else is permitted on many computers. Thus, the following would not be allowed:

8E Wrong order (E8 would be O.K.).

M# No special symbols are allowed.

QA Double letters are not allowed (by many computers).

L32 Only one digit is permitted after a letter (by many computers).

Any of the following would, however, be permissible:
J, B, Q, P4, X, N7, Z

Virtually all computers allow the above names for variables, so let's stick to these simpler rules for now.

As long as you follow these rules, you can use whatever names you wish for your variables. Just be consistent; e.g., we could have written our program as follows (the keyword LET is omitted in the examples):

```
10 N = 3333 (N to represent Number)
20 H = 6 (H to represent Hourly rate)
30 PRINT N,H
40 END
```

But, we could not have written it as follows:

```
10 E = 3333
20 R = 6
30 PRINT N,H
40 END
```

In the above case we are inconsistent. We set up our values in variables E and R, but asked the computer to print out the values that are in N and H. Many computers would not even run such a program for us, but would simply print out an error message. Other computers would assume that N and H are both zero since they had never heard of them before (in your program), and would print out two zeroes.

Enter the above program into your computer (clear your workspace first) and see what happens when you try to run it. Note the result.

KEY POINT — WHEN CHOOSING YOUR VARIABLE NAMES, KEEP THEM AS SHORT AS POSSIBLE (e.g., N rather than N1) AND TRY TO USE MEANINGFUL LETTERS AS VARIABLE NAMES.

An exception to using short names is if you have several names to keep track of. In that case, longer names are helpful, and will certainly make your program more understandable to others.

Normally, though, you are most likely to use a P to represent an individual's Pay, an E or an N to represent an Employee Number, etc. Select names that will be easy for you to remember. I may choose E for employee number, but you may feel that N is more logical.

Avoid using A and B or some such meaningless variable names. As programs become more complicated, this point will prove very helpful; but get into the habit even with short programs.

4-2. The Third Step

I would suggest that you look over the second step again before proceeding with the third.

What if we wanted to print out the details of *two* employees? Using the method we have just learned it would be necessary to write our program as follows:

```
10 E = 3333
20 R = 6
30 PRINT E,R
40 E = 4444
50 R = 5
60 PRINT E,R
70 END
```

In this case, the names of our "boxes" do not change, but the contents of them do. This is why the "boxes" are called VARIABLES. Think of these variables as "mailboxes". Like mailboxes, the names on the boxes do not change, but the contents do.

In this instance, we are printing out the details about two employees. When we run our program it will print out the two employees like this: (Remember the computer always "falls down" to the next line of the program automatically.)

```
3333      6
4444      5
```

KEY POINT — EACH TIME THE COMPUTER ENCOUNTERS THE WORD PRINT, IT BEGINS PRINTING ON A NEW LINE OF THE SCREEN OR PAPER (UNLESS THE PREVIOUS PRINT STATEMENT IN THE PROGRAM ENDED WITH A COMMA OR A SEMI-COLON IN WHICH CASE THE LINE IS TACKED ON TO THE LAST ONE).

You can see how we could print out the details for any number of employees using this method. It is a workable method, but not a very practical one for large numbers of employees.

The next step that we will take is one of the most important of our entire journey. It will be the method which we will use in almost all future programs.

Important Note At the risk of sounding repetitious, reread the previous few pages if you are at all unsure of any of the material we have just covered. It is essential that you understand new concepts as we go along since new material constantly builds on what has been discussed to date. Do not be in too much of a rush to press on.

Also, be sure to try all the programs that are given in the examples. It means a little more typing and digging out commands, etc., but that is exactly what you want at this stage of your learning. Lots of practice is the secret.

Summary

The LET statement is used to assign a value to a VARIABLE. The keyword LET is optional in most versions of BASIC; it is not allowed at all in some. Variable names must be a single letter or a single letter followed by a digit for many versions of BASIC.

We will see in a future chapter that LET statements are also used for calculations.

Exercises

4-1. Review questions:

a. When we type in a program, is it necessary to LIST it before we can RUN it?

b. What is a VARIABLE?

c. What are the rules for naming variables?

d. How do you decide what name to use for a variable?

4-2. Which of the following are valid variable names for your computer? For those that are not, indicate what the error is.

a. E4	b. 7G	c. K87
d. T	e. M#	f. B9
g. QW	h. K8F	i. Q

4-3. Write a program to assign the two values 8877 and 9.98 to variables and print them out on a line.

4-4. The Invoice Exercise

Use LET statements to enter the data to your program.

Input data:	Item #	Price($)	Quantity sold
	6217	10.00	5
	3424	69.95	100

4-5. We have three values 10, 4, and 500 and we want to print them out on the screen in all the possible sequences. Write a program that will accomplish this. Your output should look something like this:

```
10      4       500
10      500     4
4       10      500
4       500     10
500     4       10
500     10      4
```

Hint: Assign the three values to three variables first, using LET statements. Then use several PRINT statements to produce the various sequences.

5 READ and DATA Statements
(and the GOTO Statement)

In this chapter we begin to use the true power of the computer. We will look at looping and the BASIC statements that it requires.

New Vocabulary DATA statement, ENDLESS LOOPS, GOTO statement, LOOP, READ statement

---

5-1. The Fourth Step

Although the method of Chapter 4 does do what we want, you can see that it would be tedious to print out details for 200 employees. It would be necessary to add 600 more lines to the program. (Two LET statements and one PRINT statement for each of the 200 employees.) Imagine doing this for a large company with five thousand employees.

There is a better way. It involves the use of some new BASIC statements: the READ and DATA statements, and the GOTO statement (think of GOTO as the two words GO TO). Using these statements, our program would look like this:

```
10 DATA 3333,6 (1st set of data)
20 DATA 4444,10 (2nd set of data)
30 READ E,R
40    PRINT E,R (indented for clarity)
50 GOTO 30
60 END
```

This program forms the basis for examples to be used in the next few chapters. It will grow in complexity as the text progresses. Each chapter, we will incorporate any new concepts into this sample program.

Clear your workspace, type in the program and RUN it. Do *not* type in my comments in parentheses. The output should look like this:

```
3333     6
4444     10
(OUT OF DATA error message or equivalent)
```

The OUT OF DATA error message is to be expected at this stage — we will eliminate it in a later chapter. The actual wording of it varies from computer to computer. Make a note of the wording yours uses so that you will recognize it in the future.

Let's examine our program, and find out how the new statements are used.

5-2. The DATA Statement

The DATA statement is used to enter data into a program. It will hold our data until our program is ready to use it. It is an alternate method to using LET statements.

Each DATA statement requires a line number, the word DATA and our individual data values listed after it, separated by commas.

Let's get into the habit of including the particulars of a single person, item, customer, etc., in a DATA statement by itself. We will refer to the data values for an individual person, item, etc., as a SET of data; i.e., each DATA statement will contain one set of data.

KEY POINT — THERE SHOULD NOT BE ANY COMMAS WITHIN THE DATA VALUES THEMSELVES.

For example, 4444 is forty-four hundred and forty-four, but 4,444 would be interpreted by the computer as two numbers: four, and four hundred and forty-four.

It is also important that the data be in the correct order, since the computer will pick up the data in the same sequence as it is on the DATA statement(s).

Though it may seem strange at first, the DATA statements themselves may be placed anywhere in the program. The only important thing is that they be in the order you wish the computer to process them. It is recommended that you place them near the top or bottom of your program, out of the way to some extent.

5-3. The READ Statement

The READ statement is used to actually read your data into the computer's memory. The computer will look at the READ statement and note how many variable names there are in it (in our case there are the two: E and R), then "look at" the DATA statement(s) and read as many data values as necessary to fill these variables.

READ and DATA statements go together like salt and pepper. You do not find one without the other. When the computer encounters a READ statement,

it knows automatically that it must read the data from a DATA statement, so it searches for one.

Remember that we are going to ensure that the number of data values in each DATA statement will be the same as the number of variable names in the corresponding READ statement.

Explanation of READ and DATA Statements One of the first things that the computer does with our program is to take all of our DATA statements and write them in one long string in a special area of its memory. (This isn't exactly what it does, but it will be helpful if we think of it that way for now.)

This is why the DATA statements can be placed anywhere in the program – the computer will locate them and set up its data area before it starts the actual "work" involved in your program.

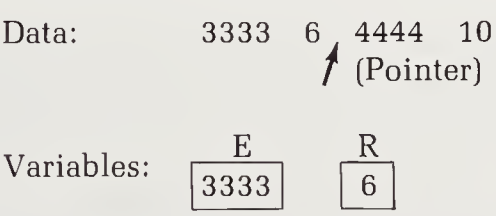
It is our responsibility to ensure that the data is in the correct order when we key in our program. In our case, the first values it will read are the value 3333 into E and the value 6 into R when it comes to line 30 in our program

```
30 READ E , R
```

E is the first variable in the READ statement and 3333 is the first data value in the DATA statement, so the 3333 is placed in E. Likewise, the second data value (6) is placed in the second variable (R).

The computer will place a POINTER in the data string at this point, so that it can remember where it left off. It is like our keeping a finger on a spot so we know where to return to. Thus the pointer will be after the 6 in the DATA statement after it has performed the READ statement the first time.

Inside the computer memory the situation is:



The computer then goes to the next line of our program

```
40 PRINT E , R
```

and sees that it must print out the two values that are in E and R. It will print them in zones 1 and 2 remember.

```
3333      6
```

Let's repeat the program for easier reference.

```
10 DATA 3333,6 (1st set of data)
20 DATA 4444,10 (2nd set of data)
30 READ E,R
40     PRINT E,R
50 GOTO 30
60 END
```

5-4. The GOTO Statement

Next the computer falls to line 50 and finds a "GOTO statement"

```
50 GOTO 30
```

(The GOTO may be entered on most computers as GO TO if you prefer. Some computers will print it on the screen as GOTO in any case; others leave it as GO TO.)

The GOTO statement will change the normal sequence of a program by sending it to another line in the program. The computer has no choice — it must move or "branch" to the statement number given. In programming terminology, the GOTO statement is referred to as an UNCONDITIONAL BRANCH statement.

The line number we wish to send the computer to is simply placed after the "word" GOTO. In our case, the GOTO statement sends the program back to line 30 and, thus, enables us to reuse lines of our program. Otherwise we would have had to rewrite the lines as we did before. (See Section 4-2 of Chapter 4.)

This "going back" in the program is referred to as LOOPING back. The portion of our program, from the READ statement to the GOTO statement, is called the LOOP (lines 30 to 50 inclusive). This will prove to be an extremely powerful feature of the language. For clarity, the BASIC statements within the loop are indented in the program listings.

Once back at line 30 of the program, the computer sees that it is to read two values again.

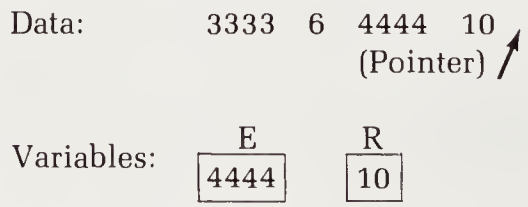
```
30 READ E , R
```

Therefore, it looks for our data again (in our DATA statements). It will pick up where it left off from before by noting where it left the pointer, and will simply take the first two values that it finds after the pointer. It moves the pointer once again of course.

These two new values will erase the two that were in the variables from before. That is, they are written "over top of" the old values that were in memory.

Thus 4444 replaces the 3333 in E, and 10 replaces the 6 in R. This is fine because we have already printed out the old values and do not need them anymore in our program. (As we said before, the variable names do not change, but the contents of the variables do.)

The situation inside the computer now looks like this:



The computer then goes to the next line in our program

```
40 PRINT E , R
```

and prints out the employee number and hourly wage rate (the ones now in E and R). These will be printed as indicated before, in zones 1 and 2.

Our output should appear as follows:

```
3333      6
4444      10
```

The computer then falls to the next line

```
50 GO TO 30
```

which loops it back to line 30 again (the READ statement). This time, however, when the program goes to the DATA statements, it finds that the pointer is at the end of the data. We have sent the program back to the READ statement once too often.

The computer tells us this fact by stopping at this point and printing out a message such as "OUT OF DATA". The final output from our program then will look something like this:

```
3333      6
4444      10
OUT OF DATA (This message varies with the computer used.)
```

Our program has ended!

Did you follow all that? The concept of the LOOP is key to writing programs. Maybe you should review it before moving on.

5-5. "Help! The Thing Won't Stop!"

Now that we are beginning to use loops, we also have the potential to get ourselves into some tricky situations.

For example, a program sent to the incorrect statement (usually with a GOTO statement) may end up "chasing its tail" until the power goes off if we don't stop it. This is called an ENDLESS LOOP or an INFINITE LOOP.

Sometimes it is obvious that the program is in an endless loop: the same thing keeps printing out, over and over again, endlessly.

Other times, it is not nearly so obvious - maybe the screen is blank and nothing at all seems to be happening - but the program is not ending either. About the only indication that the program is in an endless loop in this case is if it is taking much longer to run than it should. Any programs that we type in this text should run in a matter of seconds.

You could always turn the machine off and back on again, but with most computers you will lose what you have already done. If it is a long program, which you do not have saved on tape or disk, it could take a while to retype it.

With most computers we can stop an endless loop by pressing the BREAK key (or perhaps a "control-C" or a "control-Y" combination). Find out

beforehand how to stop a program that will not stop on its own - before you need to use it.

Now that you have found how to stop an endless loop on your computer let's try it; e.g., if we change line 50 in our program to:

```
50 GOTO 40
```

and then RUN the program, it will go into an endless loop. You must stop it. After you have, change line 50 back to the way it was.

Summary

We have learned the following statements to date:

Statement type	Purpose	Examples
1. LET	To assign a value.	40 LET X = 20 40 X = 20
2. PRINT	To print out values.	55 PRINT X , R 55 PRINT TAB(3),X 55 PRINT X ; R
3. READ	To read data from a DATA statement. It reads one set of data at a time.	40 READ D , K
4. DATA	To input values to a program.	100 DATA 40 , 66 (matches the READ statement above)
5. GOTO	To change the sequence of a program.	150 GOTO 40

We have learned that INPUT data can be included in a program in DATA statements and OUTPUT is printed using PRINT statements.

We discovered that LOOPING in a program allows us to reuse specific statements of a program.

As a minimum, skim through the previous pages. Even better, read them through once again. You will find that it is a bit like seeing a movie a second or third time – each time you see something you missed before, or only caught a glimpse of. It will be time well spent.

Once you feel confident about the previous material, you will be ready to move on to the next chapter where we will study calculations.

Exercises

When you write the programs required in the following exercises you will find that your program will probably differ slightly from one that someone else

might write. There is a certain amount of room for creativity and variety in programming and different individuals may use slightly different means to accomplish the same end. Some of those means may be more efficient than others, but I will not attempt to explain that here. Experience will usually show you the preferred methods.

I would like to emphasize at this point that even if you feel that you understand the principles we have discussed in this chapter, it is extremely important that you go through the process of keying in the programs and running them anyway.

You may discover that you are not as comfortable with the new concepts as you think you are. If nothing else, you will strengthen your knowledge of the topic under discussion.

5-1. The Invoice Exercise

Rewrite the program using READ and DATA statements instead of LET statements. The data is repeated here for your convenience.

Input data:	Item #	Price($)	Quantity sold
	6217	10.00	5
	3424	69.95	100

5-2. Write a program that will read in the following data and print it out.

Dept #	Division	No. of employees
16	1	18
24	2	27
30	1	103

5-3. Write a program that will read in the following data in the sequence given, but print it out in the opposite sequence; i.e., balance owing in the first column and customer number in the second.

Customer #	Balance owing($)
3456	333.44
1234	1800.00
765	70.00

5-4. "What Will Print ?" Exercise (or "Play Computer")

Study the following program. If it were keyed into the computer and RUN, what would be printed on the screen ?
(If your computer uses two print zones per line, skip to exercise 5-5.)

In this exercise you are asked to play the role of the computer, analyzing the program line by line.

This type of exercise is best done on a piece of paper. Use a series of columns and label each of them with one of the variables in the program; i.e., F, S, and T. Then go through the program one line at a time much as

the computer must. (You'll of course be a fair bit slower.)

```
10 DATA 3, 140, 888
20 READ F,S,T
30 PRINT F,S,T
40 PRINT T,S,F
50 PRINT F,T
60 PRINT F," ",S
70 PRINT S,S,S
80 PRINT " ",F
90 END
```

5-5. What would print if we change the commas to semi-colons in the program in exercise 5-4?

Type in the program and RUN it to see if you were right. You might type in lines 10, 20, 30, and 90 to begin with. RUN the program and note the output. Then add each of the other PRINT statements, one at a time, running the program after each addition. In this way it will be easier for you to tell what happens with each individual PRINT statement.

6 Calculations
(More of the LET Statement)

In most programs we want to perform certain calculations; e.g., calculate an individual's pay, determine closing balances, etc. We must examine, then, how calculations are added to a program.

New Vocabulary ASTERISK, CONSTANTS, EXPONENTIATION, HAT symbol, ORDER OF OPERATIONS

6-1. Calculations in a Program

As our example of using calculations, let's take the program we have been using in the last few chapters. We will assume that our employees have worked a 40-hour week for us and that we want to calculate their pay in our program.

We wish to produce a report that will look like this:

(number)	(rate)	(pay)
3333	6	240
4444	10	400

Our program will now look like this: (Clear your workspace and type it in.)

```
10 DATA 3333,6
20 DATA 4444,10
30 READ E,R
35    P = R * 40 (added)
40    PRINT E, R, P (changed)
50 GOTO 30
60 END
```

I have omitted the keyword LET. Note, too, that line 35 has been added and line 40 has been changed.

Line 35 multiplies the employee's hourly rate by 40 to calculate weekly pay. Notice that we have included the number of hours worked as a CON-

STANT in our multiplication. A constant is a value that does not change. Thus, the 40 hours would always be 40 for our program.

However, since employee numbers, pay rates, and pay will vary with employee, we use VARIABLES for them.

Let's examine line 35 of our program:

```
35 P = R * 40
```

The symbol "*" (asterisk or star) is used to tell the computer that you want it to multiply the numbers on either side of it. You cannot use an "X", parentheses, or any other form we use. The asterisk is the only symbol that the computer understands to mean multiply.

The computer calculates the employee's pay and stores it in the variable P in memory. (We made that name up.)

The computer can perform the following mathematical operations:

Operation:	**Symbol used:**
Addition	+ (plus sign)
Subtraction	- (minus sign)
Multiplication	* (asterisk or star)
Division	/ (slash)

Another helpful function, but one which we will not need, is exponentiation (or raising a value to a power). It uses the "hat" symbol (^), an "upwards arrow" (↑), or a double asterisk (**), depending on the computer being used.

Use the correct symbols; e.g., be sure to use the correct slash for division. It is the one that slopes upwards to the right (/) rather than the left.

Parentheses may also be used to simplify complicated expressions. Anything within parentheses is calculated first by the computer.

We now know the symbols that we may use, but there are a few other points to keep in mind as well when we perform calculations.

6-2. Rules for Calculations

Only one variable name is permitted on the left-hand side of the equal sign. Thus, we could not write line 35 like this:

```
35 R * 40 = P (incorrect)
```

Although such equations are perfectly permissible in algebra, they are not acceptable to a computer. This is because the computer automatically looks at the right-hand side of an equal sign, performs whatever calculations are indicated there, and puts the result in the variable name on the left-hand side. If it tried to do this with the line above, it would find the value of P, then try to put

it in the variable "R * 40", which it would recognize as an illegal variable name. (Remember, variable names cannot contain special symbols.) The computer will treat this as an error, and will not run your program.

However, we could have written it as

```
35 P = 40 * R
```

instead of

```
35 P = R * 40
```

There is no difference in these two expressions mathematically nor as far as the computer is concerned.

In this instance, it might help to think of the equal sign as meaning "is replaced by" rather than "is equal to". The computer replaces the contents of the variable named on the left-hand side of the equal sign with the result of the calculations that are on the right-hand side of the equal sign.

When the computer "sees" line 35 it realizes that it must set up a variable named P to hold the results of the calculation. Thus, it will create box P and put the result of R * 40 into it.

Remember that the READ statement contains only E and R. We will be calculating pay (P) using R * 40 but pay (P) is *not* included on the READ statement. Rather, it is calculated using values which were read in.

The computer will "create" a variable to hold pay when it is told to do so. This is acceptable to the computer as long as it has heard of all of the variables that appear on the right-hand side of the equal sign.

For most computers the word LET is optional in calculation statements.

If your computer allows its use, you may include LET or omit it as you wish. There is no particular advantage to either choice, except perhaps that omitting LET makes the program a little shorter in memory and involves a little less typing.

After performing the calculation on line 35 for us, the computer then drops to the next line of our program

```
40 PRINT E,R,P
```

and prints our results as follows:

```
3333      6      240
```

If your computer only has two print zones your output will look like this:

```
3333      6
240
```

Although this is acceptable for this simple program, it gets awkward to read the output if the program prints more than one line of information.

To overcome this limitation, use the TAB function (or possibly semi-co-

lons) as discussed in Chapter 3. The TAB function is probably the better of the two for our purposes; e.g., you might use

```
40 PRINT TAB(2); E; TAB(8); R; TAB(15); P
```

The program then drops to line 50

```
50 GOTO 30
```

which loops it back to the READ statement on line 30

```
30 READ E, R
```

There, it reads the next employee number and pay rate (moving the pointer in its memory as necessary) then drops to line 35 where it calculates weekly pay:

```
35 P = R * 40
```

It then drops to line 40

```
40 PRINT E, R, P
```

where it will print out the results for our second employee.

Our output now looks like this:

```
3333     6      240
4444     10     400
```

As before, the computer loops back to the READ statement once more, finds that there is no more data, and prints out an OUT OF DATA message below the last line of output.

In the above example we assumed that both employees worked 40 hours. What if each employee had worked a different number of hours? For example,

Employee Number	Hourly Rate($)	Hours Worked
3333	6.00	40
4444	10.00	35

We now have three variable data values to read in since all three data values do indeed vary.

Our program will now look like this:

```
10 DATA 3333, 6, 40
20 DATA 4444, 10, 35
30 READ E, R, H
35   P = R * H
40   PRINT E, R, P
50 GOTO 30
60 END
```

Since we need to read in three data values for each employee, we need three variable names in our READ statement. Then, in line 35, we use the variables R and H to calculate pay (P).

Many people have difficulty in sorting out the actual sequence of statements in their early programs. What will seem obvious and logical to you a few programs from now is often confusing at the start.

It may help to consider that the sequence of BASIC statements in your first few programs will generally be:

DATA statements
READ statement
LET statement(s) (calculations)
PRINT statement(s)
GOTO statement
END statement

It may seem more logical to you if you think of how you would produce the information that your program does; that is, how would you do it by hand?

Assume that you are given the input data for the employees on a piece of paper of some sort - perhaps a printed form. In this case you would probably proceed as follows:

a. Read the input values for the first employee from the given data.
b. Calculate the pay for the first employee.
c. Write the answers on a new piece of paper (your report with a title and headings of some sort.)
d. Go back to the given data to read the values for the second employee.
e. Calculate the pay for the second employee.
f. Write the answers for the second employee on your report.
g. Your report is now complete (for two employees).

This is precisely the sequence you tell the computer to follow when you write a program. The main difference is that you use the GOTO statement to tell the program to use some of the same statements over again: namely, the READ statement, the LET statement(s), and the PRINT statement(s).

6-3. Order (or Hierarchy) of Operations

For those of you who cringe at the sight of anything even remotely resembling algebra, this discussion might be a little bothersome. It is not really algebra though, so read on.

It should be mentioned at this point that sometimes when we write a mathematical expression, it can be interpreted in more than one way. Consider the expression R = X + Y/10. We can assume that the variable X has a value of 10 and the one called Y has a value of 20.

The above expression could be considered to mean

$$R = \frac{X + Y}{10} \text{ (answer is 3)}$$

or it could be interpreted as

$$R = X + \frac{Y}{10} \text{ (answer is 12)}$$

As humans, we might view the expression one way or the other, depending upon our own experience. A computer, on the other hand, interprets it according to a set of predetermined rules it has been given. It uses an "order of operations" which it has stored internally. This order of operations is as follows:

1. Parentheses (or brackets)
2. Raising to a power (exponentiation)
3. Multiplication and Division
4. Addition and Subtraction

The computer will begin at the left-hand side of the expression, performing calculations in the above order. Thus it will first examine the expression for parentheses. *Within* these, it will begin at the left-hand side, performing any exponentiation first, then from left to right again, performing any multiplication or division, and finally, any addition or subtraction.

If there are no parentheses, the above order still holds true.

Hence, the example we were looking at before

```
R = X + Y/10
```

will be interpreted as

$$X + \frac{Y}{10} \text{ (answer of 12)}$$

This is because division is higher in the order of operations than addition is. Thus the Y/10 is performed first, then X is added to the result.

In order to eliminate any confusion in mathematical expressions (and hence minimize errors), it is a good idea to include parentheses in all but the simplest expressions.

Our sample expression could then be written in one of two ways :

```
R = (X + Y)/10
```

or

```
R = X + (Y/10)
```

depending upon which interpretation we wish.

Even though the second choice does not require parentheses, it is probably a good idea to include them. They certainly do no harm, and if they help at all to keep your thinking straight, then they serve a useful purpose.

The reason that I have explained this rule in such detail is that as your programs become more complicated, this order of operations is more apt to result in errors which are difficult to detect.

6-4. Multiple Calculations on a Single Line

It is possible to combine several calculations on a single line. For example,

```
50 M = (3 * C) + (40/(8 * X)) - (0.50 * D)
```

Sometimes this is helpful. At other times, however, it makes the program unnecessarily complicated or confusing. Complicated expressions are also more error prone.

You are often better off to break up lengthy calculations into smaller units and combine them later.

For example, the expression above could also be written as:

```
50 M1 = 3 * C
52 M2 = 40/(8 * X)
54 M = M1 + M2 - (0.50 * D)
```

or some such combination of parts of the initial expression.

6-5. Calculations in a PRINT Statement

It is also possible to perform calculations in a PRINT statement:

```
10 DATA 40, 100
20 DATA 60, 200
30 READ M, X
40    PRINT (8 * M)/(4 + X)
50 GOTO 30
60 END
```

Little is gained using this technique. In fact, you will often find it inconvenient to use. It is available as an option, however.

The way we would normally do the above is:

```
10 DATA 40, 100
20 DATA 60, 200
30 READ M, X
40    Q = (8 * M)/(4 + X)
45    PRINT Q
50 GOTO 30
60 END
```

6-6. Rounding

Any of our calculations that have resulted in answers with decimal positions to date, have not been rounded; e.g., we might have received an answer such as 34.569 when we would have preferred that the computer round the answer to

34.57. It is possible to round our answers using something called the INTEGER function. (See Appendix B for an explanation.)

6-7. Percentages in Calculations

Any calculations involving percentages require that you use the decimal or fractional equivalent of the percentage involved in your programs; e.g.,

8% becomes 0.08 or 8/100

You cannot use the % symbol – the computer will not understand it. If you do include it, you will receive an error message.

Slowly but surely, we are delving deeper into the world of programming. In the next chapter we will examine the concept of ALPHANUMERIC (or STRING) data; i.e., we want to be able to add employee names to our program. Names are one type of alphanumeric data.

Summary

Calculations can be performed in BASIC using certain symbols :

( ) parentheses – to simplify expressions and thus minimize errors and confusion.
\+ for addition
\- for subtraction
* for multiplication
/ for division
↑ or ↑ or ** for exponentiation

These symbols must always be used.

Only one variable name is permitted on the left-hand side of the "equal sign", e.g.,

```
40 W = (P * A)/3
```

The keyword LET is optional on most computers.

We learned that if we are "fuzzy" in our mathematical expressions, the computer will evaluate them according to its "order of operations" (also called the "hierarchy of operations"). This order is as follows:

1. Parentheses
2. Exponentiation (raising to a power)
3. Multiplication and Division
4. Addition and Subtraction

CONSTANTS may be inserted in expressions. In the example,

```
40 W = (P*A)/3
```

the 3 is a constant; that is, its value does not change (contrast with the term VARIABLE).

An Approach to Solving Problems

For the first few exercises in this book you may feel at a loss as to where to begin or how. If that's the case, these tips may be of some help to you:

a. Look at what you are given as input data in the exercise. Make up variable names for each data item; e.g., E for employee number, D for department number, etc.

Write down the names you choose so that you will be able to keep your thinking straight. These variable names will appear in your READ statement.

b. Look at what is required as output; i.e., what is to be printed? You may find that some of the information that is to be printed out was given with the input. If that is the case, make a note of these variable names – they will be the same as they are on the READ statement.

c. For those values that are required for output that were not provided as input, it will be necessary to perform certain calculations. Determine what these must be. Make up variable names as necessary and write the necessary instructions.

d. The variable names of those values that are to be printed must appear on the PRINT statement.

e. As mentioned earlier, the sequence of BASIC statements in the program will be DATA, READ, LET (calculate), PRINT, and GOTO (at least for the first few programs). Remember that DATA statements can be placed anywhere in the program.

Exercises

6-1. The Invoice Exercise

Item #	Price($)	Quantity sold
6217	10.00	5
3424	69.95	100

Calculate the extended price for each item (price times quantity) and print out ITEM NUMBER, PRICE, and EXTENDED PRICE for each of the two items.

Your output should be as follows: (spacing may vary)

(item #)	(price)	(extended price)
6217	10	50
3424	69.95	6995

6-2. Write a program that will read in the following data for a retail sales outlet:

Department No.	Total sales($)	No. of sales personnel
12	5000	10
33	6600	20

Calculate the average sales per salesperson for each department. Print out DEPARTMENT NUMBER, NUMBER OF SALES PERSONNEL, and the AVERAGE SALES (in dollars) for each department.

The first line of output should be:

(dept. no.)	(no. of salespersons)	(average sales)
12	10	500

6-3. What Will Print ? ("Play Computer")

If the following program were typed into the computer and RUN, what would the output be ? (This program adds together the number of items in stock in each of three warehouses.)

```
10 DATA 466, 30, 15, 6
20 DATA 866, 40, 80, 10
30 DATA 529, 7, 2, 15
40 READ I, W1, W2, W3
50   T = W1 + W2 + W3
60   PRINT I; W1; W2; W3; T
70 GOTO 40
80 END
```

After you have tried this question, clear your workspace and type the program in. Then RUN it and see if you were right.

6-4. Given the data:

Item #	Quantity Sold	List Price($)	Cost($)
1630	2000	5.00	4.50
2518	160	20.00	15.75
5286	10	65.50	50.00
9340	500	14.95	10.50

Write a program to print out a report of the following format:

(item)	(quantity)	(unit profit)	(total profit)
1630	2000	.5	1000

Hint: To determine unit profit, subtract the cost from the list price. Total profit is then quantity times unit profit.

6-5. Given the following:

Item #	List Price($)	Discount Rate(%)
5281	400	6
1786	600	10
437	750	8
8261	45	0

Write a program that will print out the following report:

(item #)	(price $)	(discount $)	(net price $)
5281	400	24	376

Note that the discount rate is a percentage, but the actual discount (in dollars) is to be printed out on the report.

The computer does not understand a percentage symbol - you will receive an error message if you try to use one in a calculation. You must use the equivalent decimal or fraction for the percentage in question; e.g., .08 or 8/100 for 8%.

Note: Net price is the list price, less the discount($).

6-6. An insurance company wants to calculate total premiums for a number of policyholders. Premiums are based on a rate per $1000 of the face value of the policy.

Input data:

Policy #	Rate($) per $1000	Face Value($)
1005	7.25	7500
2264	5.15	5000
5017	9.00	10500
3497	6.25	6500

Write a program to calculate the total premium for each policyholder. Print out a report as follows:

(policy #)	(face value $)	(total premium $)
1005	7500	54.375

7 The End-of-Data Test

This chapter introduces us to the BASIC statements we need to end our programs without the error message we have been receiving to date.

New Vocabulary DECISION point IF statement

7-1. Ending a Program

You have noticed to this point that when we run our programs, they always end by printing out a message such as OUT OF DATA LINE #20.

It may have been an annoyance, but our programs worked, nonetheless. Now, however, we wish to have our programs end "cleanly". This will be necessary in a later chapter when we wish to keep the total of a column of data, and print it out at the bottom of the report.

The way that we eliminate this OUT OF DATA message is as follows:

a. Add a "dummy set" of data to our program. It must be the *final* set of data and it must be a *complete* set. Recall that a set of data consists of the data values for one person, one item, etc.

b. Test for this dummy set of data in our program with something called an IF statement. If we can spot the dummy set of data, then we know that we are at the end of our data and we can tell the program to end properly instead of having it go back to the READ statement.

Our program will now look like this:

```
10 DATA 3333,6 (1st set of data)
20 DATA 4444,10 (2nd set of data)
30 DATA 0,0 (dummy set of data)
40 READ E, R
50    IF E = 0 THEN 90 (the IF statement)
60    P = R * 40
70    PRINT E, R, P (you may need the TAB function)
80 GOTO 40
90 END
```

Clear your workspace and type it in.

KEY POINT — THE DUMMY SET OF DATA MUST BE THE LAST SET OF DATA AND IT MUST ALSO BE A COMPLETE SET.

The above example includes dummy entries for employee number and wage rate (both zero). Just one zero would have been incomplete. The dummy set of data must match the valid sets in type of data and number of values. This is true no matter which data value you use in your end-of-data-test.

If you do not have two data values in your dummy statement, you will receive an error message.

The data values that are used in the dummy set must be values that will not occur for valid data. For example, use an employee number of 0, or maybe 99999. Most people use zeros or nines as their dummy values.

As a result of adding the dummy data line

```
30 DATA 0, 0
```

and the IF statement

```
50 IF E = 0 THEN 90
```

the program output will now look like this:

```
3333     6    240
4444    10    400
```

It may seem like a lot of bother for what we accomplish (i.e., getting rid of the OUT OF DATA message) but, as mentioned earlier, we have no choice if we wish to be able to print out columnar totals at the bottom of reports in the future; and, of course, it does make for a more attractive report, which in itself is a benefit.

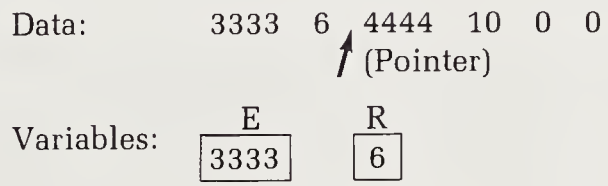
Explanation Let's have a look at what goes on in the program to see how the OUT OF DATA message was eliminated.

In examining our program, we observe that the first thing the computer sees is the DATA statements, which it makes note of.

It then encounters the READ statement

```
60 READ E, R
```

and reads in the first set of data. The situation inside the computer's memory is:



7-2. The IF Statement

Now the computer goes to line 50 and sees the statement:

```
50 IF E = 0 THEN 90
```

(By the way we could have used 50 IF R = 0 THEN 90, since R is also equal to zero when the dummy data is read.)

The basic format for an IF statement can be considered to be (for now at least):

IF condition THEN line number

where the condition is an expression such as we have used,

```
E = 0
```

and the line number is another line of the program that you wish to "branch to" if the condition is true.

By the way, some computers permit (or require) the form:

IF condition THEN GOTO line number

In our sample program, the computer looks at the condition (is E = 0?), and if it is satisfied, that is, if it is true, then the program will branch to line 90 as ordered to. If it is not true, it just continues to the next line of the program (line 60).

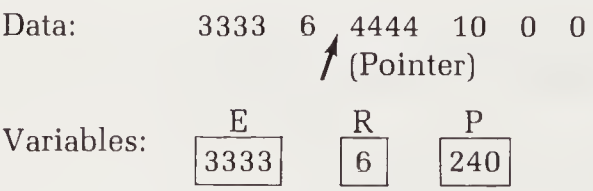
This is what we call a *decision point* for the program. Depending upon the value of E, it will either jump to line 90, or simply fall down to the next line of the program (line 60 in our example). If the value of E is equal to zero, it will branch to line 90; otherwise, it will continue on to line 60 and calculate the pay for the first employee.

Since we are dealing with our first set of data

```
10 DATA 3333, 6
```

E is not zero, so the program continues to line 60. At line 60 it will calculate the pay.

The situation inside the computer's memory is now:



The program then falls to line 70 where it will print out the answers (employee number, rate, and pay). Output to this point will be as follows:

```
3333     6     240
```

KEY POINT — IF THE CONDITION IN AN IF STATEMENT IS TRUE, THE PROGRAM WILL BRANCH TO THE LINE NUMBER INDICATED. IF IT IS NOT TRUE, THE PROGRAM SIMPLY GOES TO THE NEXT LINE BELOW THE IF STATEMENT.

IF statements are referred to as CONDITIONAL STATEMENTS. We have used one as a "conditional branch"; i.e., the program will branch to another line if a certain condition is satisfied. In our case, the condition is an end-of-data test.

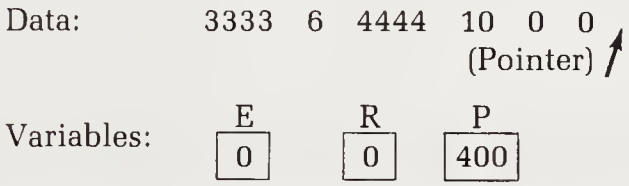
Next, the program falls to line 80, which sends it back to line 40, where it will read the next set of data

```
20 DATA 4444, 10
```

The program will deal with this set of data as it did with the first. The output (so far) would be:

```
3333      6     240
4444     10     400
```

Now, the program returns to line 40 to read the next set of data (the dummy set). The situation in the computer's memory is now:



The next line of the program to be executed is line 50. This is the IF statement, which checks to see if the last set of data read in is the dummy set (Has E been replaced by zero?). Because it finds that the set of data in the computer's memory is indeed the dummy set, the program will take the "action" path of the IF statement, jumping to line 90 and ending "cleanly".

The OUT OF DATA message that we have been receiving to date will not be printed any more!

KEY POINT — THE IF STATEMENT THAT TESTS FOR THE DUMMY DATA SHOULD ALWAYS BE IMMEDIATELY AFTER THE READ STATEMENT. THAT WAY THE PROGRAM WILL DETECT THE DUMMY DATA AS SOON AS IT READS IT AND IT WILL NOT PERFORM ANY CALCULATIONS USING THE DUMMY DATA.

The IF statement turns out to be a very powerful feature in a programming language. It provides a means of constructing a "fork in the road" in a program.

Depending upon whether a particular variable is a certain value or not, we will be able to take one fork or the other.

This change in our program now permits us to print out totals in our reports, which we have not attempted to this point, but will be doing very soon.

Summary

By adding a dummy set of data to our program we are able to eliminate the OUT OF DATA message that has been printing to date.

This dummy set must be the same format as the valid data and must be a complete set; e.g., if the valid data is in the form:

```
200 DATA 16, 22, 50
```

then the dummy set must be in the form:

```
210 DATA 0, 0, 0
```

The dummy set of data is detected through the use of an IF statement of the form:

```
60 IF N = 0 THEN 300
```

and the program is ended "cleanly".

Exercises

7-1. The Invoice Exercise

Add a dummy set of data to your data and include an end-of-data test in your program to detect it.

Item #	Price($)	Quantity sold
6217	10.00	5
3424	69.95	100

Output should be:

(item #)	(price)	(extended price)
6217	10.00	50
3424	69.95	6995

7-2. If the READ statement of a program were as follows:

```
40 READ X, M
```

which of the following DATA statements could be used as the dummy set?

a. 200 DATA 9999
b. 200 DATA 0
c. 200 DATA 0, 9999

d. 200 DATA 999, 9
e. 200 DATA 99, 99, 99

7-3. The following investments earn interest at the rates indicated:

Investment ($)	Interest rate (%)
4000	10
600	15
100000	8

Write a program that will show the value of the investments at the end of one year.

Recall that you cannot include commas *within* data values and you are not able to use a percentage symbol in your calculations.

Print out the ORIGINAL INVESTMENT and the VALUE AT YEAR-END for each amount shown. Be sure to add a dummy set of data and test for it. The first line of output should be:

```
4000      4400
```

7-4. Input data:

Employee #	Sales ($)	Commission Rate (%)
6789	14000	10
4050	8000	9.5
3838	20000	12

Write a program to print a report of the following form:

(employee #)	(sales)	(commission %)	(commission $)
6789	14000	10	1400

Note that commission rate (%) and commission paid ($) are printed on the report.

Hints: **1.** Commission paid is calculated by multiplying sales ($) times commission rate (%).
2. To use a percentage in your calculations it must be in decimal form (.10), but to print it out, it should be printed as a whole number (10). One way to do this is to divide and/or multiply the commission rate by 100 at some point in your program.

8 The Programming Cycle

This chapter offers us a means of organizing our work, thus saving us time in the long run.

New Vocabulary	ALGORITHM	DOCUMENTATION
	CODE	FLOWCHARTING
	DEBUGGING	PROGRAMMING CYCLE

8-1. Steps in the Programming Cycle

So far our examples have been very simple. Because of this, little planning or analysis has been needed. As programs become more involved, however, it is helpful if we have a well-defined plan-of-attack. The PROGRAMMING CYCLE offers us this. We can follow the steps it suggests to arrive at a solution to our problem:

The Programming Cycle

1. Analyze the problem and plan a solution.
2. Flowchart the solution.
3. Write (or code) the program.
4. Test and debug the program.
5. Document your solution.

Let's look at each of the above steps more closely.

1. Analyze the Problem The first step is to clearly define what is required in our problem and to formulate a plan to accomplish this. The plan can often take the form of designing the input, processing and output steps of a problem.

It is often best to look at the output first. We will then be able to determine what must be provided as input and what processing is necessary to produce

the desired output; e.g., if our problem is to produce paycheques, we will need employee numbers, names, hours worked, wage rate, deductions, tax rate, etc., as input. As far as processing is concerned, we know we must calculate gross pay, then subtract tax and deductions, to arrive at the employee's net pay. You could jot down some notes describing these requirements.

2. Flowchart the Problem This analysis and planning step may be partly combined with step 1, since flowcharting is really an attempt to draw a diagram of the logic required to solve a problem. As such it will be a diagram of the program which we will use to solve our problem. More on this in a moment.

Some authors suggest the use of ALGORITHMS for the planning stage. An algorithm is a series of operations or steps designed to produce a solution for a problem. In effect, we used an algorithm above in deciding what calculations were required to produce our paycheques, and in what sequence these calculations had to be performed.

Our algorithm for the above example might be as follows:

a. Read in employee number, name, wage rate, deductions, etc.

b. Multiply hours by wage rate to obtain gross pay.

c. Subtract deductions (including tax) from gross pay.

d. Print out a cheque with name, net pay, etc.

3. Write the Program Once the planning (steps 1 and 2) is completed, the program is written. This is often referred to as the CODING stage. The program itself is called the CODE. It is highly recommended that you do this away from the computer, perhaps at a desk or at a comfortable table, since it helps if you have lots of room to spread out your paperwork.

4. Test and Debug The program must next be tested to see if it performs as desired. If it does not, we make the necessary changes and try again. Because it involves a great deal of trial and error, this is the stage where you are apt to be pulling your hair out.

You should work out the results by hand for one or two sets of data; i.e., perform the calculations involved. Often a program will print out results when you run it, but not the correct ones. Sometimes it will be obvious at a glance that the results are in error, but at other times they may seem correct. Unless you try a few calculations by hand, you cannot be sure that your program is performing as it should.

Errors of this nature are referred to as logic errors. That is, you are following all the rules of BASIC in writing your statements, but you have made a slip of some sort in writing the expressions you use or in what you are asking the computer to print out. Perhaps you are multiplying in an expression when you should be dividing, or you have forgotten to subtract a value. There are many

possibilities. Checking one or two results by hand will catch most errors of this type.

By the way, errors in programs are called BUGS, hence the process of finding and correcting errors is called DEBUGGING.

Appendix A may be of help in this step.

5. Documentation Once the bugs are out of the program and it is performing as you want, it is time to DOCUMENT the program. DOCUMENTATION means setting up the paperwork to show or explain such things as what the program is doing, what department uses its information, where the input data is obtained from, etc. It would also include a flowchart, a listing of the program and a sample of the report that it produces.

This documentation is valuable for future reference by the programmer. It can also be used to explain to others what the program does, how it does it, etc.

Because common sense is at the base of the above cycle, many people follow it or something close to it automatically.

You may find that when you write simple programs, you do not have to follow a formal procedure. That's fine, but remember that as programs become more involved, the above problem-solving approach will prove helpful. Organization of your work is very important and the programming cycle offers you a tool to help in that regard.

Next we will examine flowcharting in detail.

8-2. Flowcharting — A Picture is Worth a Thousand Words

Often before we design or build something, we draw a diagram of our project. The diagram proves useful in setting out what is to be done and in what sequence it can best be accomplished.

It is also helpful to have a diagram of what is required in a program before we begin to "build" it. The diagram used to design a program is called a PROGRAM FLOWCHART.

Definition: A program flowchart is a diagram of the logical steps required to solve your problem. It will include what information is read in as input, what calculations are performed, and what output is printed.

Flowcharting is a somewhat contentious topic today. Some authorities feel that it is of questionable value. They suggest that other more efficient means can be used to record program logic. Others support its use, suggesting that flowcharting can be a valuable tool for the programmer, the user and anyone else associated with a program.

There are a number of arguments for the use of this technique:

a. It offers the valuable advantage of helping you sort out your logic before you write your program, thus saving you considerable "trial-and-error" time in the programming process.

By learning the concepts of flowcharting, you will be able to draw up a

"picture" of the logic required to solve your problem. Then, by following it, you will be able to write your program almost directly.

b. Flowcharting is used more and more today outside the computer industry as an aid in explaining paper flows, corporate structure, etc. Hence, an understanding of it may help you in other pursuits.

c. Flowcharting has been used in the computer industry since its inception and, as such, an understanding of it is necessary if you are to understand those flowcharts already in existence, particularly for those of you who would like to work in the computer industry.

Flowcharting is probably of less value in writing shorter programs (as most of those in this text are) than it is in writing longer ones.

Many people find this technique very confusing and end up drawing a flowchart for a program only after they have already written the program. Although it is still of value after the fact, those who master the art of flowcharting and draw their flowcharts *before* they write their programs will gain a considerable advantage over those who do not. It is worth your while to attempt to master it as soon as possible; it can soon pay dividends, especially when programs become more complicated.

Flowcharting Symbols In order to make the flowchart understandable to as many people as possible, we use standard symbols when we draw one. These symbols have been agreed to by those in the computer industry. A FLOWCHARTING TEMPLATE is used to simplify the drawing of them. These templates are usually made of plastic and are relatively inexpensive.

There are several symbols on a flowcharting template but we will only use a few of them. The basic symbols are shown in Figure 8-1.

Let's look at a flowchart for our program from the last chapter:

```
10 DATA 3333,6 (1st set of data)
20 DATA 4444,10 (2nd set of data)
30 DATA 0,0 (dummy set of data)
40 READ E, R
50    IF E = 0 THEN 90 (the IF statement)
60    P = R * 40
70    PRINT E, R, P (you may need the TAB function)
80 GOTO 40
90 END
```

The flowchart follows the logic of our program. Each symbol is the appropriate shape for the statement concerned you will note.

Normally we do not show the following on flowcharts:

DATA statements.

The printing of blank lines.

There are also a number of rules to keep in mind for flowcharting:

a. Use the standard symbols — do not make up ones of your own.

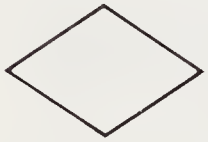
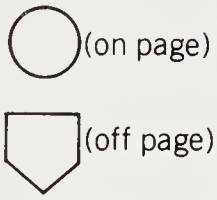
		Symbol Name	*Corresponding Basic Statement*
1.		Terminal (Start or end)	a. None for start b. END for end
2.		Input/Output	a. READ for input b. PRINT for output
3.		Process (for calculations or assigning values to a variable)	a. LET
4.		Decision (to indicate a possible branch in your program)	a. IF
5.		Connectors (to indicate that flowchart continues elsewhere on the same page or another page)	
6.		Flowline	a. None (for those between adjacent symbols) b. GOTO for those branching to a symbol that is not adjacent

Fig. 8-1 Flowcharting Symbols

b. Have the flowchart read from top to bottom, and from left to right as much as possible.

c. Use arrowheads on flowlines.

d. Use on-page and off-page connectors to simplify flowcharts when necessary.

Use English in the symbols of the flowchart or English and BASIC, but not just BASIC. This is not a widely accepted rule but it is a good idea. The English will make it easier to trace your flow of logic when you refer to your flowchart.

Important Note A point to keep in mind as you are learning flowcharting is that the decision symbol, the "diamond", is the only one that can have more than one line exiting from it. All of the others can only have one.

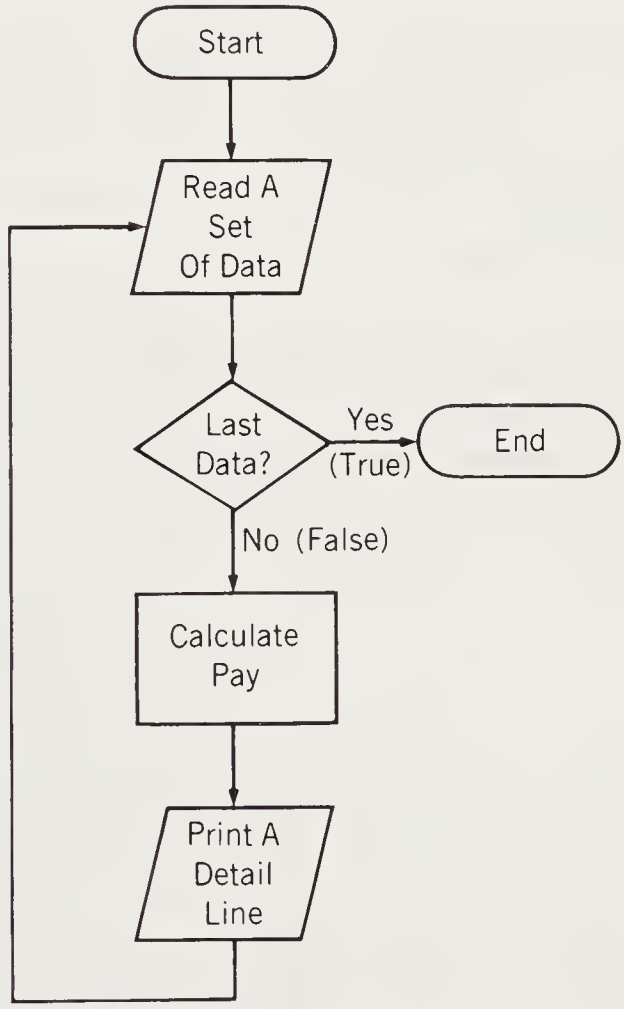


Fig. 8-2 Flowchart of Sample Program

Flowcharting – Yes or No? Flowcharting may be considered as an optional topic. It is recommended, however, that you at least read through the material to become familiar with this powerful tool. The choice to use it in your programming is at your discretion or that of your instructor.

By the way, you would be well advised to draw your flowcharts in pencil and keep an eraser handy - you will use it frequently.

Summary

By following an organized approach to solving a problem, you are less likely to omit something important or make an error. You may also save yourself considerable time in the process.

The programming cycle outlined in this chapter is such an approach. FLOWCHARTING is a key part of this cycle.

Exercises

8-1. Define the following terms:

a. PROGRAMMING CYCLE

b. BUG

c. FLOWCHART

d. ALGORITHM

e. DOCUMENTATION

8-2. Although we do not normally draw flowcharts after writing the program, it would be good practice for you to go back to earlier chapters and draw flowcharts for the exercises.

In future, of course, you should draw the flowcharts first.

9 Alphanumeric Data (and the REMARK Statement)

In most programs we want to include names as well as numbers. This chapter will show us how this is done. We will also be introduced to the REMARK statement, which will allow us to write notes to ourselves (or to others) within a program.

New Vocabulary	ALPHANUMERIC	LOOP section
	CLOSING section	REMARK statement
	OPENING section	STRUCTURED PROGRAMMING

9-1. The REMARK Statement

Before we begin to discuss the concept of alphanumeric data, let's look at the REMARK statement.

Since our memories at times fail us, or someone else might be looking at our program in the future, it would be helpful if we could make notes within our program; for example, we might wish to record who wrote the program, when it was written, what variables were used, etc. This is particularly important in longer programs.

It is in fact possible to do this with what are called REMARK statements. A REMARK statement looks like this :

```
10 REM I. M. BELL JULY 7, 1999
```

A REMARK statement must be numbered as any other line of a program would be, and identified by the letters REM (you can use the full word REMARK but it is not necessary). The remark itself can be any words, symbols, etc., that you wish to use.

REMARK statements are for humans and the computer will thus ignore them. The remarks will be printed out when you LIST your program but they will not appear anywhere when you RUN it. You may use as many REMARK statements as you wish and may place them anywhere in your program. They are very helpful and I recommend that you get in the habit of using them often.

It takes only a few extra minutes to add them, and they may be especially useful in helping you to follow the logic of long, complicated programs.

Another valuable use of REMARK statements is simply to space program listings. By adding lines such as

```
53 REM
55 REM ************************************************************
57 REM
```

at appropriate points in a program, it is divided into more easily recognized sections, which often make it simpler to read and understand.

Anything that helps us with understanding should be used often. Get in the habit of employing REMARK statements to explain what the variable names in a program represent.

9-2. Types of Data

The computer does not care about the actual values of our data; e.g., whether it is $2.55 or $6800.75, but it does care about the type of data we use.

It divides data into two types: NUMERIC and ALPHANUMERIC (or STRING).

NUMERIC Data Numeric data will consist of information that can be used in arithmetic calculations. As mentioned before, numeric data can contain only the following:

a. The digits 0 through 9.

b. A decimal point.

c. A sign (+ or –). If we do not include a sign, the computer will assume that the number is positive (+).

To this point, we have used only numeric data.

KEY POINT — DO NOT INCLUDE A DOLLAR SIGN ($) IN NUMERIC DATA IN A PROGRAM. THE ONLY THINGS ALLOWED IN NUMERIC DATA ARE DIGITS, A SIGN, AND A DECIMAL POINT. IF YOU INCLUDE A $ YOU WILL RECEIVE AN ERROR MESSAGE.

ALPHANUMERIC (String) Data Alphanumeric data cannot be used in arithmetic calculations. Some examples are

a. names such as HILL, DALE, SMITH, etc;

b. addresses such as 866 FOURTH AVE;

c. codes such as 3216B, 9C24766, 45HK78, etc;

d. phone numbers such as 555-2518 or 1-416-555-0210.

Note that an address usually contains both numbers and letters. The computer cannot, however, consider a piece of data to be both numeric and alphanumeric; and since an address does not qualify as numeric data because it includes letters, it is considered to be alphanumeric data by the computer.

Likewise, the codes given in the examples above are alphanumeric data. The phone numbers are alphanumeric because they include dashes.

The fact that most of a data item is numeric is not important. The key factor is that the item contains something other than digits, and a decimal point and/or a sign. Thus, by definition, it must be considered to be alphanumeric data.

Some people use the term ALPHANUMERIC to describe non-numeric data. Others refer to it as STRING, ALPHAMERIC or ALPHABETIC data.

The computer distinguishes between numeric and alphanumeric data because it treats them differently in its circuits.

It will only do arithmetic operations (addition, subtraction, etc.) on numeric data. Alphanumeric data cannot be used in calculations, however. It can only be "read in" and/or "printed out".

We, as programmers, help the computer to keep the alphanumeric and numeric data sorted out by choosing appropriate variable names. Not that we have any choice in the matter. If we do not follow the rules, the computer will not run our program for us but will print out some message such as DATA OF WRONG TYPE, a common sight to beginning programmers.

In previous chapters we have only used numeric data. Now let's assume that we want to add some alphanumeric data (e.g., employee names) to the program which we wrote in the last chapter.

Our input will now be as follows:

(name)	(number)	(hours worked)	(hourly rate)
HILL	3333	40	6
DALE	4444	30	10

Let's now assume that we have been asked to produce the following report:

(number)	(name)	(hours worked)	(pay)
3333	HILL	40	240
4444	DALE	30	300

Note that we have switched the order of employee number and employee name. This is simply to emphasize that data can be printed in any order we choose and it doesn't depend on how it was read in.

Before we look at the program that will accomplish this, there are a couple of points to note.

9-3. Sections of a Program

It will be helpful in this chapter, and in later ones, if we begin to think of our programs as having three basic sections in them. This lends structure and orga-

nization to our program, important characteristics of what is called STRUCTURED PROGRAMMING.

Structured programming is a style of writing programs which results in their being easier to understand and change, if the need arises. Essentially, it involves writing clear, logical statements in sections, or modules, and organizing these modules in a logical sequence.

To assist the programmer in this regard, some versions of BASIC have special statements or groups of statements (see Appendix D).

The three program sections that will help us to follow the principles of structured programming are:

a. The "opening" section to set up starting values, etc. It is performed only once. In our program this section has thus far only included the DATA statements. REMARK statements are often included here listing pertinent facts about the program.

b. The "loop" section, which is repeated several times. The computer goes "round and round" the loop. In our program so far this was the section from the READ statement to the GOTO statement inclusive. It is performed *once for each set of data*. Thus, in our simple program which included two employees, the loop portion of the program was performed twice.

This is usually the portion that includes calculations and the printing out of employee information, pay data, etc.

c. The "closing" section to end the program. It is performed only once. It has simply been the END statement so far.

Back to the report we wish to produce.

The flowchart for a program which would produce such a report is shown in Figure 9-1.

Using the above flowchart we would code the following program:

```
                10 REM PROGRAM USING ALPHANUMERIC DATA
                20 DATA "HILL", 3333, 40, 6
Opening         30 DATA "DALE", 4444, 30, 10
                35 DATA "X", 0, 0, 0
                40 DIM N$(10)

                50 READ N$, E, H, R
                55   IF H = 0 THEN 90
Loop            60   P = H * R
                70   PRINT E, N$, H, P
                (or semi-colons or TAB)
                80 GOTO 50

Closing         90 END
```

Clear your workspace and type the above program in. Note the double quotation marks around the employee names, including the dummy name.

Looking at the program itself in the light of the three sections mentioned above, our program includes the following:

Opening section – lines 10 to 40 inclusive.

Loop section – lines 50 to 80 inclusive.

Closing section – line 90 only.

RUN the program to see that it produces the report that we want. LIST the program for practice.

9-4. Rules for Alphanumeric Data

Let's examine that program. There are a number of rules which govern the use of alphanumeric data. These must be adhered to rigidly.

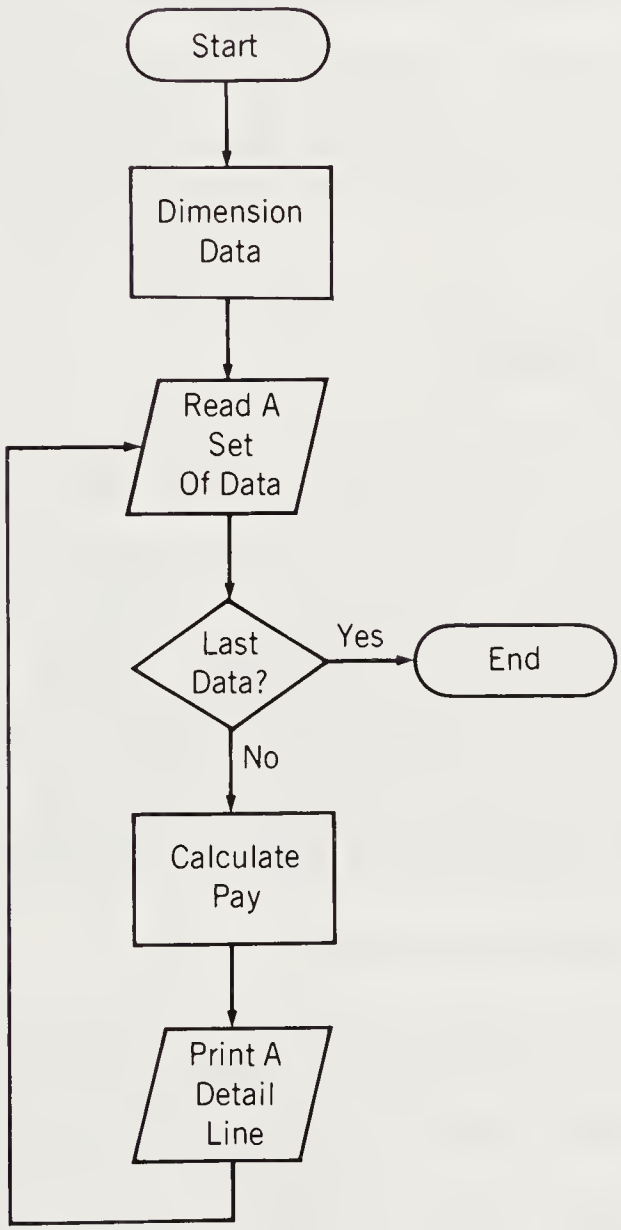


Fig. 9-1 Flowchart of Sample Program

Rule 1. The alphanumeric data itself must be included in DATA statements in the same way as numeric data. Some computers require that alphanumeric data be enclosed within quotation marks (double quotes rather than single). Note lines 20, 30 and 35 in the program above.

Even if your computer does not require quotation marks, it will still accept them. Try writing the DATA statements without quotation marks around the alphanumeric data to see if your computer requires them or not. If it does you will receive an error message when you leave them out. Most microcomputers do not need them.

Rule 2. The variable name for the "box" that will contain the alphanumeric data must be a single letter of the alphabet followed by a dollar sign ($); e.g., N$, A$, G$, etc.

Many computers allow more forms but we will manage with this one. Experiment a little if you wish. Check if longer names are permitted on your computer: they are easier to remember and more meaningful as well.

Some computers allow any numeric name with a $ added to the end of it as a valid alphanumeric variable name; e.g., E5$, G$, T2$, etc.

Rule 3. Line 40 in the program

```
40 DIM N$(10)
```

is called a DIMENSION statement. This statement is mandatory with alphanumeric variables for many, but not all, computers. Again you must test yours to see.

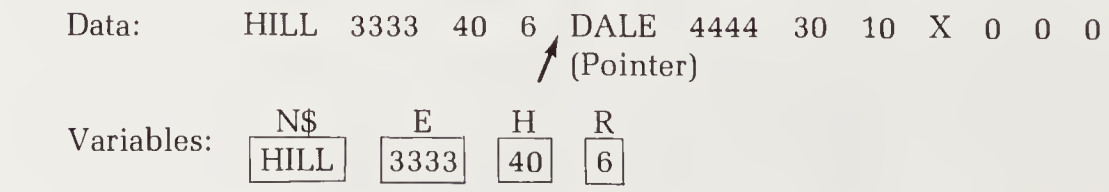
The DIMENSION statement consists of the letters DIM followed by the alphanumeric variable name involved. This name in turn is followed by a number in parentheses. The parentheses must be included. This number is the maximum size that the data can attain. Many computers only require a DIM statement if the data involved will exceed ten positions in length.

In our case, we have allowed for 10 positions as a maximum; hence, any name longer than that will not fit, but will be truncated (cut off) at 10 letters.

By the way, if we had two alphanumeric items in each set of data (each DATA statement), we could write a DIMENSION statement such as the following:

```
40 DIM N$(10), D$(12)
```

The Program Let's analyze our program. After the computer has read line 50, the following situation exists inside the memory of the computer:



The next line that the computer finds is our end-of-data test.

```
55 IF H = 0 THEN 90
```

In looking at the value of H, the computer sees that it is not zero (we are now dealing with the HILL data), so it continues on to line 60.

At line 60 it finds that it is to multiply H by R and put the results in P. P does not exist at this point, so the computer creates a "box" in memory and labels it P.

The situation inside the computer's memory is now as follows:

Data: HILL 3333 40 6 DALE 4444 30 10 X 0 0 0
(Pointer)

Variables:	N$	E	H	R	P
	HILL	3333	40	6	240

(The data and the pointer remain unchanged.)

The next line of the program

```
70 PRINT E, N$, H, P
```

tells the computer to print out the values for employee number (E), name (N$), hours (H), and pay (P). These will appear on the screen/paper as follows:

```
3333     HILL     40     240
```

(Note that we chose not to print hourly rate.)

The next line of the program that the computer encounters is

```
80 GOTO 50
```

which sends it back to line 50.

Thus, we have completed the loop section of the program once and it is time to go through it again. This time it will perform the same operations on the second set of data. (Recall how it moves the pointer in the DATA statements.)

Now it reads in the second set of data, by going to the DATA statement, checking to see where the pointer is, and getting the next four items of data it encounters. The situation inside the computer would then be as follows:

Data: HILL 3333 40 6 DALE 4444 30 10 X 0 0 0
(Pointer)

Variables:	N$	E	H	R	P
	DALE	4444	30	10	240

Note that the pay in P is still that for HILL, since the computer has not yet reached the line that calculates pay.

Line 55 is next:

```
55 IF H = 0 THEN 90
```

which is our end-of-data test. It is not the dummy set of data so the computer continues to line 60. It takes the values from H and R as before, multiplies them together and puts the result in P. Now we have:

Data: HILL 3333 40 6 DALE 4444 30 10 X 0 0 0
(Pointer) ↑

Variables:

N$	E	H	R	P
DALE	4444	30	10	300

Next the computer goes to line 70, which tells it to print the contents of E, N$, H, and P. We now see on the screen/paper:

```
3333    HILL    40    240
4444    DALE    30    300
```

Line 80 again sends the computer back to line 50 where it reads the dummy set of data. The following situation now exists:

Data: HILL 3333 40 6 DALE 4444 30 10 X 0 0 0
(Pointer) ↑

Variables:

N$	E	H	R	P
X	0	0	0	300

The computer then goes to line 55 of the program. It sees that H is now zero (finally) and branches to line 90, which is the END statement.

The program is finished.

Important Notes

1. Note that the dummy set of data is a *complete* set as we said it must be. Also note that the *type* of data values of the dummy set must match the type of data values of the valid sets of data; that is, it contains one alphanumeric value, followed by three numeric values:

30 DATA "X", 0, 0, 0
No other form would work. We could not use
30 DATA 0, 0, 0, 0
or
30 DATA "X", "X", "X", "X"

or any other combination: it must match the valid data in both the number of data values and the type.

2. It may seem odd to you at first but we can treat numbers as alphanumeric data, e.g., the employee number, as long as we will not be performing any arithmetic operations on them.

Since this is the case for employee number in our program, we could use E$ as the variable name for it. (It would then have to be DIMENSIONED on some computers.)

Since we are performing calculations using the hours and wage rate though, we cannot use alphanumeric variable names for them.

If any of the data items you are looking at are all numbers, I recommend that you always set them up in numeric variables whether or not you wish to use them in arithmetic expressions. It keeps life simpler.

Listing Portions of a Program As programs get longer, you will find that the listing of the program may not fit on the screen. There is an easy solution to this. Simply list part of your program by using the following as a guide:

LIST 55 will list only line 55
LIST 300-999 will list all lines from 300 to 999

(It is not necessary that lines 300 or 999 exist.)

Some computers use the format LIST -100 to list all lines from the beginning of the program, up to and including line 100. They would also use the format LIST 100- to list all lines from 100 to the end of the program.

Summary

ALPHANUMERIC or STRING data may be considered as anything that is not NUMERIC data; i.e., data with anything other than digits, a decimal point, and a sign.

Alphanumeric data can be included in DATA statements in much the same way as numeric data. (Some computers require that alphanumeric items be included in quotation marks within the DATA statements.)

Variable names for alphanumeric data consist of a letter of the alphabet followed by a $; e.g., N$. Most computers require a DIMENSION statement if the data length exceeds some minimum value (often ten).

REMARK statements are used to add comments to a program.

Exercises

9-1. The Invoice Exercise

Since we have now discussed flowcharting, you should draw a flowchart for this exercise. Since we started this program several chapters ago, this will be an exception to our rule of drawing the flowchart before we write our program.

Add the item names STAPLER and CHAIR to the two sets of data in the exercise.

Item Name	Item #	Price($)	Quantity sold
STAPLER	6217	10.00	5
CHAIR	3424	69.95	100

Output should now be:

(item name)	(item #)	(price)	(extended price)
STAPLER	6217	10.00	50
CHAIR	3424	69.95	6995

9-2. Input data:

Flight #	Destination	Fare($)	No. of passengers
195	ROME	900	150
246	PARIS	800	200
432	ATHENS	1000	300

Draw a flowchart, and then write a program that will calculate the total revenue for the airline from each flight.

Print out FLIGHT #, DESTINATION, NUMBER OF PASSENGERS, and TOTAL REVENUE for each flight.

For the first flight the output should be:

(flight #)	(destination)	(fare $)	(total revenue $)
195	ROME	900	135000

9-3. Input data:

Item name	Item #	Code	Price($)
ELECTRIC KNIFE	9447	SA	30.00
DISHWASHER	2468	LA	400.00
UTENSIL SET	8877	MI	25.98
TOASTER	975	SA	44.95
MICROWAVE OVEN	5432	LA	425.00

Assuming a tax rate of 5%, draw a flowchart and write a program to print out a report using the following format:

(item #)	(item name)	(code)	(tax)	(total price)
9447	ELECTRIC KNIFE	SA	1.5	31.5

9-4. Input data:

Customer #	Name	Opening Balance($)	Payment($)
8642	BUELL	4000	2200
5960	VALLIER	6500	2500
1122	SYVERSON	3100	1400
4321	HARE	7200	4000

Write a program that will print out the following for each customer.

(name)	(customer #)	(opening balance)	(closing balance)
BUELL	8642	4000	1800

(Note the slight change in sequence of data on the report vs. the input data.)

9-5. Input data:

Item Name	List Price($)	Cost Price($)
CLOCK	60.00	35.00
CAMERA	350.00	280.00
CALCULATOR	49.95	37.00
WATCH	149.50	75.00

Draw a flowchart and write a program that will determine the profit or loss for each item based on selling them at the list price, at a sale price of 10% off list, and at a sale price of 50% off list.

Output (for the first item) should be as follows:

(item name)	(profit at list)	(at 10% off)	(at 50% off)
CLOCK	25	19	-5
			(a loss)

10 Titles and Headings

Most reports or output that we use in business will include a title and/or headings. They prove to be very helpful of course and they also look more professional. In this chapter we will add these to our program.

New Vocabulary BUILT-IN function LINE OVERFLOW
COSMETICS

10-1. The Method

To this point we have not included a title or headings in our reports.

Let's add these to our last program to produce the following report:

```
          PAYROLL REPORT
NUMBER    NAME     HOURS     PAY
  3333     HILL       40      240
  4444     DALE       30      300
```

The flowchart for the program required to accomplish this is shown in Figure 10-1.

The required report could be produced by making a few changes to our program from the last chapter. It will now appear as follows:

Opening

```
5 REM SAMPLE WITH TITLE AND HEADINGS
10 DIM N$(10) (if required by computer)
20 DATA "HILL", 3333, 40, 6
30 DATA "DALE", 4444, 30, 10
35 DATA "X", 0, 0, 0
37 REM . . . E IS EMPL # . . . H IS HOURS
38 REM . . . R IS RATE . . . P IS PAY
40 PRINT " ", "PAYROLL REPORT"
(or use the TAB function)
45 PRINT
50 PRINT "NUMBER", "NAME", "HOURS", "PAY"
(or use the TAB function)
```

```
            60 READ N$, E, H, R
            65   IF H = 0 THEN 90
Loop        70   P = H * R
            75   PRINT E, N$, H, P
            (or use the TAB function)
            80 GOTO 60

Closing     90 END
```

Clear your workspace and type this program into the computer.

It helps in understanding how titles and headings are printed out, if we recall the general layout of a program. We suggested that a program has three sections, namely the "opening section", the "loop section", and the "closing section". Remember the loop section is from the READ statement to the GOTO

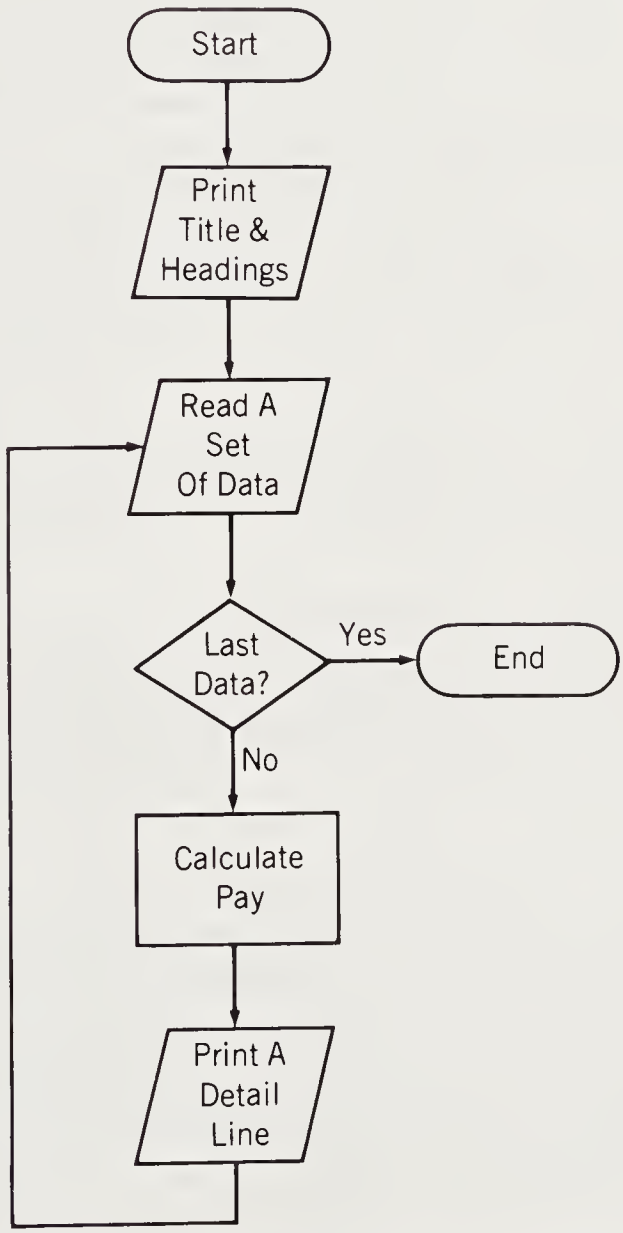


Fig. 10-1 Flowchart of Sample Program

statement, the opening section is that section before the loop, and the closing section is that part after the loop.

We stated that the opening and closing sections were only performed once by the computer, but the loop section is performed several times — in fact it is performed once for *each* set of data that you have in your program. For example, once for each person, each part, each customer, etc.

Because we only want the title and headings printed out *once*, at the top of the report, it seems logical that we should include the printing of the title and headings in the *opening* section of the program, since the opening section of the program is also at the top, and it is only performed once. Note how a title is printed (line 40):

```
40 PRINT "   " , "PAYROLL REPORT"
```

Any time words (also called LITERALS) are to be printed, as in a title or headings, these words must be enclosed in quotation marks.

In line 40 you will notice that there is a set of quotation marks before our title and that the quotation marks have a blank space between them. The purpose of printing this blank space is to center our title.

Thus, we tell the computer to print a blank space in zone 1 and our title "PAYROLL REPORT" in zone 2. This will center it (approximately), on the screen/page . If your computer has five zones, printing in the third one will center the title reasonably well. For any other number of zones you will have to experiment a little. A little trick that helps in centering titles is to use something like this:

```
20 PRINT "      PAYROLL REPORT"
```

Since the blank spaces are *inside* the quotation marks the computer will print them out as well, having the effect of moving the title over.

Note that if you wrote the above line as

```
20 PRINT      "PAYROLL REPORT"
```

it would not space the title at all; it would print it at the left-hand side of the screen/paper. The blanks must be within the quotation marks.

By the way, some computers allow you to omit the blank space in quotation marks for skipping zones, allowing you to simply include the *commas*. Line 40 would then appear as follows:

```
40 PRINT , "PAYROLL REPORT"
```

Try this on your computer and see what happens. If you receive an error message, you will have to use the quotation marks as in the original program above.

Note that if your computer has only two zones per line, you could use the TAB function (Chapter 3).

For example you might print the title by typing:

```
40 PRINT TAB(14); "PAYROLL REPORT"
```

Centering a Title Exactly To center a title exactly, proceed as follows:

a. Count the number of characters in the title itself. This must include blanks between words. For example, the title SALES ANALYSIS REPORT contains 21 characters: 19 letters and two blanks.

b. Subtract this number from the total number of print positions on a line on the screen/paper. This might be 22, 32, 40, 80, 120, 132 or some other number, depending on the computer system and/or the device being used. If it were 40 print positions per line, the difference would be 40 - 21 or 19 positions.

c. Since the result of the subtraction in **b** is the number of extra spaces on the line (after the title prints), we can center the title exactly by placing one-half that number of spaces before the title, and one-half after it.

In this case, if we place 19/2 or 9 spaces before the title, it will be exactly centered – or at least as exact as we are able to get since we cannot space one-half of a position. You may also decide to place the extra space before the title. Whether you use 9 or 10 blanks before the title, it will certainly appear to be centered to all but the most discerning eyes.

Line 45 of the program prints a blank line. The word PRINT with nothing after it accomplishes this. We include a blank line simply for readability, to make the report a little more pleasing to the eye.

Then in line 50 our headings are printed:

```
50 PRINT "NUMBER", "NAME", "HOURS", "PAY"
```

These will print in zones one to four, since as we said before the computer starts printing in zone one and continues to print in consecutive zones until it is finished. The commas control the printing in the various zones, remember. If your computer only has two zones, the headings will be typed on two lines instead of one. The output would appear as follows:

```
NUMBER    NAME
HOURS     PAY
```

This is too confusing. Use the TAB function to overcome this problem. We must tell it in the PRINT line how we wish it to tab. This is done as follows :

```
50 PRINT TAB(2);"NUMBER";TAB(10);"NAME";TAB(16);"HOURS";
TAB(24);"PAY"
```

Likewise, for the data itself we could try

```
75 PRINT TAB(2); E, TAB(9); N$; TAB(17); H; TAB(23); P
```

The cursor/print-head will move over to print position two and print the word NUMBER there, then to print position ten, and print the word NAME there. It will then go to print position 16 to print the word HOURS, and finally to print position 24 and print the word PAY.

In a similar manner the data will be spaced beneath the headings. You can experiment a little with different TAB values to line the data up with the headings more accurately.

Long Lines to Type? With some of the lines of our programs getting longer, you may find that as you type them in, they "overflow" onto the next line of the screen. This is all right as far as most computers are concerned, up to a point.

Some computers will automatically consider the continuation of the line as part of the original one. The limit of this continuation varies from computer to computer. Others permit the use of some continuation character, such as an "and" sign (&). If this is the case with your computer, at some appropriate point in your line (perhaps after a comma), type the continuation character and press RETURN (ENTER). Then complete your line and press RETURN (ENTER) again.

Important Note Occasionally a line we type may exactly fill a line on the screen. When this happens the computer automatically returns to the left-hand side of the screen.

If this occurs, you must still press the RETURN (ENTER) key so that the computer knows that you have finished the line. Otherwise, it thinks your next line is part of the previous one.

When you do press RETURN (ENTER), there will be a blank line on the screen but that is as it should be. It will not appear when you LIST your program.

Titles and Headings vs. Alphanumeric Data Titles and headings add a confusing aspect to programming for many. Some people think of them as a form of alphanumeric data and thus feel that they need variable names to contain them. This is not the case, however. They are indeed alphanumeric in nature but they do not *vary*; hence, they do not require variable names. Think of them as *alphanumeric constants*. Because they will not change or vary, the rules of the last chapter for alphanumeric data do not hold for them. That is why we simply place the words that we want printed as a title and/or headings in quotation marks.

The computer prints exactly (literally) what is *between* the quotation marks, which leads to such items being referred to as LITERALS. The quotation marks are not printed, only what is between them.

A literal, then, is any value enclosed in quotation marks that is to be printed out. This could include a title, headings, a label, or anything else that is to be printed.

Alphanumeric variables are only used for alphanumeric data that varies such as employee names, item names, etc.

Are you still confused? It is a difficult concept to grasp at first. Try re-reading this chapter if you are still a little uncertain about how it all ties together.

Cosmetics (Making our reports look better.) Want to get a little fancy for our report? Let's try underlining our title. To do this we would insert an additional line as follows:

```
42 PRINT " " , "_____ _____"
```

Notice that the layout of line 42 is exactly the same as the layout of the title line, except that a dash (—) has been substituted for each letter of the title. By the way, some computers require that you use the minus sign or equal sign for underlining, rather than a dash.

You will notice that the underlining actually prints on the line immediately *below* the title, whereas a typewriter would print an underline closer to the title itself. This is not an inconvenience for most people, and it still makes the title stand out.

In a similar manner, you may wish to underline your headings:

```
55 PRINT "————————","————————","————————","————"
```

Some people prefer one long line as:

```
55 PRINT "————————————————————"
```

Be a little creative in your report layout. Make it readable and attractive as well.

Longer Headings Sometimes headings are too long to fit into a zone. Remember that the zones are usually only 20 positions long at most, depending on the computer being used. If the headings that you wish to use are too long, you have at least five choices:

a. Abbreviate the headings; e.g., EMPL. NO. for EMPLOYEE NUMBER.
b. Print them on more than one line; e.g.,
EMPLOYEE
NUMBER
c. Use the TAB function in the PRINT statement.
d. Use semi-colons in the PRINT statement.
e. A combination of the above.

Titles and headings are very important on reports, of course, and should be included in every case. Otherwise, a report appears to be simply columns of data on a sheet of paper. Computers don't need titles and headings but humans certainly do. Get in the habit early of including them in all your programs.

10-2. Dating Reports

In making reports more acceptable to a reader, it is important to include the date on which they are printed. This should always appear on the first page of the report, and usually in the top right corner.

Many computers have the capability of printing the date for us when we request it. This is because they have what is called a BUILT-IN function or LIBRARY function. This function contains the date and it can be called forth at any time to be printed on the report. Almost all larger computers have this feature but most of the smaller ones do not. Consult your manual or your instructor to see if yours does. If it doesn't, you could always add the date to the title

line or print it on a separate line. Each time that you run the program, however, you must remember to change the date.

In the case of the built-in function, it always retrieves the current date for you automatically from the computer system.

Summary

Titles and headings are added to our programs using a PRINT statement and enclosing the title or headings within double quotation marks. The TAB function is also helpful in providing accurate spacing.

You are able to skip any given print zone by "printing" a blank space in that zone.

If your computer has a BUILT-IN FUNCTION for the date, it can be easily added to any of your reports.

Exercises

10-1. The Invoice Exercise:

Input data: (item)	(item #)	(price)	(quantity)
STAPLER	6217	10.00	5
CHAIR	3424	69.95	100

Add the title "PINE RIDGE OFFICE SUPPLIES" and the following headings: (You may have to abbreviate them or print them on two lines.)

ITEM NAME	ITEM NUMBER	PRICE	VALUE
STAPLER	6217	10	50

Alter your flowchart for this exercise to reflect this addition.

10-2. The following employee data is for the MOOSE JAW APPAREL COMPANY:

Name	Dept.	Hourly Rate($)	Bonds($)	Union Dues($)
ANDERSON	14	10	10	5
COXFORD	12	8	20	4
DOUGLAS	12	7	20	6
PECK	14	12	0	5
POWELL	12	9	14	6

Draw a flowchart and write a program to calculate net pay for each employee. Assume that a work week is 35 hours, that deductions shown (bonds and union dues) are per week, and that there is a tax rate of 20% on gross pay.

Print out a report of the following form:

MOOSE JAW APPAREL COMPANY

NAME	DEPT. #	TOTAL DEDUCTIONS (INCL. TAX)	NET PAY
ANDERSON	14	85	265

(Tax for ANDERSON is 20% of $350, or $70.)

11 Columnar Totals, Counts, and Averages

Often in reports, we wish to include totals of columns of data (columnar totals), counts of employees, and averages. In this chapter we will address each of these operations in turn.

New Vocabulary COUNTS COLUMNAR TOTAL
INITIALIZATION

11-1. Calculating a Columnar Total

Let's change our program from the previous chapter so as to produce a report that looks like this:

PAY REPORT

NUMBER	NAME	HOURS	PAY
3333	HILL	40	240
4444	DALE	30	300
TOTAL HOURS		70	(columnar total)

To produce a total we can simplify the procedure if we recall the three sections of a program as discussed in earlier chapters. These sections were the opening, the loop, and the closing sections.

In order to keep a total and print it out on our report, we will use these three sections. Note that the total is the only thing that has been added to the report from the last chapter.

To produce any total, there are three steps to keep in mind. Each of these three steps is performed in a different section of the program. We must do the following:

a. The total must be *initialized* (started off at zero) in the opening section of the program. (Again this must be qualified – some computers do not require that this step be included. Since some do, though, and since some other programming languages do, many people get in the habit of including it anyway.)

If this is not necessary for your computer, you need only be concerned with points **b** and **c**.

b. The individual value to be totalled (which is read in or calculated) must be added to the running total within the *loop* section.

c. The total is printed out in the *closing* section.

The flowchart for the required program is shown in Figure 11-1. Note the "sections" of the flowchart corresponding to the sections of the program.

Our program will be as follows: (Clear your workspace and type it in.)

```
          5 REM . . . PROGRAM USING TOTALS
          10 DIM N$(15)
          12 REM . . . THE TAB FUNCTION MAY BE USED
          15 T = 0 (added – initializes total to zero)
          20 DATA "HILL", 3333, 40, 6
          30 DATA "DALE", 4444, 30, 10
Opening   35 DATA "X", 0, 0, 0
          40 PRINT " ","PAYROLL REPORT"
          (or use the TAB function)
          45 PRINT
          50 PRINT "NUMBER", "NAME", "HOURS", "PAY"
          (or use the TAB function)
          55 PRINT

          60 READ N$, E, H, R
          65   IF H = 0 THEN 85 (changed)
Loop      70   P = H * R
          72   T = T + H (added)
          75   PRINT E, N$, H, P
          (or use the TAB function)
          80 GOTO 60

          85 PRINT (added)
Closing   87 PRINT "TOTAL HOURS"; T (added)
          90 END
```

Let's analyze our program with the changes we have made to see how it produces our total for us. While we do this keep a copy of the program in front of you for easy reference.

Notice that lines 15, 72, 85, and 87 have been added and line 65 has been changed.

Explanation Recall that the opening section of the program (lines 5 to 55 inclusive) is performed once. That is why we included the printing of the title and headings in this section. We only want them printed once as well. A common error of beginning programmers is to include the printing of title and headings within the loop section of the program. This results in the title and headings being printed above *every* line of your output.

After the opening section, the program enters the loop section where the first set of data is read in. You already know that the purpose of line 65 is to test for the end-of-data condition.

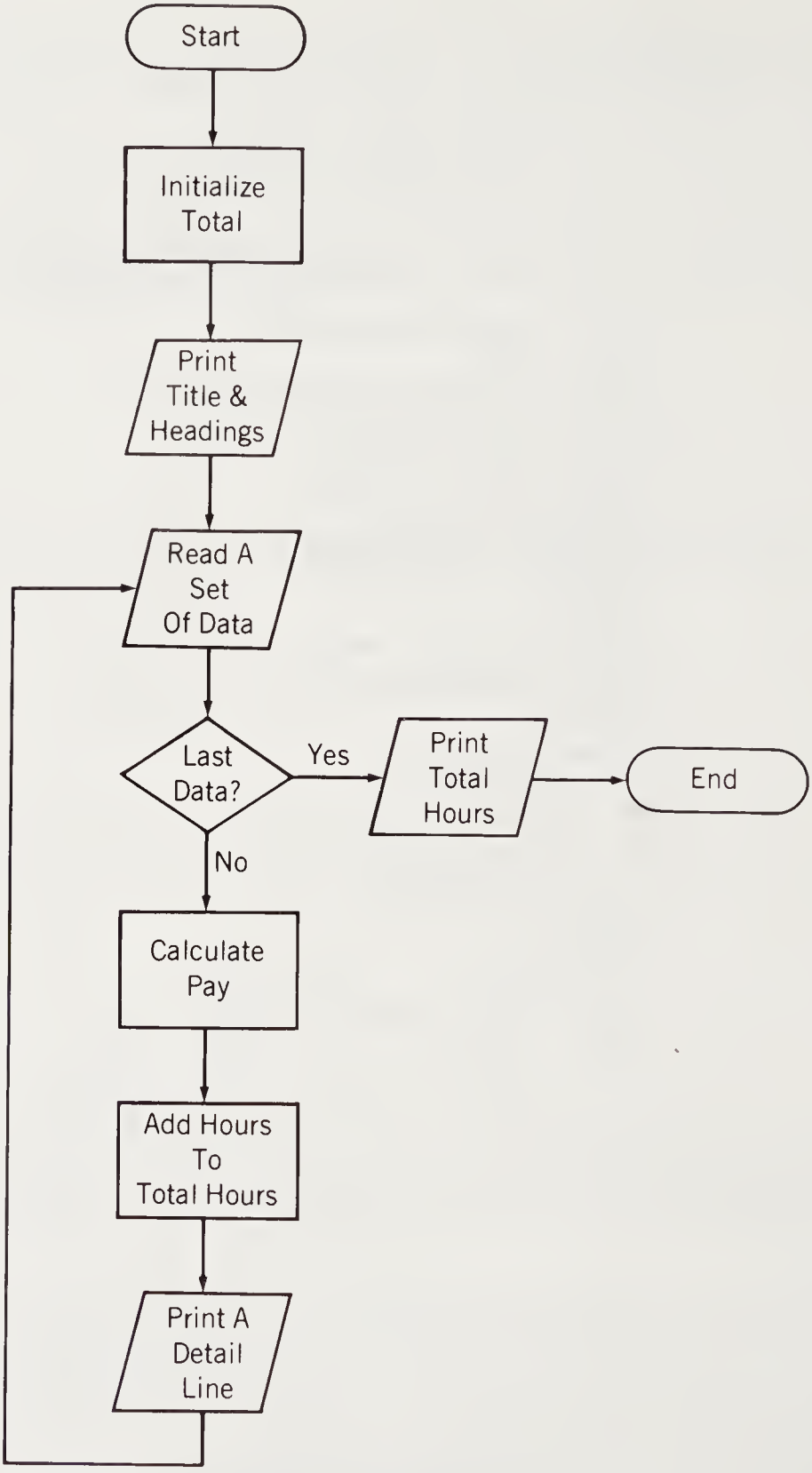


Fig. 11-1 Flowchart of Sample Program

Next, pay is calculated (line 70); then, line 72 adds the employee's hours to the total hours.

The form of the calculation to do this differs from what we have seen so far:

```
72 T = T + H
```

This would not make sense at all to us if we were to think of our algebra. (If you have not studied algebra or if you have forgotten most of what you did study, the concept might be even easier for you to accept.) Think of the equal sign as meaning "is replaced by" instead of "is equal to" in this case.

Recall that when we discussed calculations in an earlier chapter, we mentioned that the computer looks for the equal sign, performs whatever is to the right of it, and places the answer into the variable on the left-hand side of it. This being the case, the first thing that the computer does is to add T and H together.

The *first* time through the loop, the H is the hours for HILL (40) and the value in T is zero, since it was set to zero in line 15. Thus, the computer adds the 0 and the 40 together in its circuits, and places the result in the variable on the left-hand side.

The new total replaces the zero that is in the variable T, which is fine, since we are simply using T to keep the running total of the hours.

The fact that the variable T is on both sides of the equal sign is what results in our obtaining an accumulated total, as we will soon see. This is a confusing concept to many and this section should be reviewed, until it makes sense to you. Either that or accept it on faith for now.

The computer then goes through the remainder of the loop, printing out the detail line for the first employee (HILL), then line 80 sends it back to line 60 to read the next employee (DALE).

After checking for the dummy data,

```
65 IF H = 0 THEN 85
```

it goes on to line 70 and calculates DALE's pay:

```
70 P = H * R
```

Next it encounters line 72 again.

```
72 T = T + H
```

This time the computer finds that the H (DALE's hours) contains 30, and the variable T contains the 40 from the last time through the loop.

It adds the T and H together as before (40 + 30) to arrive at a new total of 70 and places this latest total back into T (the variable on the left-hand side of the equal sign remember). We, thus, have the running total we wanted.

The computer goes on to line 75 to print the details for DALE, and line 80 sends it to the top of the loop again (line 60).

This time the computer reads the dummy set of data; hence, N$ will contain X, and E, H and R will each contain a zero.

Thus, when the computer checks H,

```
65 IF H = 0 THEN 85
```

it finds that the condition is true (i.e., H is zero). It will then branch to line 85, as line 65 says it should when H is equal to zero. This means that the computer "escapes" from the loop, and branches to the closing section of the program.

The first thing that the computer does in the closing section is to print a blank line:

```
85 PRINT
```

Then, as per line 87, it prints the words "TOTAL HOURS" and whatever value it finds in the variable T

```
87 PRINT "TOTAL HOURS "; T
```

In this case it finds our total of 70.

With this step, the program ends. (It's probably a good idea to go through this section again.)

General Format for Keeping a Total Note that any variable names could have been used in our program to keep a total. However, a total will always have the form :

T = T + A

where A is *what* is being totalled, and T is *where* the total is being kept.

Thus, M = M + Q would be a total of whatever Q happened to be and would be kept in the variable named M. All the rules we discussed would have to be followed, of course, for the total M. Namely, it would have to be initialized in the opening section of our program (using M = 0), "added to" in the loop (M = M+Q), and printed out in the closing section.

Multiple Totals How do you think we would change our program so that it would keep a total of the combined pay for all employees, as well as the total hours?

Our output would then look like this:

```
                PAY REPORT

NUMBER          NAMES      HOURS      PAY
3333            HILL         40       240
4444            DALE         30       300

TOTAL HOURS      70
TOTAL PAY       540
```

This operation will require more lines and variables in our program but the procedure is still identical to what we have described above.

To keep track of the two totals, we will need two separate variable names

to hold them: T for total hours and T2 for total pay. Therefore, we must initialize both of them in the opening section of the program; i.e.,

```
15 T = T2 = 0
```

This method is shorter than using two separate lines to zero the totals; however, (sure are a lot of "howevers" in this programming business, aren't there ?), . . . however, not all computers allow this form. You will have to try yours and see what happens. If yours does not permit it, you will have to zero the totals on separate lines:

```
15 T = 0 (as before)
17 T2 = 0 (added)
```

Many computers allow yet another alternative, namely, separating individual statements with colons; e.g.,

```
15 T = 0 : T2 = 0
```

(Only one line number is necessary.)

Then, in the loop section, we will add the hours to T and the pay to T2.

```
72 T = T + H (as before)
73 T2 = T2 + P (added)
```

It should be stressed, in the case of hours, that they are read in when a set of data is read:

```
60 READ E, N$, H, R
```

The pay on the other hand must be calculated before it can be added to the total pay T2.

What this means is that the statement that calculates total hours can be placed anywhere in the loop and still produce the correct results for us.

But since the pay must be calculated before it can be added to a total, the line that calculates total pay

```
73 T2 = T2 + P (added)
```

must be *after* the line which calculates the pay for the individual; i.e., after

```
70 P = H * R (as before).
```

If you place it before this line your output will be in error.

In the closing section of the program, we will print out the totals:

```
87 PRINT "TOTAL HOURS    "; T (as before)
88 PRINT "TOTAL PAY    "; T2 (added)
```

Note the spaces inside the quotation marks after the labels. These ensure that the totals themselves do not end up cramped against the labels. The semi-

colons cause the values in T and T2 to be printed closer to the labels than commas would have.

Why not experiment a little with these to see what happens.

11-2. Keeping a Count

Counting can be considered to be a "special total", namely, a total of ones. Thus, if we start a total at zero, and add one to that total each time we encounter it, then we are in effect counting by ones.

Since what we usually want to count is the number of employees we read in our program, or the number of items, or the number of customers, etc., what we really want to do is to count the number of times that the computer performs the *loop section of the program*. That is, each time that the computer goes through the loop, it reads in another employee, item, etc.

Therefore, what we will do is put the counter inside the loop section of the program. Then, each time the program goes through the loop, it will add one to the count.

In the opening section we will include a statement such as C = 0 which will zero the counter for us, in the same manner as we zeroed an ordinary total.

In the loop section we would then include a statement of the form:

```
C = C + 1
```

The C is the "where the total is being kept" remember, since it is repeated on both sides of the equal sign, and the 1 is the "what is being totalled".

In the closing section we would have a statement such as:

```
PRINT "COUNT IS ", C
```

Thus, the count is exactly like a total in its form, since as we said, it is really just a special total.

The variable name used for a count is optional, of course, but most often a C is used. If you have already used C in your program for something else, a K is often the next choice. Of course, any valid variable name may be used.

Numbering Output Lines Total counts are often printed at the bottom of a report as we have described above, but we also may want to number our output lines; i.e.,

```
1. .....
2. .....
3. ..... etc.
```

Our count will help us out here.

To accomplish this, the C (or whatever variable you used for the count) must be the first variable in the PRINT statement which prints these lines; e.g.,

```
80 PRINT C; N$; E; R
```

(See the example in the next section.)

11-3. Calculating an Average

Quite often we wish to calculate an average, whether it be the average age of a group of people, the average salary of a group of workers, the average mark of a group of students, etc.

In each case the process is the same - we add up the numbers involved and divide by the number of values there are. For example, the average of the numbers 6, 9, and 15 is ten. We add the numbers together and in this case divide by three.

If we are to calculate an average in a program, we must perform the equivalent of the above steps within the program.

From the above discussion, we see that calculating an average involves keeping both a total and a count.

Assume, for example, that we want to calculate the average pay of our employees (HILL and DALE). As humans, we would add the pay for each of the two employees together and divide by how many of them there were (in our case, two). Hence, the average pay of the employees would be 240 + 300 divided by two, or $270.

In order to do this in a program, we need to keep the total pay and a count separately. Then, we will divide the total by the count, giving the average pay, in the same manner as we would do ourselves. The flowchart for such a program is shown in Figure 11-2.

We must change our previous program as follows to accomplish this:

```
                5 REM . . . CALCULATING AVERAGES
                7 REM . . . THE AVERAGE IS CALCULATED IN THE
                8 REM . . . CLOSING SECTION OF THE PROGRAM.
                10 DIM N$(10)
                15 T = 0 (initializes total pay to zero)
                17 C = 0 (initializes count to zero)
                20 DATA "HILL", 3333, 40, 6
                30 DATA "DALE", 4444, 30, 10
Opening         35 DATA "X", 0, 0, 0
                40 PRINT " ","PAYROLL REPORT"
                45 PRINT
                50 PRINT "NUMBER", "NAME", "HOURS", "PAY"
                (or use the TAB function or semi-colons)
                55 PRINT

                60 READ N$, E, H, R
                65    IF H = 0 THEN 85
                67    C = C + 1 (adds 1 to counter)
Loop            70    P = H * R
                73    T = T + P (adds pay to total pay)
                75    PRINT E, N$, H, P
                (or use TAB or semi-colons)
                80 GOTO 60
```

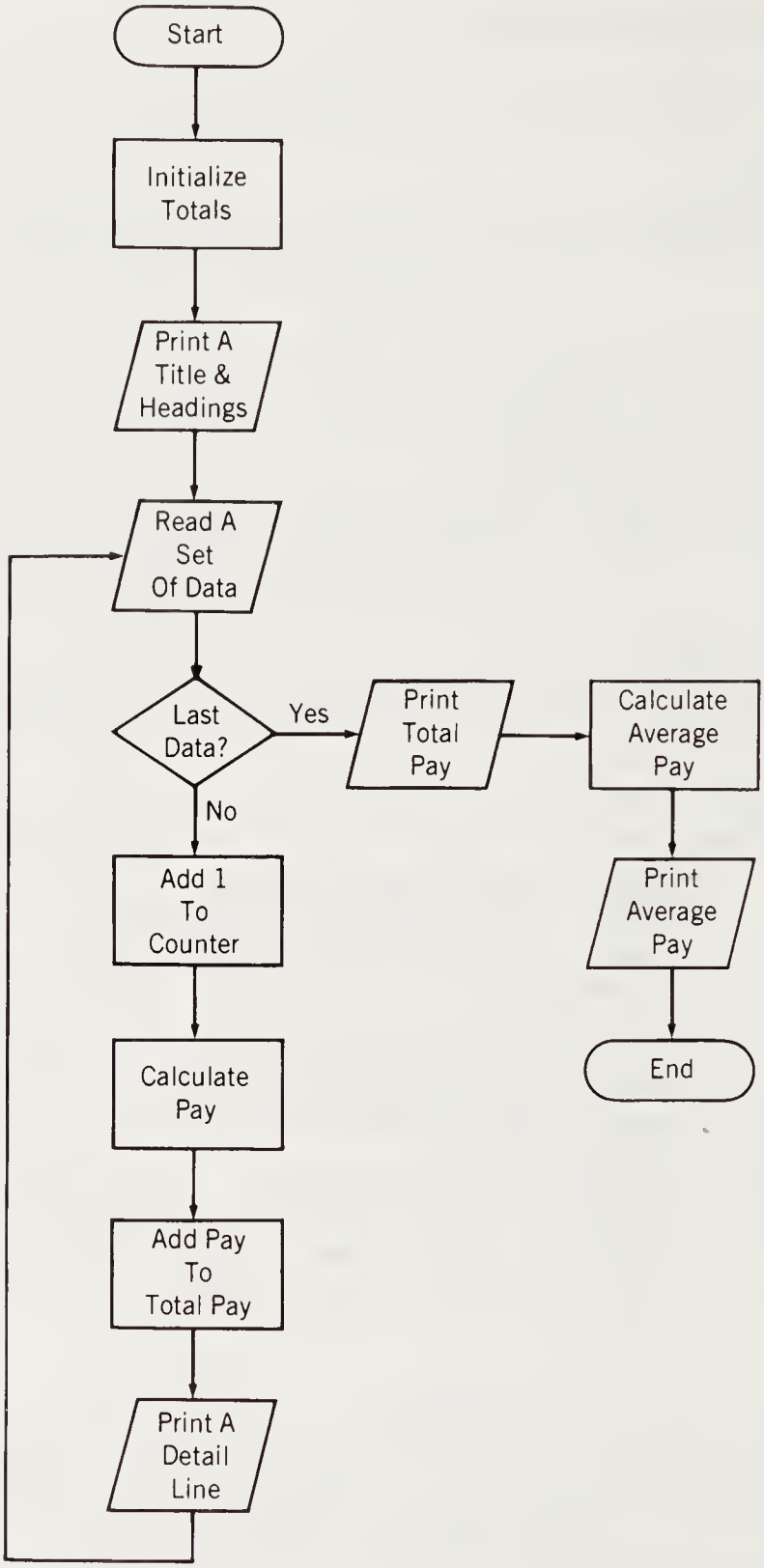


Fig. 11-2 Flowchart of Sample Program

```
            85 PRINT
            86 REM . . . CALCULATE AVERAGE IN THIS SECTION
            87 PRINT "TOTAL PAY "; T
Closing     88 A = T/C
            89 PRINT "AVERAGE PAY "; A
            90 END
```

We are forgetting total hours in this example to make the program a little shorter.

Note that lines 15 and 17 zero T and C. Thus, T will still be our total pay, and C will be our count of the number of employees.

Line 67 is where we add 1 to the counter each time the program passes through the loop; that is, each time an employee is read.

Then, in the closing section of the program, line 88 performs the calculation of the average for us. That is, it divides the total pay, T, by the number of employees, C. This is precisely how we would do it, if we were calculating the average pay by hand. Line 89 prints out the average pay just calculated.

Make the necessary changes to the program you have in your computer, RUN it, and confirm that the program does in fact do as we expect it to.

Averages are calculated for a multitude of things, but the procedure will always be the same as that in the above example. The only difference will be in the particular variable names chosen for the total, the count, and the average itself.

Numbering Output Lines We mentioned that we could use our counter to number our output lines. To do this in the above program, simply change line 75 as follows:

```
75 PRINT C; E, N$, H, P
```

The semi-colon after the C ensures that the headings still line up reasonably well. Make this change and RUN the program to check the results.

RENUMBER Command Now that we are squeezing more and more lines between existing lines, we may get to the point where we run out of possible numbers to add. For example, maybe we have already added lines 75, 76, 77, 78, and 79 to a program. We now want to squeeze yet another line in between 78 and 79, but there is no room.

Some versions of BASIC allow the use of decimals in line numbers. If yours does, you could add line numbers such as 78.5 and 78.6 between lines 78 and 79.

Some, that do not allow decimals, permit the use of a RENUMBER (RENUM, for short) or a RESEQUENCE command. What this command does is renumber the lines of your program. Commonly, it will start the new numbering at 10 and use an increment of 10 for the rest of the program; i.e., 10, 20, 30,

40, etc. It will also automatically adjust all statement numbers referenced in GOTO statements, IF statements, etc.

If your computer allows the use of the RENUMBER command, just type RENUM and press RETURN (ENTER). No line number is used since it is a command.

That's all there is to it. It's done in a flash. Then LIST your program to see the new line numbers and add lines as desired.

If your computer does not permit use of this command, you must retype several lines of your program, changing line numbers as necessary. Take special note of any GOTO statements or IF statements that may have to have line numbers in them changed.

Summary

Totals can be tricky. However, if you keep in mind the three sections of a program and remember that one operation must be performed in each section, then they should not cause you too much grief. Remember:

a. Start them off at zero in the opening section.

b. Add to them within the loop section.

c. Print them out in the closing section.

A count is often desired in reports or programs. A count will always be of the form C = C + 1.

The same rules that must be kept in mind for totals apply to counts as well. As we have said, a count is really just a special total.

To calculate an average, we use a total and a count and then divide the total accumulated by the count of the number of items, persons, etc., involved.

Exercises

11-1. The Invoice Exercise

Input data

Item	Item #	Price($)	Quantity
STAPLER	6217	10	5
CHAIR	3424	69.95	100

Calculate the average value of the items ordered (i.e., the average of the extended prices). Print this average, and a count of the number of products on the invoice, below the main report.

Change your flowchart to show these additions.

Your output should look something like this:

PINE-RIDGE OFFICE SUPPLIES

ITEM #	ITEM NAME	PRICE($)	VALUE (extended price)
6217	STAPLER	10	50
3424	CHAIR	69.95	6995

AVERAGE VALUE 3522.5
NO. OF ITEMS 2

11-2. A class of students attained the following marks on a test:

RENNIE	86
SIMPSON	75
DEWEY	70
BENDALL	90
SHUTTLEWORTH	80

Draw a flowchart and write a program that will read in the data and calculate the class average.

Print a report with a title and headings and list the input data on it. Print the class average (with a label identifying it) below the student names and marks.

11-3. Given the following employee data for the WOLFE ISLAND SHIPPING CO.:

Employee	Year of Birth	Year of Hire	Division
ANDERSON	1940	1965	B
NEUN	1952	1972	A
EDWARDS	1947	1977	A
PERKINS	1960	1982	B

Draw a flowchart and write a program to print a report of the following format:

WOLFE ISLAND SHIPPING COMPANY

NAME	DIVISION	AGE	YEARS OF SERVICE
ANDERSON	B	44	19

. . .
COUNT OF EMPLOYEES 4
AVERAGE YEARS OF SERVICE 10
AVERAGE AGE OF EMPLOYEES 34.25
(The sample figures assume the year is 1984.)

11-4. Given the following results of a typing test:

Name	Time for Test (mins.)	Words Typed	Errors
HALL	6	525	8
MATHER	9	900	9
WOO	9	800	9
PRENTICE	6	750	4
NEWMAN	8	820	8

Draw a flowchart and write a program to produce the following type of report:

TYPING CLASS

NAME	TIME FOR TEST	WORDS PER MINUTE
HALL	6	63.5

. . .

CLASS AVERAGE IS 79.2 W.P.M.

Assume that for each error, you must subtract 3 w.p.m. from the initial result.

11-5. Given the following temperatures (in Celsius) for a given week:

MONDAY	28
TUESDAY	24
WEDNESDAY	30
THURSDAY	32
FRIDAY	25
SATURDAY	20
SUNDAY	22

Draw a flowchart and write a program to print out the input data as given as well as the average temperature for the week. Have the program count the days (do not just include a 7 in your program) and print the average below the daily temperatures.

12 Selection of Data (The IF Statement)

This chapter allows us to treat some data one way, and other data another way. This is referred to as conditional branching.

New Vocabulary	IF-THEN-ELSE statement	LOGICAL SYMBOLS
	CONDITIONAL branching	SELECTION

12-1. Selecting Specific Data

To this point we have performed the same operations for all sets of data read in a program. In other words, we calculated the pay for *all* employees, added *all* employees into the department total, printed *all* employees out, etc. (For simplicity we have only used two sets of data so far.)

What if we wanted to select certain employees based on some criterion? The IF statement will allow us to do this. Recall in Chapter 7 that we used the basic IF statement in our end-of-data test. Some computers use the form:

IF condition THEN GOTO line number

We also mentioned that, for most versions of BASIC, the GOTO is optional; hence, they would permit the form:

IF condition THEN line number

Because the program will branch to another line of the program if a certain condition is true, the IF statement is referred to as a CONDITIONAL BRANCH statement. (Contrast this with the GOTO statement which we said earlier was an UNCONDITIONAL BRANCH statement. It caused a branch in the program in all cases, not just under certain conditions.)

Let's assume we have three employees, as listed below:

Name	Hours worked	Wage rate($)
Hill	40	6.00
Dale	30	10.00
Bell	15	8.50

Let's also assume that we want a pay report. We will forget the title and headings, the total pay, and average pay for this example in order to simplify the program somewhat. However, we are going to be more selective in this program, since we only want to print out those employees who earn $7.00 per hour or more.

The flowchart for the program is shown in Figure 12-1.

The program would be as follows: (Clear your workspace and type it in.)

```
5 REM . . . PROGRAM TO SELECT DATA
8 PRINT "NO.   NAME   HOURS   PAY"
```

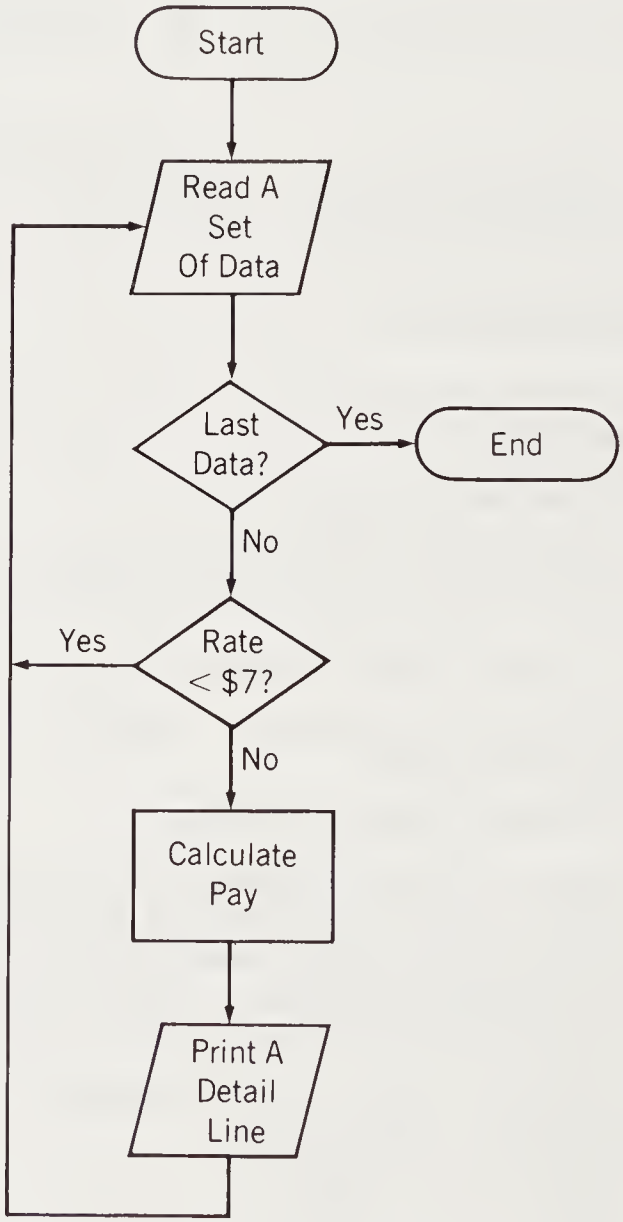


Fig. 12-1 Flowchart of Sample Program

```
10 DIM N$(10) (optional on some computers)
20 DATA "HILL", 3333, 40, 6
30 DATA "DALE", 4444, 30, 10
40 DATA "BELL", 5555, 15, 8.5
45 DATA "DUMMY", 0, 0, 0
50 READ N$, E, H, R
60    IF H = 0 THEN 100
70    IF R < 7 THEN 50 (added)
75    P = H * R
80    PRINT E, N$, H, P (or use a TAB function)
90 GOTO 50
100 END
```

Let's analyze the program. The only new line is line 70. (Note that the symbol < means "is less than".)

You will notice that line 70 immediately follows the line that tests for end-of-data. Thus, once the computer has established that the set of data that has just been read is not the dummy set, it immediately checks our wage rate criterion. That is, we do not want to print the name of any individual earning less than $7.00 per hour.

Note that we have asked the question "in the reverse" in the program. In other words, we could ask about the hourly rate in either of two ways:

a. IF H < 7 (as we did)

or

b. IF H >= 7 (>= means greater than or equal to)

The advantage of using the first method is that the program becomes a little simpler to write. You may have to accept my word on that for now, but you will soon discover why.

So, when we have a question, the object of which is to eliminate some of our data, it is often best to put it in such a way that the IF statement asks about the data which we do *not* want. In the above example, we did not want people earning less than $7.00 per hour, so we worded our IF statement that way. Line 70 in the program then has the effect of asking if the individual makes less than $7.00 per hour. If such is the case, the program will branch back to line 50 as it is told to do. What this means is that it branches back to read the next employee, *without* going to line 80 to print out the details for the employee just read in. This is exactly what we want.

KEY POINT — THE IF STATEMENT CAUSES A BRANCH TO ANOTHER LINE IN THE PROGRAM IF THE CONDITION IN THE IF STATEMENT IS SATISFIED; BUT, IF THE CONDITION IS NOT SATISFIED, THE PROGRAM WILL SIMPLY CONTINUE TO THE LINE FOLLOWING THE IF STATEMENT.

In the program above the first employee read in is HILL. HILL is earning $6.00 per hour. When the program encounters line 70, it looks at the hourly

rate and determines that it is less than $7.00; thus, the condition in the IF statement is satisfied. Therefore, the program will branch back to line 50. Since the program skips the PRINT statement, HILL is not printed out.

It now reads the next employee (DALE), falls down to line 60 to test for end-of-data, finds that it is not the last set of data, and so continues to line 70 which is our test of hourly rate. This time in doing the comparison, the program discovers that the hourly rate is not less than $7.00 per hour, so it will not branch to line 50 . Hence the computer "falls down" to the next line below the IF statement where it will calculate pay

```
75 P = H * R
```

It drops to line 80, where it prints out the details for DALE, and then falls to line 90 which sends it back to line 50 to read the next employee in (BELL). The same argument holds for BELL that did for DALE since the employee is also making more than $7.00 per hour. Thus, BELL will be printed out too.

The program then goes back to the READ statement once more, reads the dummy set of data, and because of the end-of-data test in line 60, branches to line 100 and ends.

Only two employees will be printed out in this program (DALE and BELL).

The IF statement has allowed us to select certain of our input data from those presented. In this example, we have made our selection based on a single criterion (hourly wage rate); however, it is possible to select data for printing (or for calculations) based on several criteria. Often this would involve more than one IF statement, and indeed it is possible to write some very complex programs using IF statements.

Let's look at another example. Suppose we want to calculate raises for our employees based on their current pay rate. Those earning $8.00 an hour or less are to receive a 12% raise while those earning more than $8.00 an hour are to receive 9%.

Also, assume that the company wants to keep a total of how much the raises in each category are going to cost it. (Just add the hourly raises together. See lines 74 and 82 of the following program.)

A report is to be printed that lists employee number, name, hourly wage, and the amount of the raise for each employee. It will also print the total raise by category below the main report.

The flowchart for this program is shown in Figure 12-2.

A program to accomplish this would be:

```
5 REM . . . PAY RAISES PROGRAM
6 REM . . . T1 IS TOTAL OF CATEGORY 1
7 REM . . . T2 IS TOTAL OF CATEGORY 2
8 REM ******************************************************
10 DIM N$(10)
15 T1 = T2 = 0 (you may need separate lines)
20 DATA "HILL", 3333, 40, 6
30 DATA "DALE", 4444, 30, 10
```

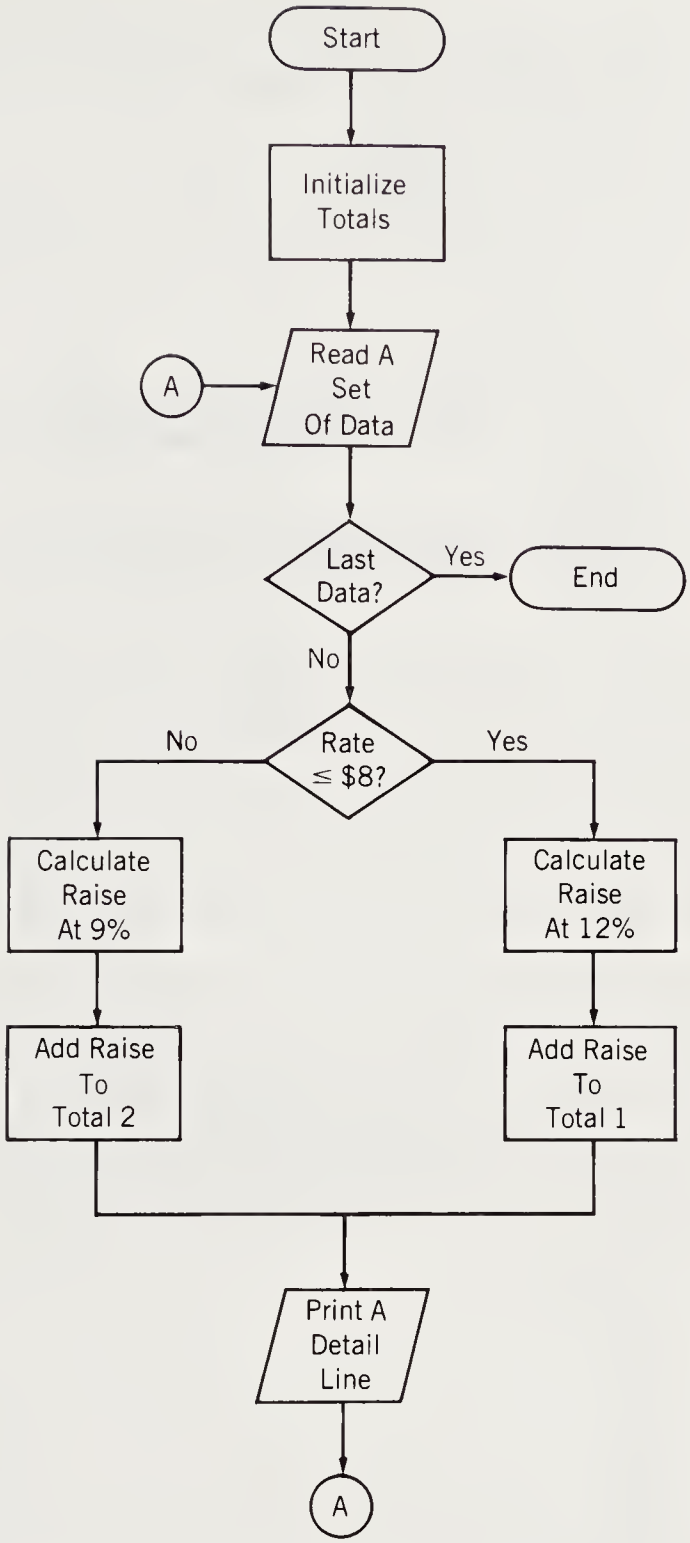


Fig. 12-2 Flowchart of Sample Program

```
40 DATA "BELL", 5555, 15, 8.5
45 DATA "DUMMY", 0, 0, 0
47 PRINT "NO.   NAME   HOURS   RAISE"
50 READ N$, E, H, R
60   IF H = 0 THEN 100
70   IF R <= 8 THEN 80
72   A = R * 0.09
74   T2 = T2 + A
76   GOTO 95
80   A = R * 0.12
82   T1 = T1 + A
95   PRINT E, N$, H, A (or use a TAB function)
96 REM . . . TOTALS FOR RAISES ARE OF HOURLY RATES
97 GOTO 50
100 PRINT
110 PRINT "TOTAL OF RAISES - CATEGORY 1 : "; T1
120 PRINT "TOTAL OF RAISES - CATEGORY 2 : "; T2
130 END
```

When the program reaches line 70, it looks at the hourly rate. If it is $8.00 per hour or less, the program branches to line 80 and calculates a raise at 12%. Otherwise, it continues to line 72 and calculates the raise at 9%. It also adds the raise calculated into the appropriate total, either T1 or T2.

In either case, the particulars for each employee are printed out. (Note that whichever path is taken, line 95 is finally reached.)

The totals by category are printed beneath the employee data.

Here is another example. Assume that input data consists of a number of sales records for a company. Each record contains a branch number and a sales figure($) associated with that branch. To keep the problem a little simpler, let's say that the company only has four branches (numbered 1 through 4).

The sales figures included with the branch number represent sales for individual salespersons, so there will be several sets of data for each branch.

These sets of data (also called RECORDS) will be in random order. For example, the input data might be:

Branch #	Sales ($)
3	4000
2	600
2	7000
1	6000
4	900
3	450
2	1400
etc.	

The requirement is to add together total sales by branch. Therefore, in our program we must first check each record to see what branch the sales are asso-

ciated with and add those sales to a total sales figure for that particular branch. We will, thus, need four separate totals.

The following program will do what we want. (Clear your workspace and type it in.)

Note that I have departed from our usual rule of one set of data per DATA statement. If you have stuck with me to here you can handle this with no problem at all. Right? Recall the key point that data must be in the correct sequence. The pointer will be moved to the proper point in the data each time that some is read.

```
10 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
12 DATA 3, 4000, 2, 600, 2, 7000
13 DATA 1, 6000, 4, 900, 3, 450
14 DATA 2, 1400, 0, 0
15 T1 = T2 = T3 = T4 = 0 (you may need separate lines)
17 REM T1 IS TOTAL FOR BRANCH 1, T2 FOR BRANCH 2, ETC.
20 READ B, S
30    IF B = 0 THEN 110
40    IF B = 1 THEN 80
50    IF B = 2 THEN 90
60    IF B = 3 THEN 100
70    T4 = T4 + S
75 GOTO 20
80 T1 = T1 + S
85 GOTO 20
90 T2 = T2 + S
95 GOTO 20
100 T3 = T3 + S
105 GOTO 20
110 PRINT
120 PRINT "TOTALS ARE :"
125 PRINT " BRANCH 1 : " ; T1
130 PRINT " BRANCH 2 : " ; T2
135 PRINT " BRANCH 3 : " ; T3
140 PRINT " BRANCH 4 : " ; T4
150 END
```

The output will look like this:

```
TOTALS ARE :
  BRANCH 1: 6000
  BRANCH 2: 9000
  BRANCH 3: 4450
  BRANCH 4: 900
```

Notice what we have done. The IF statements in lines 40, 50, and 60 test for branches 1, 2, and 3 and send the computer to a different line of the program in each case. Branch 4 on the other hand need not be tested for, since if the branch number being examined is not one of the first three branches, it

must be branch 4. (We will assume that there are no mistakes in our data for now.)

Since the program will not be sent to another line number if the branch involved is number 4, the program will "fall down" to line 70 and the sales will be added to the total sales for branch 4.

Depending upon the number of the branch involved, the sales will be added to the correct total. In each case, the computer is then sent back to line 20 where it reads the next set of data.

When it reaches the dummy set of data, it will branch to line 110 and print out the totals.

Finis!

Data Errors What if we did want to allow for errors in our data? What if, for example, a branch number of 6 was accidentally keyed in as data? (The only valid branches are 1 to 4.)

A slight change in our program will detect invalid branch numbers and point these out to us. The program will now appear as follows (Note the addition of lines 63, 65, and 67.):

```
10 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
12 DATA 3, 4000, 2, 600, 2, 7000
13 DATA 1, 6000, 4, 900, 3, 450
14 DATA 2, 1400, 0, 0
15 T1 = T2 = T3 = T4 = 0 (you may need separate lines)
17 REM T1 IS TOTAL FOR BRANCH 1, T2 FOR BRANCH 2, ETC.
20 READ B, S
30    IF B = 0 THEN 110
40    IF B = 1 THEN 80
50    IF B = 2 THEN 90
60    IF B = 3 THEN 100
63    IF B = 4 THEN 70 (added)
65    PRINT "INVALID BRANCH NUMBER "; B; S (added)
67 GOTO 20 (added)
70 T4 = T4 + S
75 GOTO 20
80 T1 = T1 + S
85 GOTO 20
90 T2 = T2 + S
95 GOTO 20
100 T3 = T3 + S
105 GOTO 20
110 PRINT
120 PRINT "TOTALS ARE :"
125 PRINT " BRANCH 1 : " ; T1
130 PRINT " BRANCH 2 : " ; T2
135 PRINT " BRANCH 3 : " ; T3
140 PRINT " BRANCH 4 : " ; T4
150 END
```

If you understood the original example you may be able to sort this one out on your own.

If the information read in is for a valid branch (1 to 4), the program will branch to one of the statements 80, 90, 100, or 70. The sales figure will be added to the appropriate total and the program will be sent back to line 20 in each case.

However, if the branch is any other number, the program will continue on to line 65 where it will print out an error message and the invalid data for us. It then returns to line 20 for more data.

Our report will now indicate any data with an incorrect branch number.

12-2. Variations of the IF Statement

Your computer may allow IF statements in the form:

IF condition THEN action

The condition is, as before, some comparison. The action may be a calculation, a PRINT statement, or any other valid BASIC statement. If the condition is true, the action will be performed.

If your computer permits its use, the above program could be simplified as follows:

```
10 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
12 DATA 3, 4000, 2, 600, 2, 7000
13 DATA 1, 6000, 4, 900, 3, 450
14 DATA 2, 1400, 0, 0
15 T1 = T2 = T3 = T4 = 0 (you may need separate lines)
17 REM T1 IS TOTAL FOR BRANCH 1, T2 FOR BRANCH 2, ETC.
20 READ B, S
30   IF B = 0 THEN 110
40   IF B = 1 THEN T1 = T1 + S
50   IF B = 2 THEN T2 = T2 + S
60   IF B = 3 THEN T3 = T3 + S
70   IF B = 4 THEN T4 = T4 + S
75   IF B <= 4 THEN 20
80   PRINT "INVALID BRANCH NUMBER " ; B ; S
90 GOTO 20
110 PRINT
120 PRINT "TOTALS ARE :"
125 PRINT " BRANCH 1 : "; T1
130 PRINT " BRANCH 2 : "; T2
135 PRINT " BRANCH 3 : "; T3
140 PRINT " BRANCH 4 : "; T4
150 END
```

(We haven't allowed for negative branch numbers here.)

Include an invalid branch number in a DATA statement to see if it is detected; e.g., change line 14 to:

```
14 DATA 2, 1400, 8, 1000, 3, 100, 0, 0
```

Explanation Only one of the statements 40 through 70 can be true, so only one of them will be performed for a given set of data. Again the sales will be added to the correct branch total.

As before, an invalid branch number will be detected and an error message will be printed out.

This form of testing for an invalid branch is used, since the program looks at all of the statements in the program using the above technique; i.e., it is not sent anywhere after the sales are added to the correct branch total.

Therefore, when the program checks line 75 it determines whether the branch number was less than or equal to four. If it was (and, hence, a valid branch number) it is sent back to get the next set of data. If not, it continues on and prints the error message before returning to read the next set of data.

If your computer permits the above technique to be used, it will probably permit you to replace lines 75 and 80 in the above program with a single line such as:

```
75 IF B > 4 THEN PRINT "INVALID BRANCH NUMBER " ; B ; S
```

Again, the error message will print only if an invalid branch number is detected.

Programs can become very involved using IF statements. Consider the following example:

Let's say that we want to produce a pay report for employees. However, the pay structure is a little more involved than we have used to date.

If employees work over 40 hours in the week, overtime pay is at time-and-a-half. For any overtime beyond 20 hours, pay is to be calculated at double-time. Employees working less than 20 hours in a week receive a lump-sum bonus of $25.

A program that would accomplish this would include several IF statements due to the complexity of the above conditions.

The following program will do what we want:

```
5 REM. . .N$ IS NAME. . .E IS EMPLOYEE #. . .H IS HOURS
6 REM. . .R IS WAGE RATE
10 DIM N$(10)
20 DATA "HILL", 3333, 40, 6
25 DATA "DALE", 4444, 30, 10
30 DATA "BELL", 5555, 15, 8.5
35 DATA "ROSS", 6666, 65, 9
40 DATA "GOFF", 7777, 50, 7
45 DATA "X", 0, 0, 0
50 READ N$, E, H, R
60    IF H = 0 THEN 100
70    IF H <= 60 THEN 80
75    P = (40 * R) + (20*R*1.5) + (H-60)*R*2
76    REM .. LINE 75 IF HOURS ARE OVER 60
77    GOTO 95
```

```
80   IF H <= 40 THEN 90
85   P = (40 * R) + ((H-40) * R*1.5)
86   GOTO 95
87   REM .. LINE 85 IF HOURS ARE 40 TO 60
90   P = H * R
91   REM .. LINE 90 IF HOURS ARE 40 OR LESS
92   IF H > 20 THEN 95
94   P = P + 25
95   PRINT N$, H, R, P
97 GOTO 50
100 END
```

This program would produce the following results:

```
HILL.   40    6     240
DALE    30    10    300
BELL    15    8.5   152.5
ROSS    65    9     720
GOFF    50    7     385
```

Of course, an actual company payroll would be much more complicated than even this. There would be several deductions which may or may not apply, varying pay periods, etc.

Some forms of BASIC allow the use of a special IF statement called the IF-THEN-ELSE statement. It is a very powerful form which can often simplify an otherwise complex series of IF statements.

The IF-THEN-ELSE Statements IF-THEN-ELSE statements are permitted by some versions of BASIC. They provide an efficient means of separating your program into two distinct paths so that you can treat various sets of data differently. They find wide application in structured programming.

The general form for these statements is:

IF condition THEN calculation #1
ELSE calculation #2

The computer looks at these two statements as a pair. It is an "either one or the other" situation. If the condition is true, the computer will perform calculation #1. If it is not true, it performs calculation #2.

As an example, let's assume that we are reading some data into a program, including an item name, the item price, and a tax code. If the code is a 1 the item is taxable, if it is a 2, it is not taxable. Assume that tax is at 8%, that the variable P contains the item price and that variable C contains the tax code. We will call the tax T. We could then write:

```
110 IF C = 1 THEN T = P * .08
120 ELSE T = 0
130 PRINT "TAX IS " ; T
```

A flowchart segment for these statements is shown in Figure 12-3.

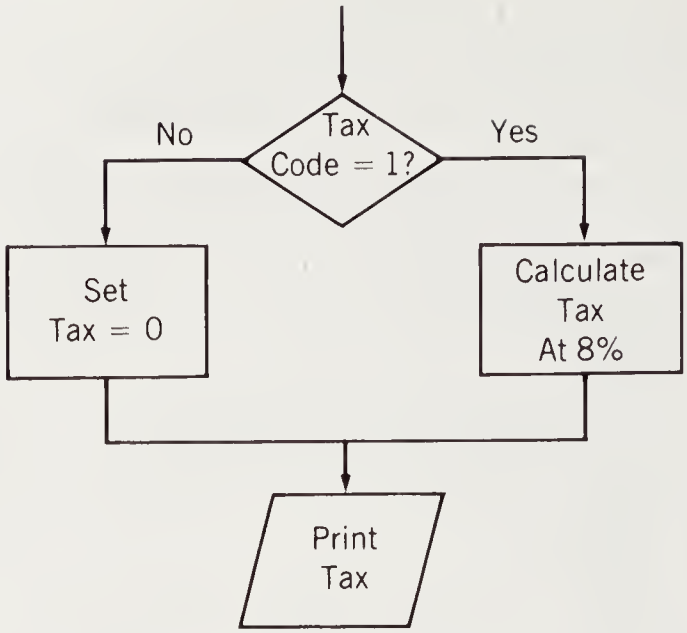


Fig. 12-3 Flowchart Segment of the IF-THEN-ELSE Statements

You might ask why bother with line 120 - if the tax is zero, don't do anything.

There is a problem with that, though. Line 130 prints the tax calculated. Since we assume that we are looking at several items, some will be taxable, some not.

If we leave out line 120, and the program comes to an item that is not taxable, it would skip the calculation on line 110, and proceed to line 130 which tells it to print out the tax. Since it did not calculate any tax this time, it will print whatever is in the variable T from before; that is, from the last taxable item. Hence, a non-taxable item ends up being printed out with some tax, when the tax should be zero.

Line 120 is vital.

The IF-THEN-ELSE-DO-DOEND Statements (These are offered in some versions of BASIC and not others.) Let's look once again at the earlier example that calculated raises. This time we will use something called the IF-THEN-ELSE-DO-DOEND statements.

These statements use the following STRUCTURE (or CONSTRUCT):

```
100 IF condition THEN DO
110 . . .
120 . . . (group 1)
130 DOEND
140 ELSE DO
150 . . .
160 . . . (group 2)
170 DOEND
```

On a flowchart these would appear as in Figure 12-4.

While the first form we looked at (IF-THEN-ELSE) allowed us to perform one or the other of two STATEMENTS, this form allows us to perform one or the other of two GROUPS OF STATEMENTS.

Each of these two groups is enclosed between the words DO and DOEND (the end of the DO).

If the condition is true, the computer will perform the BASIC statements that are in group 1 (lines 110 and 120 in the above example). If the condition is not true, it performs those in group 2 (lines 150 and 160).

The above framework always holds. The IF statement itself always ends with the words THEN DO, the DOEND is always on a line by itself, as is the ELSE DO, etc. The only flexibility that you really have is in specifying the condition, and in choosing what statement(s) will comprise each of the two groups.

The framework defines the condition to be tested for and provides separation for the two groups of statements.

The entire portion of the program using the IF-THEN-ELSE-DO-DOEND statements can be read as though it were one long sentence: "IF such-and-such a condition is true, THEN DO this group of statements (group 1), otherwise (ELSE) do this group of statements (group 2)."

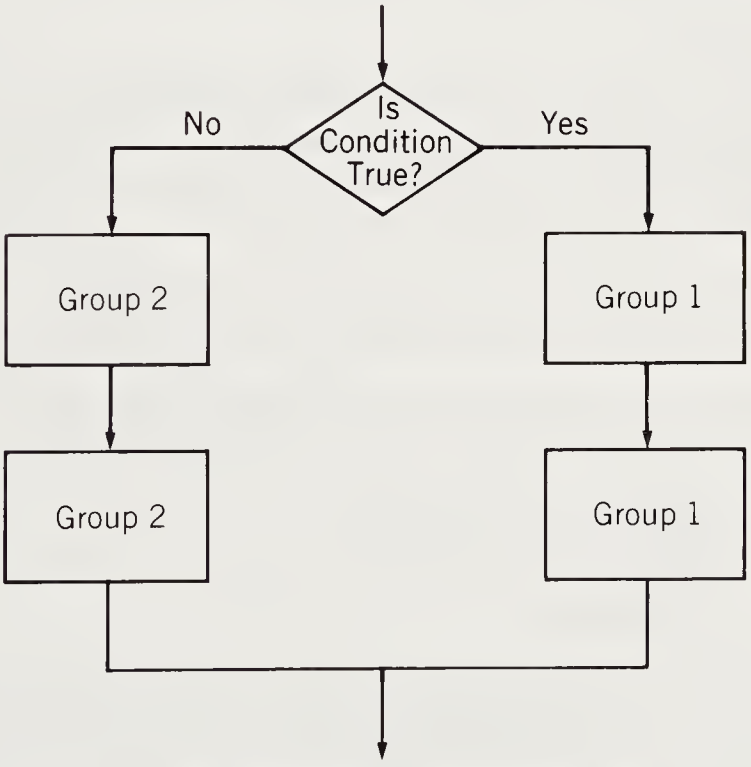


Fig. 12-4 Flowchart Segment of the IF-THEN-ELSE-DO-DOEND Statements

By the way, there may be more statements in one group than the other, if you wish. I have used two in each group strictly for purposes of illustration.

Let's use these new statements for our example of calculating pay raises (see program in Section 12-1). The requirement was to calculate a raise of 12% if an employee's present wage rate was $8.00 per hour or less, or a raise of 9% if the present wage rate was over $8.00 per hour. Recall that N$ is the employee name, H is the hours worked, R is the current wage rate, and A is the raise calculated. T1 and T2 are variables to keep track of the totals in each raise category.

The following program segment will accomplish this.

```
70 IF R <= 8 THEN DO
72    A = R * 0.12
74    T1 = T1 + A
76 DOEND
78 ELSE DO
80    A = R * 0.09
82    T2 = T2 + A
84 DOEND
95 PRINT N$; H; R; A
```

Note that some lines are indented. This is for clarity. Some computers indent all statements that are between DO and DOEND statements automatically for you. It is a valuable aid that is used extensively in STRUCTURED PROGRAMMING - a disciplined method of writing well-structured, logical, easily understood programs.

If the hourly rate of an employee is $8.00 or less, then lines 72 and 74 are performed (group 1); otherwise, lines 80 and 82 are (group 2).

No matter which of the two groups is performed, the program will then advance to line 95 (the first one below the IF-THEN-ELSE-DO-DOEND structure). Thus, it will perform lines 72 and 74 and then line 95, or lines 80 and 82 and then line 95.

These statements offer a means of writing a program which "flows" better than it otherwise would. They are used extensively in structured programming.

At this point, I should mention that the ELSE clause is always optional; that is, we can use only the upper portion of the structure if we wish:

```
100 IF condition THEN DO
110 . . .
120 . . .
130 DOEND
```

In this case the group of statements (110 and 120) will be performed if the condition is true, but if the condition is not true, the program will skip lines 110 and 120. Thus, the computer either performs the group of statements or not, depending upon the result of testing the condition.

We could have a series of this form as well. For example, remember the program we wrote to calculate total sales by branch (see Section 12-1). What if

we also wanted to count the number of sales figures we added to each branch total? In that case, we would be counting salespersons and the following routine would get the job done:

```
40 IF B = 1 THEN DO
42   T1 = T1 + S
44   C1 = C1 + 1
46 DOEND
48 IF B = 2 THEN DO
50   T2 = T2 + S
52   C2 = C2 + 1
54 DOEND
56 IF B = 3 THEN DO
58   T3 = T3 + S
60   C3 = C3 + 1
62 DOEND
64 IF B = 4 THEN DO
66   T4 = T4 + S
68   C4 = C4 + 1
70 DOEND
```

For each branch, we calculate the total sales and the count of how many sales were added to the total. Four branches are tested for. Depending on which branch is involved, the appropriate lines of the program will be performed.

12-3. Relational Symbols Used

We used the symbol (<=) for "less than or equal to" in this chapter. There are a number of others that most computers allow as well. A more complete list would include:

Symbol	*Meaning*
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Some computers are fussier than others. The fussy ones insist that <= be used for "less than or equal to". For the not so fussy ones, either <= or =< is acceptable. You will have to test yours to see what it requires.

12-4. Boolean Operators (Logical Operators)

Boolean operators offer a great deal of power to those computers that allow their use. They permit us to combine conditions in an IF statement.

Most computers that allow the IF-THEN-ELSE structure also allow these.

Boolean operators include the words AND, OR, and NOT. They are used as in the following example:

Assume that we wish to calculate raises for employees again, but this time the raises are according to the following schedule:

Current hourly rate	Raise (%)
less than $5.01	16
5.01 - 6.00	15
6.01 - 8.00	12
8.01 - 10.00	10
10.01 - 99.00	6

This problem is rather complicated using the ordinary IF-THEN-ELSE-DO-DOEND structure; however, if Boolean operators are permitted, the program becomes relatively straightforward:

```
40 IF R < 5.01 THEN DO
42    A = R * 0.16
44    C1 = C1 + 1
46 DOEND
48 IF R >= 5.01 AND R <= 6.00 THEN DO
50    A = R * 0.15
52    C2 = C2 + 1
54 DOEND
56 IF R >= 6.01 AND R <= 8.00 THEN DO
58    A = R * 0.12
60    C3 = C3 + 1
62 DOEND
etc.
```

12-5. Imitation of the IF-THEN-ELSE and IF-THEN-ELSE-DO-DOEND Structures

Many versions of BASIC do not permit the use of the IF-THEN-ELSE and IF-THEN-ELSE-DO-DOEND structures. If yours does not, you are still able to perform the equivalent with the use of the ordinary IF-THEN statement. The code that you write will not be as concise nor as structured, but it will do the job, nonetheless. The early examples in this chapter illustrate this.

Let's look at the previous examples to see how we could write them using only the IF-THEN structure. The initial problem posed was to consider a number of items which may or may not have been taxable. The data included item name, item price, and a tax code. If the code was a 1, the item was taxable, if the code was a 2, it was not taxable. We assumed a tax rate of 8%.

To solve this problem before, we used the IF-THEN-ELSE structure. Using only the IF-THEN structure we could write it as:

```
110 IF C = 1 THEN 140
120 T = 0
```

```
130 GOTO 150
140 T = P * .08
150 . . .
```

A second sample problem was to calculate pay raises based on current wage rates (see Section 12-1). An employee earning $8.00 per hour or less was to be given a 12% raise, while one earning over $8.00 per hour was to receive a raise of 9%. Calculated raises were to be added to separate totals. That is, all raises of 12% were to be kept in one total; all 9% raises in a second total.

Assume that the employee name is in the variable N$, the hours worked are in H, the current wage rate is in R, and the new rate is in the variable S.

We used the IF-THEN-ELSE-DO-DOEND structure to solve this problem before.

Using the IF-THEN structure only, the code might look like this: (Indentation is for clarity.)

```
70 IF R <= 8 THEN 80
72    A = R * 0.09
74    T2 = T2 + A
76    GOTO 95
80 A = R * 0.12
82 T1 = T1 + A
95 PRINT N$; H; R; A
97 GOTO 50
```

If more choices are involved - as in the example using branch totals (Section 12-1) – the IF-THEN statement can be used in yet another way.

Recall that we read in a branch number (B) and the associated sales (S). We wanted to add the sales to the appropriate branch total. The following program segment illustrates a slightly different use of the ordinary IF-THEN structure to solve the above problem.

```
100 IF B > 1 THEN 110
102    T1 = T1 + S
104    GOTO 150
110 IF B > 2 THEN 120
112    T2 = T2 + S
114    GOTO 150
120 IF B > 3 THEN 130
122    T3 = T3 + S
124    GOTO 150
130 T4 = T4 + S
150 . . .
```

You will find that in the case of the IF statement there is considerable flexibility in how you write your programs.

Summary

The IF statement offers us a means of having a program branch to one or the other of two possible paths, or perform one or the other of two sets of statements. A combination of IF statements increases this choice to as many paths as we may require in a program.

There are several forms of the IF statement. All computers accept the basic format:

```
70 IF B = 2 THEN 120
```

but many accept the form:

```
70 IF B = 2 THEN GOTO 120
```

Only some will accept the more advanced structures (also called CONSTRUCTS) such as the IF-THEN-ELSE and the IF-THEN-ELSE-DO-DOEND. It is important to remember, however, as stated above, that these structures can be imitated using the basic IF-THEN statement.

Exercises

12-1. The Invoice Exercise

Input data:

Item	Item #	Price($)	Quantity
STAPLER	6217	10.00	5
CHAIR	3424	69.95	100

Change the program so that it will only print out items whose extended price is greater than $1000. This will mean that only one item will be printed (the chair). Ensure that you still include both items in the calculation of average extended price and the count. See Exercise 11-1 if you have forgotten the requirements.

Hint: The key to this exercise is in the placement of the IF statement which you add. Try to decide beforehand where it should be placed by studying your flowchart. Alter the flowchart to allow for the change.

12-2. Given the following employee data:

Employee #	Name	Present Salary($)
3434	DRAKE	20000
2468	CARTER	18000
9876	DRACKETT	25000
6543	DUNDON	30000

Draw a flowchart and write a program to calculate raises for these employees. If they currently earn less than $ 22 000 their raise will be 15%, if they earn $ 22 000 or more their raise will be 10%.

Print the results as follows:

EMPL #	NAME	OLD SALARY($)	RAISE($)	NEW SALARY($)
3434	DRAKE	20000	3000	23000

12-3. Given the following data for Rob-Lee Clothiers Inc.

Item #	Item	Price($)	Tax Code
3344	BLOUSE	25.95	1
5566	SHOES	34.50	2
7788	SHIRT	18.00	1

(Tax code of 1 = taxable; 2 = not taxable)

Draw a flowchart and write a program to print out the following report. Use a tax rate of 6%.

ROB-LEE CLOTHIERS INC.

ITEM NAME	ITEM #	PRICE	TAX	TOTAL PRICE
BLOUSE	3344	25.95	1.557	27.507
				etc.

12-4. Given the following for Bowman Retailers:

Customer #	Name	Balance Due[$]	Age of Debt (days)
1234	DILLON	14000	90
5791	SHERMET	18000	60
7777	STACEY	14500	60
8844	SUMMERS	10000	120
8965	TWARDOWSKI	12500	60
2468	WESSELS	9250	120

Draw a flowchart and write a program to print a report which lists all accounts that are *90 days old* or more. Include the totals shown.

BOWMAN RETAILERS

CUSTOMER #	NAME	BALANCE DUE
1234	DILLON	$ 14000
		etc.

NUMBER OF ACCOUNTS (60 DAYS OLD)_____TOTAL DUE $_____
NUMBER OF ACCOUNTS (90 DAYS OLD)_____TOTAL DUE $_____
NUMBER OF ACCOUNTS (120 DAYS OLD)_____TOTAL DUE $_____

12-5. Given the following data concerning a number of bank account transactions:

Account #	Opening Balance($)	Amount($)	Code
67123	30000.00	5400.87	C
89898	450.84	89.95	D
24680	1700.00	251.77	C
12321	625.33	1624.80	D

(Code C indicates a credit, code D a debit.)

Draw a flowchart and write a program to produce a report in the following form:

ACCOUNT #	OPENING BALANCE	CLOSING BALANCE
67123	30000	35400.87

12-6. Alter 12-5 so that it will also keep and print a total of the opening and closing balances for each account. Alter the flowchart as necessary before you change the program.

13 The INPUT Statement

This chapter introduces us to the statement that allows us to "converse" with the computer.

New Vocabulary CONVERSATIONAL INTERACTIVE
INPUT statement

13-1. Accepting Data Through the Keyboard

To this point we have entered our data into the computer by including it within our programs in assignment statements or in DATA statements. We wrote our programs, debugged them, typed RUN, and sat back and waited for the magic to happen (after a bit of struggling perhaps).

Another very useful method to enter data into a program is by using what is called an INPUT statement. This statement allows input data to be submitted to a program through the keyboard, as the program is running.

This is called INTERACTIVE or CONVERSATIONAL mode because we are interacting or conversing with the computer. Essentially, we carry on a dialogue with it.

For instance, when you play games on the computer, it might ask you to enter your name, a bet, a guess, etc. In each case, we provide data to the program via the keyboard and the INPUT statement is used to accept our reply, whether it be our name, a number or whatever.

The computer waits for our reply. We type it in and press RETURN (ENTER).

The INPUT statement will replace the READ statement in the program and there will *not* be any need for DATA statements in the program. Remember that the READ and DATA statements are a team.

The key limitation in using DATA statements is that we must know the data before we run our program. We must then include it within the program in DATA statements, as we have been doing.

With the INPUT statement, on the other hand, it is possible to run the program and decide what data you wish to use *as you are running the program.*

Let's look at an example:

```
10 REM . . . AGE CALCULATION IN MONTHS
20 PRINT "HOW OLD ARE YOU "
30 INPUT A
40 M = (A + 1) * 12
50 PRINT "ON YOUR NEXT BIRTHDAY YOU WILL BE "; M ;
   " MONTHS OLD."
60 END
```

This program asks for the user's age. It will then calculate the age in months, and print out the result.

There are a couple of important points to mention here. First, when the computer reaches the INPUT statement it will automatically stop, print a question mark on the screen or paper, and wait for your reply to the question that was asked. *It will do nothing more until you respond to its prompt.* For our example the output would look like this:

```
HOW OLD ARE YOU
?
```

Where did that question mark come from?

The INPUT statement caused it to be printed. This should be kept in mind as we write programs that use the INPUT statement. This saves us adding a question mark to our actual question, since it automatically will appear on the screen or paper to remind the user that a reply is expected.

You will notice that the question mark printed below the actual question. If you would prefer that it be printed on the same line as the question itself, simply add a semi-colon to the end of the PRINT statement that poses the question:

```
20 PRINT "HOW OLD ARE YOU ";
```

Recall that the effect of a semi-colon at the end of a PRINT statement is that the next item printed will not be typed on the next line, but rather on the end of the current line.

The prompt would then appear as :

```
HOW OLD ARE YOU ?
```

and the cursor/print-head would be immediately after the question mark.

Once you type in your age, remember that you must press the RETURN (ENTER) key.

By the way, visual prompts are extremely important. If just question marks start popping out on the screen, people are often unsure what they are supposed to do. This is particularly true if they are asked to input several values.

When the above program is run, any age can be entered. It will be able to calculate the person's age (on their next birthday) in months.

Type the program into your computer and RUN it. When the question mark appears on the sceen, type in your age and press RETURN (ENTER).

The program will print out the answer in the form:

```
ON YOUR NEXT BIRTHDAY YOU WILL BE 240 MONTHS OLD.
```

Alphanumeric Input The INPUT statement can accept alphanumeric data as input as well. The same rules as always apply; namely, the alphanumeric variable name must be a letter of the alphabet followed by a $, and the chosen name must be dimensioned. These requirements vary with different versions of BASIC, remember.

It is also possible to have more than one element of data entered with a single INPUT statement. Consider the following program:

```
5 REM .. PROGRAM TO CALCULATE TAX (AT 7%)
10 DIM N$(15)
20 PRINT "ENTER THE ITEM NAME AND ITS PRICE."
25 PRINT "ENTER A PRICE OF ZERO TO QUIT."
27 REM. . . NOTE THAT LINE 20 IS THE PROMPT AND
28 REM. . . LINE 25 TELLS THE USER HOW TO STOP THE
29 REM. . . PROGRAM - OTHERWISE IT NEVER WOULD
30 INPUT N$, P
(your computer may require a semi-colon)
35 IF P = 0 THEN 80
40 T = P * .07
50 T1 = P + T
60 PRINT N$, P, T, T1
(or use the TAB function)
70 GOTO 20
80 END
```

This program asks for an item name and its price. It then calculates the tax (at 7%) on the item and prints out the item name, its price, the tax, and the total cost (price + tax).

For example, if we type in

```
"RADIO", 40
```

and press RETURN (ENTER) (Don't forget the commas between values that you enter and quotation marks around any alphanumeric data.), the output would look like this:

```
RADIO      40.00      2.80      42.80
```

The GOTO statement on line 70 allows the program to accept many items (one at a time of course).

When we wish to end the program, we enter any *name* for an item, but a *price* of 0. As always, we must enter a complete set of data. Thus, we need both an item name and a price, even though the end-of-data-test only looks at the price of the item; e.g., we could enter "X", 0.

The INPUT statement thus allows us to input data through the keyboard, rather than in DATA statements. This is extremely helpful in many instances.

13-2. Variations of the INPUT Statement

Some versions of BASIC allow more than one form of the INPUT statement to be used. For example, some permit us to combine our prompt with our variable name(s):

```
50 INPUT "ENTER ITEM NAME AND PRICE ", I$, P
```

If yours doesn't, you will have to use the form we employed above.

Some computers require a semi-colon after the prompt:

```
50 INPUT "ENTER ITEM NAME AND PRICE " ; I$, P
```

You might try each of these with your computer to what format(s) it allows.

Flowcharts An INPUT statement is shown on a flowchart using the same symbol as a READ statement.

A complicated program could include both INPUT statements and READ/DATA statements. Information which is the same for every running of your program would be in DATA statements within the program, while information which varies each time would be entered through INPUT statements.

Summary

The INPUT statement increases the flexibility of the language considerably, giving us a completely different method of inputting data to the computer. We, in effect, become involved in a two-way exchange with the machine. This mode of operation of a program is called INTERACTIVE or CONVERSATIONAL.

Exercises

13-1. The Invoice Exercise

Input data:

Item	Item #	Price($)	Quantity
STAPLER	6217	10	5
CHAIR	3424	69.95	100

Change the program so that it asks you to INPUT the data as the program is running. The request for input will be in the middle of the report.

This cannot be avoided. For a business report, we would not use this statement in this instance.

Print out the ITEM NAME, ITEM #, ITEM PRICE, and EXTENDED PRICE (price times quantity) for each item entered.

Alter the flowchart for the program first.

13-2. Draw a flowchart and write a program that will accept a Fahrenheit temperature as input and print out the equivalent Celsius temperature in the format:

THE CELSIUS FOR 50 IS 10. (This example assumes that the user entered 50 as the Fahrenheit temperature.)

Include brief instructions for the user; e.g., THIS PROGRAM ASKS YOU TO ENTER A FAHRENHEIT TEMPERATURE. IT WILL THEN CALCULATE THE EQUIVALENT IN CELSIUS . . . etc.

Allow the user to input several values without having to rerun the program each time.

The formula for conversion is C = (5/9)(F-32).

Hint: Remember that you still must use the multiply symbol in your program between the (5/9) and the (F-32).

13-3. Try changing exercise 13-2 so that it allows the user the option of bypassing the instructions if they already know how to use your program. Alter the flowchart as required.

Hint: Use an INPUT statement and an IF statement to do this.

13-4. Draw the flowchart and write a program that will allow a teacher to input several names and marks for a class of students and have the program print out a report in the following form:

```
   MARKS REPORT

NAME          MARK
ADAMS           70
THOREAU         82
CANT            77
NUMBER OF STUDENTS   3
CLASS AVERAGE   76.333
```

Assume that the teacher enters the names and marks for one student at a time. The request for data will print within the report. This cannot be helped at this point in our programming.

13-5. Draw a flowchart and write a program that will ask for the user's age, and print out a message giving the retirement date for the individual. Assume a retirement age of 65. Have the message print in the following form:

YOU WILL RETIRE IN THE YEAR 2020 IF YOU RETIRE AT AGE 65.

14 FOR-NEXT Loops

This chapter introduces us to a form of "controlled looping". It is especially helpful in constructing loan repayment schedules or other applications which vary in the number of lines to be printed.

New Vocabulary INDEX PARAMETERS
FOR-NEXT statement

14-1. The Format

We are now familiar with one type of loop in our programming. It has been the "working" section of our program. It starts with a READ or INPUT statement, contains the calculations which are to be performed on individual persons, items, etc., and prints the detail lines of our reports; i.e., the details about the individual person, item, etc. It ends with a GOTO statement which sends the program back to the READ statement.

The means of ending that loop was to use a dummy set of data, and an end-of-data test (an IF statement) to detect it.

The FOR-NEXT statements are another type of loop. They are used to control precisely how many times a series of statements is executed.

Consider the following program: (Indentation is for clarity: many systems automatically indent all statements between the FOR and NEXT statements.)

```
10 FOR N = 1 TO 3
20    PRINT TAB(10) ; N
30 NEXT N
40 END
```

Enter and RUN this program.

Explanation The FOR-NEXT statements are lines 10 and 30. Their only purpose is to "hold" a statement or series of statements, and to control how many times those statements are executed.

The statements they are "holding" are those between the two of them. In the example above, this consists of a single statement:

```
20 PRINT TAB(10); N
```

The simple format of the FOR-NEXT statements is:

```
(statement #)    FOR (index = starting value) TO (final value)
                 . . .
                 NEXT (index)
```

The starting value and the ending value are called PARAMETERS. A parameter can be thought of as meaning a value that defines a limit.

Thus, in the above program, the FOR statement is line 10 and the INDEX is N. An index can be thought of as a counter. Any valid variable name can be used for this counter. This index or counter is used to keep track of how many times the computer has repeated the loop.

We have told the computer to start the counter at 1 and eventually go up to a value of 3. It starts N at 1 as it is told to and proceeds to line 20 where it prints out the value of N (a 1 appears on the screen). It then goes on to line 30 where the NEXT statement tells the computer to loop back to the FOR statement.

Here, the first thing it does is to add another 1 to the counter and check to see if it has reached or exceeded the maximum value it is supposed to attain (3 in our example).

If it has not reached the maximum, it continues. In our example it will continue and print a 2 on the screen.

Again it meets the NEXT statement which sends it back to the FOR statement once more.

Here it adds yet another 1 to the counter (now 3) and continues to the PRINT statement where it will print a 3 on the screen.

Once again it reaches the NEXT statement, which sends it back to the FOR statement. This time, however, it finds that it has reached the maximum value that it is supposed to in the loop (3). It realizes that it has finished with the FOR-NEXT loop, so *it goes to the first statement below it.* In our case that is the END statement. The program is finished.

The output on the screen should be:

```
1
2
3
```

The FOR-NEXT statement has accomplished a number of things for us:

a. It has automatically kept a count for us - we did not have to include a counter such as C = C + 1 in the program.

b. It tested to see if the maximum count was reached (3 in our example). Otherwise we would have had to include an IF statement such as

```
IF C = 3 THEN 40
```

to stop it at the appropriate value.

c. Because of the two points above, our program is a little less complicated and it also flows better.

Retype line 10 with different starting and/or ending values and RUN the program again.

The STEP Parameter An important option with the FOR-NEXT statements is the STEP parameter. In our example, the program counted by ones. What if we wished to print the numbers 2 through 10, counting by twos?

To do this, we add the STEP parameter as shown:

```
10 FOR N = 2 TO 10 STEP 2
```

If we do not include this parameter, the computer assumes a step (increment) of one.

If we want a step of anything other than one, we simply add the keyword STEP with the size of the step immediately following it.

By using the STEP parameter as in the above example, the output would be:

```
2
4
6
8
10
```

The step can be a positive or negative number, and it may be fractional; e.g., 2, -3.5, -5, .02, etc.

Variables as Parameters Assume that you want to print out the numbers from one to whatever value a user chooses. The program to accomplish this could use an INPUT statement (to allow the user to enter a value), and the FOR-NEXT statements. For example,

```
10 PRINT "ENTER A NUMBER"
20 INPUT X
30 FOR N = 1 TO X
40    PRINT TAB(10) ; N
50 NEXT N
60 END
```

Notice that one of the parameters of the FOR-NEXT is a variable. The ending value depends on the number that the user enters through the keyboard. If they enter a 10 the program will print the numbers from 1 to 10 for example.

In fact any of the parameters can be variables. Hence each of the following forms is permissible:

```
40 FOR N = 1 TO 50
40 FOR M = 1 TO X
40 FOR S = Q TO 20.5
40 FOR P5 = M TO B
40 FOR Z = 40 TO 8 STEP -0.8
40 FOR B = Q3 TO C STEP 4
40 FOR J = 25 TO K
40 FOR A = T TO W STEP Q
etc.
```

The only requirement is that any of the variables that are used as parameters in the FOR-NEXT statements (starting value, final value, and increment) must exist, and they must have a value assigned to them by the time the FOR-NEXT loop is encountered.

You may also use decimal fractions as parameters, if you wish.

14-2. A Practical Example

Let's look at a more useful example now that we understand the principles involved.

Consider an investment that yields interest at the rate of 15% per year.

What would a program look like that would calculate the value of a given investment at that interest rate for any number of years specified? Assume that we wish to be able to enter any dollar value and to see the value of that investment at the end of each year up to the end of the specified time; e.g., $20 000 invested for 10 years.

Assume that the input data will consist of the amount invested and the number of years it is to be invested for.

Using the FOR-NEXT statements we have:

```
10 REM .. FOR-NEXT LOOP DEMO
20 PRINT "YEAR";TAB(8);"VALUE"
25 PRINT
30 READ I, Y
40 IF I = 0 THEN 100
50 FOR N = 1 TO Y
60    I = I * 1.15
70    PRINT N; TAB(10); I
80 NEXT N
90 GOTO 30
95 DATA 20000, 10, 0, 0
100 END
```

This is a little tricky. Line 20 prints some headings for us you'll notice. Then line 30 reads in the sum invested (I) and the number of years it is invested for (Y).

The FOR-NEXT loop will loop Y times which is just what we want; i.e., once for each year that the sum is invested.

Line 60 calculates the new amount of the investment - not just the interest - rather the new total of principal and interest. That is why 1.15 is used and not just 0.15 as would be the case otherwise.

Enter and RUN the program if you have not already. Manually check the output for the first one or two years to confirm that the results are correct.

To calculate more investments, we could add the following:

```
85 PRINT
92 DATA 10000, 12, 15000, 8
```

and change line 90 to

```
90 GOTO 20
```

FOR-NEXT statements are very useful for loans, investments, depreciations, etc., since these applications have varying periods to run. FOR-NEXT statements should normally be used in these instances and any others which lend themselves to this type of operation.

FOR-NEXT Statements on a Flowchart Representation of the FOR-NEXT statements varies from author to author you will find, though not significantly. Two representations are shown in the flowcharting segments of Figure 14-1.

Summary

The FOR-NEXT statements provide us with an easy means to control the execution of loops and, hence, output. They cannot be considered to be crucial

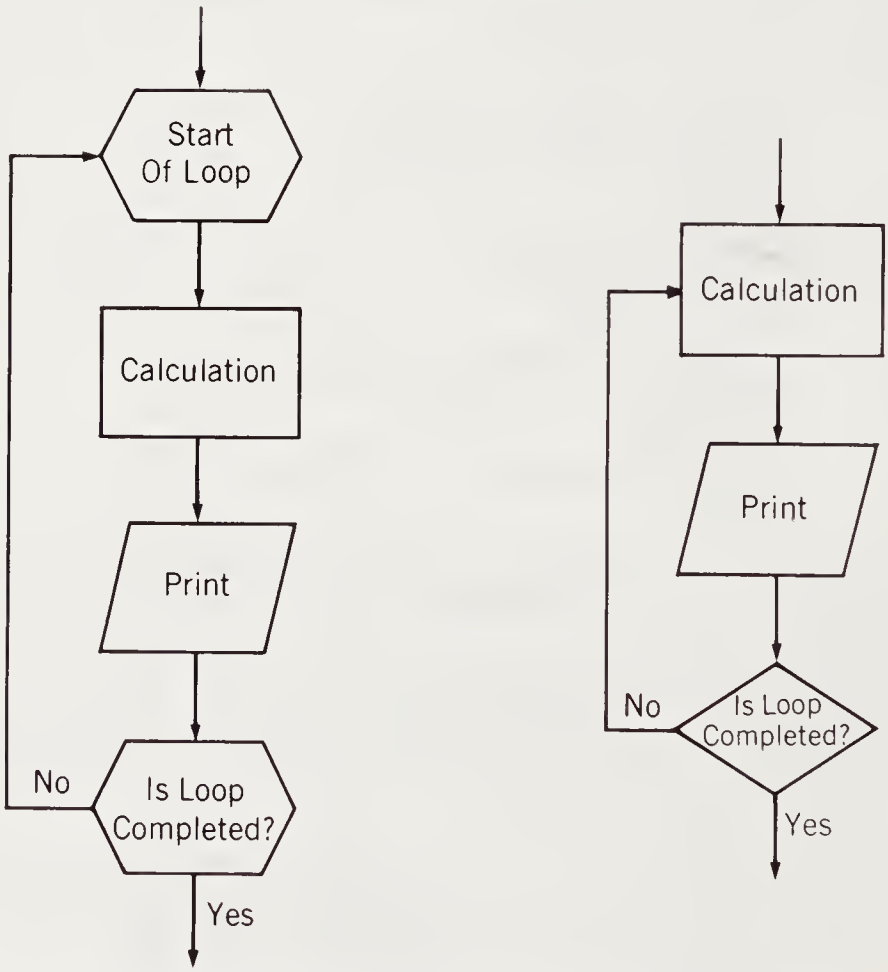


Fig. 14-1 Flowchart Segment of the FOR-NEXT Statements

statements, since we could use a combination of IF statements and a count to replace them. Nonetheless, they do make life a little easier for us. And who among us will object to that?

Exercises

14-1. (Depreciation Schedule) Given the following machine data:

Machine	Expected Life (years)	Initial Value($)
PRESS	10	200000
DRILL	15	80000
LATHE	12	325000

Draw a flowchart and write a program that will show the "straight-line depreciation" of these machines. Assume, at the end of the expected life period, that the machines are of no value.

For the press, the depreciation will be $200 000/10 or $20 000 per year.

Print the depreciation schedules in the form:

```
MACHINE : PRESS

END OF YEAR     VALUE($)
      1          180000
      2          160000
      3          140000
    etc.
```

14-2. Draw a flowchart and write a program that will produce a temperature conversion chart for Celsius to Fahrenheit temperatures. Allow the user to input the starting and ending temperatures for the chart and also the increment. For example one person might want a chart showing the temperatures from 0 to 100 with an increment of 5 degrees. Another may want those from 40 to 60 with an increment of 20 degrees.

Print the chart as in the following example:

```
TEMPERATURE CONVERSION CHART
CELSIUS     FAHRENHEIT
   40           104
   50           122
   60           194
```

This example uses a starting temperature of 40, an ending temperature of 60 and an increment of 10 degrees.

14-3. Try changing exercise 14-1 to use a depreciation factor of 10% per year (based on the current value). Change the flowchart as necessary.

Print the report in the same format as you did in exercise 14-1. Thus, for the press the schedule would begin as follows:

```
END OF YEAR     VALUE
END OF YEAR     VALUE
      1         180000
      2         162000
      3         145800
    etc.
```

14-4. Draw a flowchart and write a program that will print out whichever times table a user requests.

Print them in the following format:

```
10 TIMES TABLE
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
.
.
.
10 X 12 = 120
```

Allow the user to input his/her choice.

14-5. "What Will Print?" Exercise. If the following program were typed into the computer and run, what would the output be?

```
10 READ X
20 FOR N = 1 TO X
30    PRINT TAB(N); N
40 NEXT N
50 DATA 6
60 END
```

Note that line 30 uses a variable in the TAB function.

15 The GOSUB Statement

GOSUB statements often enable us to write a more structured, more easily understood program. They are helpful in structured programming.

New Vocabulary CONSTRUCTS SUBROUTINE
GOSUB Statement

15-1. The GOSUB Statement

A SUBROUTINE is a statement, or group of statements, that is placed in a program, often near the end, which may be "called" upon from another point (or points) in the program.

A subroutine always ends with a RETURN statement. Using subroutines in our "branch total sales" program would result in the following program. Recall that we read in sets of data, each of which contained a branch number, and a dollar sales figure associated with that branch. The sales were to be added to a total branch sales for the particular branch. (See Section 12-1.)

```
10 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
12 DATA 3, 4000, 2, 600, 2, 7000
13 DATA 1, 6000, 4, 900, 3, 450
14 DATA 2, 1400, 0, 0
15 T1 = T2 = T3 = T4 = 0 (you may need separate lines)
17 REM T1 IS TOTAL FOR BRANCH 1, T2 FOR BRANCH 2, ETC.
20 READ B, S
30    IF B = 0 THEN 130
40    IF B = 1 THEN GOSUB 80
50    IF B = 2 THEN GOSUB 90
60    IF B = 3 THEN GOSUB 100
70    IF B = 4 THEN GOSUB 110
72    IF B > 4 THEN GOSUB 120
75 GOTO 20
80 T1 = T1 + S
```

```
85 RETURN
90 T2 = T2 + S
95 RETURN
100 T3 = T3 + S
105 RETURN
110 T4 = T4 + S
115 RETURN
120 PRINT "INVALID BRANCH NUMBER"; B; S
125 RETURN
130 PRINT " BRANCH 1 : " ; T1
135 PRINT " BRANCH 2 : " ; T2
140 PRINT " BRANCH 3 : " ; T3
145 PRINT " BRANCH 4 : " ; T4
150 END
```

Explanation The IF statement with the GOSUB has the format:

IF condition THEN GOSUB line number

You will note that this is very similar to the form we learned earlier. It functions in a similar fashion as well.

Let's follow through the above program with the first set of data (3, 4000).

When line 20 (the READ statement) is encountered, the 3 is read into variable B and the 4000 is read into variable S. Line 30 is the end-of-data test. The computer finds that the branch is not zero, so it advances to line 40 where it will test for branch number 1. Since the value in B is not a 1, it advances to line 50, with the same results. It then goes on to line 60 where it tests for branch number 3. Finding that it is indeed branch 3, it will obey the GOSUB 100 part of the statement.

GOSUB 100 means "go to the subroutine" that begins on line 100 of the program. The computer will branch to the subroutine at line 100 at this point. It then performs whatever calculations, prints statements, etc. that are found at that location, "falling down" to successive lines as always, until it encounters a RETURN statement.

When it reaches a RETURN statement, it automatically returns to the line *following* the line of the program that sent control to the subroutine; i.e., the line after the one containing the GOSUB statement.

In our program, line 60 would send control to line 100 (since the data was for branch 3); the computer would perform line 100, then drop to line 105 which would send control back to line 70.

It examines lines 70 and 72, sees that they do not apply, so it falls to line 75 which sends the program back to line 20 (the READ statement).

In the above manner, each branch sales figure is added to the correct branch total. Eventually, the dummy set of data is read, and the program will then branch to line 130 where it will print out the branch totals and end.

15-2. IF Statement or GOSUB Statement?

We now have a choice of using IF statements or GOSUB statements to perform certain statements in a program. The choice of which to use is not always an easy one but you will find that the GOSUB statement often simplifies the "flow" of a program, making it easier to follow and more uniform in structure.

I suggest that you try both methods in programs at first. Experience will soon show you if either one or the other is superior.

The GOSUB statement often allows us to minimize the use of ordinary GOTO statements. In more complicated programs, there is a tendency on the part of novice programmers to use an excessive number of GOTO statements. This often leads to a very confusing program which tends to "bounce back and forth" - first down to line 145 as an example, then back to line 80, only to return to line 110 and eventually back to the READ statement on line 40 perhaps. The logic of the program becomes very difficult to follow, particularly for anyone other than the originator of it. Even the individual who wrote the program may have difficulty in sorting it out, particularly after a period of time.

GOSUB statements and structured programming techniques lessen the need for so many GOTO statements, leading to structured programming occasionally being referred to as "GOTO-less" programming. Although that may not quite be true – often a simple GOTO statement *is* the best one to use in a given circumstance – it does stress the fact that too frequent use of GOTO statements is one of the convincing reasons for the application of structured programming techniques in all programming.

Repetitive Calculation and Printing The GOSUB statement is often used to perform the same instruction(s) at various points within a program.

A simple example would be underlining in a report, or printing a line which would serve as a separator.

You could set up a subroutine such as:

```
250 PRINT
255 PRINT "_____"
260 PRINT
265 RETURN
```

which could then be "called" at appropriate points in the program. The result would be that each time it was called, a line would be printed on the screen/paper - with a blank line printed before and after it.

To offer you even more choice than you already have in your programming, there is yet another statement in BASIC that may be substituted for the IF statement and/or the GOSUB statement in certain instances. This is the ON-GOTO statement which is discussed in the next chapter.

The GOSUB Statement on a Flowchart The subroutine which a GOSUB statement refers to is usually represented on a flowchart with a predefined

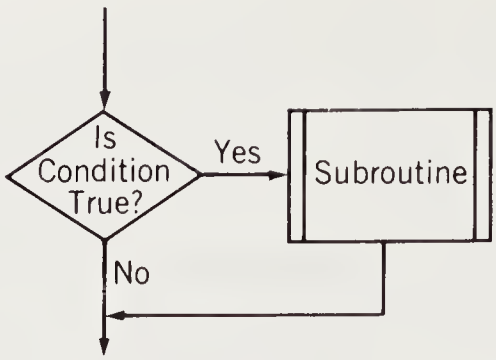


Fig. 15-1 Flowchart Segment of the GOSUB Statement

process symbol. This is simply the rectangle symbol with vertical lines added on either side as illustrated in Figure 15-1.

Summary

The GOSUB statement is the ideal statement to use in many instances. As programs become more complex you are more likely to find yourself using them to your advantage.

Exercises

15-1. A union has negotiated the following raise schedule:

Present Salary	Raise (%)
less than $10 000	15
$10 000 - $20 000	12.5
$20 001 - $30 000	11
over $30 000	9

Given the following employee data:

Number	Name	Present Salary
2345	FOX	20 000
7788	DAVIS	50 000
9876	MAUVE	9 500
4433	HUME	11 500
5566	JONSON	25 000

Draw a flowchart and write a program to produce a report of the following format:

NUMBER	NAME	PRESENT SALARY	RAISE	NEW SALARY
2345	FOX	20000	2500	22500

Use subroutines and GOSUB statements in your program.

PART II

This portion of the textbook includes instructions of BASIC that tend to vary from one computer to the next.

16 The ON-GOTO Statement

This statement offers an alternative to the IF statement in certain instances.

New Vocabulary GOTO N OF statement ON-GOTO statement

16-1. Its Purpose

When there are a number of choices to be made in a program, we can use a series of IF statements as discussed in Chapter 12. If there are several choices, however, the program can become excessively long. In certain instances the ON-GOTO statement provides us with a more efficient means of doing the same thing. Because of this, it is a useful statement in structured programming.

Recall the program from Section 12-1 which calculated total sales by branch. Remember there were four branches in the company and data records consisted of a branch number and a sales figure for a salesperson in that branch.

The program is repeated here for easy reference.

```
10 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
12 DATA 3, 4000, 2, 600, 2, 7000
13 DATA 1, 6000, 4, 900, 3, 450
14 DATA 2, 1400, 0, 0
15 T1 = T2 = T3 = T4 = 0 (you may need separate lines)
17 REM T1 IS TOTAL FOR BRANCH 1, T2 FOR BRANCH 2, ETC.
20 READ B, S
30    IF B = 0 THEN 110
40    IF B = 1 THEN 80
50    IF B = 2 THEN 90
60    IF B = 3 THEN 100
70    T4 = T4 + S
75 GOTO 20
80 T1 = T1 + S
```

```
85 GOTO 20
90 T2 = T2 + S
95 GOTO 20
100 T3 = T3 + S
105 GOTO 20
110 PRINT
120 PRINT "TOTALS ARE :"
125 PRINT " BRANCH 1 : " ; T1
130 PRINT " BRANCH 2 : " ; T2
135 PRINT " BRANCH 3 : " ; T3
140 PRINT " BRANCH 4 : " ; T4
150 END
```

Using the ON-GOTO statement we can replace the three IF statements (lines 40, 50, and 60) with a single line:

```
40 ON B GOTO 80, 90, 100, 70
```

(Note: Your computer may use the alternate form discussed in the next section.)

This new statement has the general form :

```
ON V GOTO L1, L2, L3, L4, . . .
```

where V is some variable in your program and L1, L2, L3, etc. are line numbers of statements within the program. I have suggested above that we use the variable B (for branch) and line numbers 80, 90, 100, and 70.

Explanation When the program reaches an ON-GOTO statement it looks at the variable name within the statement (B in our program). If the value of that variable is 1, the program will then branch to the *first* statement number listed after the GOTO. If the value of the variable is 2, it will branch to the second statement number listed after the GOTO, etc.

The critical point here is that the value of the variable in question must be a whole number greater than zero (1, 2, 3, 4, etc.), and it must be in the range of how many statement numbers are listed after the GOTO; e.g., if there are five statement numbers listed after the GOTO, the variable in question must have a value from 1 to 5 inclusive. If the variable has a value of 0 or less, or 6 or more, you may receive an error message when the program reaches the ON-GOTO statement. Some versions of BASIC simply ignore the ON-GOTO statement in this case.

It is never necessary to use the ON-GOTO statement, but it is handy at times. The important thing to keep in mind is that the variable you use must be able to assume relatively low integer values, or ones that can be easily changed to low integers. If that is not the case, it is probably easier to use IF statements.

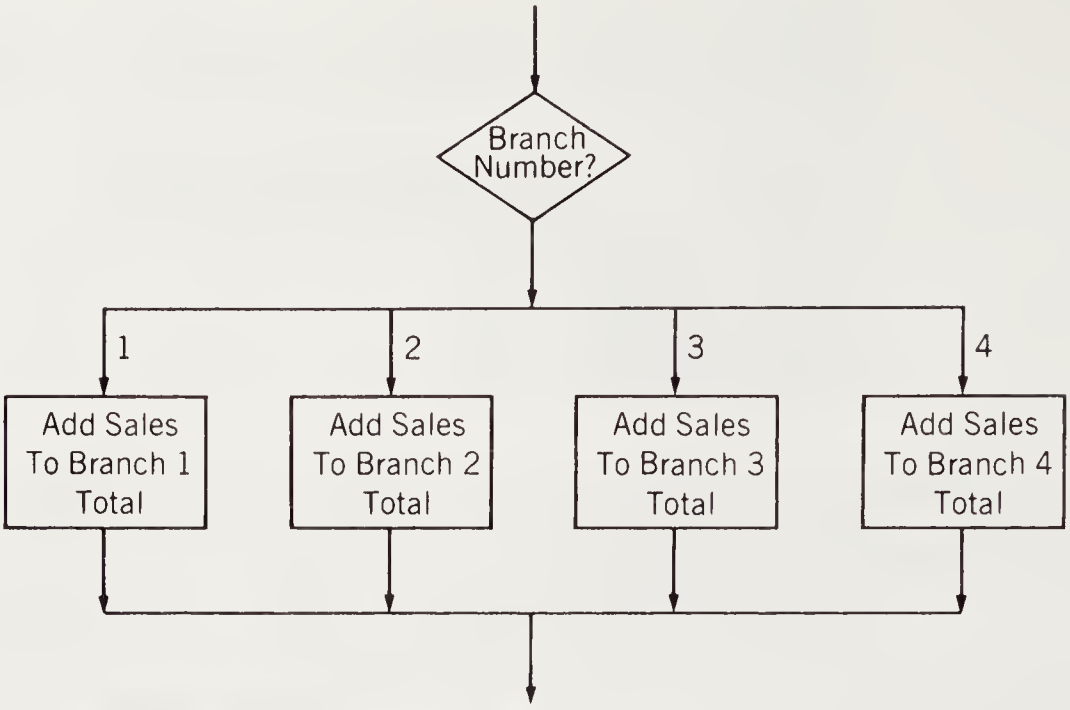


Fig. 16-1 Flowchart Segment of the ON-GOTO Statement

16-2. Variations of the ON-GOTO Statement

Another form of this statement used by some computers is the GOTO N OF statement. It performs the same function as the ON-GOTO statement. Only the format is different.

Thus, instead of writing

```
40 ON B GOTO 80, 90, 100, 70
```

as we did in the above program, we would use

```
40 GOTO B OF 80, 90, 100, 70
```

The results will be identical; i.e., if the value of B is 1, the program will branch to line 80, if it is 2, it would branch to line 90, etc.

The ON-GOTO Statement on a Flowchart The ON-GOTO statement is usually represented on a flowchart as illustrated in Figure 16-1.

17 The PRINT USING Statement

This statement enables us to print more attractive and professional looking reports. It is not available on all computers. Of those that permit its use, the form varies from one computer to another.

New Vocabulary IMAGE statement PATTERNS

17-1. Advantages of the PRINT USING Statement

We touched on this statement earlier in the text. This statement does the following for us:

a. It automatically aligns numeric values in columns on the decimal point; e.g.,

```
 32.07
   .69
155.00
```

b. It automatically rounds numeric values for us to the position we indicate; e.g., to two decimals.

c. It allows us to use the full print line without regard to print zones.

d. It allows us to insert dollar signs in a report easily.

e. It prints trailing zeros; that is, a value of $8.50 will print as 8.50 (with or without a $ sign as you choose) rather than as 8.5, as it has been printing in our output.

Different versions of BASIC use different forms of this statement. In principle, however, each is the same.

If your computer allows the use of PRINT USING statements, simply include them in your program in the exact position as the ordinary PRINT statements were. "Out with the old, in with the new."

17-2. The Format of the PRINT USING Statement

The PRINT USING statement actually requires a second statement, usually called an IMAGE statement. The purpose of the IMAGE statement is to tell the computer where and how to print the data indicated in the PRINT USING statement itself. It is saying, in effect, "PRINT this information USING the IMAGE statement as a guide".

The general form of this pair of statements can thus be thought of as being:

50 PRINT USING "what is to be printed"
60 IMAGE "where and how to print it"

The IMAGE statement distinguishes between numeric and alphanumeric data, by the way, so we will need some sort of indicators to note this difference.

Let's look now at an example.

Suppose that we want to print the title "SALES REPORT" beginning in print position 20. A common form of the PRINT USING/IMAGE statements of many computers is:

10 PRINT USING 20; "SALES REPORT" (what to print)
20: (19 spaces) **'LLLLLLLLLLL** (where and how to print it)

Line 10 says in effect, print the words SALES REPORT using line 20 as a guide.

Line 20 (any line number can be used as long as it matches the number in the PRINT USING statement) is the "image" that the computer is to use. This IMAGE statement must begin with a colon (:) and use the proper PATTERNS for data which are to be printed. The 19 spaces indicated in the example are counted out and included in the statement, not written as shown above. Hence, when you key in line 20, it will actually look like this:

```
20:          'LLLLLLLLLLL
```

The colon can be considered to be the left-hand side of the page. The remainder of the IMAGE statement is then an actual "image" or reflection of a print line.

Our example says to space over to print position 20 and print the title SALES REPORT. The single quote is counted as one of the 12 positions for the title (don't forget the space between SALES and REPORT).

There must be a pattern for each field or string that is to be printed. These patterns use the following symbols:

Pattern	*Meaning*
a. 'LLLLL	(A single quote must be in the first position.) Left-justify the output; that is, start printing it in the left-most position of the pattern location.

b. ###.##	A numeric value will print here. At most it will have three digits before the decimal point and two after it.
c. $	May be inserted in front of a numeric pattern. It will be printed with the actual value; e.g., $ ####.##

Other less frequently-used symbols are available on most systems that employ the PRINT USING statement.

The example above showed how to print a title. What about headings? The following would print the headings NAME, SALES, and COMMISSION at the specified positions.

```
40 PRINT USING 45; "NAME", "SALES", "COMMISSION"
45 :(4 spaces) 'LLL (5 spaces) 'LLLL (6 spaces) 'LLLLLLLLL
```

The first pattern ('LLL) has four positions, corresponding to the four letters in the word NAME. The other patterns are of a length corresponding to the headings to be printed in each case. The spaces between patterns can be adjusted as you wish.

Printing Within the Loop How can we print values within the loop portion of the program? An example of this would be:

```
60 PRINT USING 65; N$, S, C
65:'LLLLLLLLL (3 spaces) $####.## (4 spaces) $####.##
```

These statements will print the values in N$, S, and C as per line 65; that is, the name will begin in print position 1 and use the first 10 positions ('LLLLLLLLL requires 10). Any of the 10 not used are left blank.

The computer then spaces over three positions and prints the value of S as per the pattern $####.## which means it will print a $ followed by the numeric value of S, up to a maximum of 9999.99. This number will be aligned with the decimal point; i.e., values will print as follows:

```
$8004.75
$   68.70
$  100.00
etc.
```

The $ may or may not "float" depending upon the system you are using. If it does, it will float "up against" the first digit. The above output would then appear as:

```
$8004.75
  $68.70
 $100.00
etc.
```

PRINT USING statements appear in a program in the exact location that an ordinary PRINT statement would. The only difference is that the method of printing has changed. No change is necessary on flowcharts, either.

By the way, to help you plan your reports, you could use a PRINTER SPACING CHART. These are printed forms which have all print positions numbered.

Other Computers Some computers implement the PRINT USING statement in a different manner. Instead of locating the patterns in their actual positions in the IMAGE statement, they are described and written in sequence. Also, the actual word IMAGE is used in the IMAGE statement.

To print the above title using this form, we would type in:

```
10 PRINT USING 20; "SALES REPORT"
20 IMAGE 19X, 12A
```

Note the use of a semi-colon after the IMAGE statement number in line 10.

The IMAGE statement employs three letters to describe its patterns, instead of symbols as the other method did. These letters are:

	Letter	*Meaning*
a.	X	Spaces
b.	D	Numeric data (think of D for digits)
c.	A	Alphanumeric data

Thus, the IMAGE statement above tells the computer to space over 19 spaces and print an alphanumeric data value of 12 positions (in this case, the words SALES REPORT), starting in print position 20.

Note that the space between SALES and REPORT counts as a position, since the computer treats spaces as characters when they are between quotation marks.

To print the same headings as we did above using this new method, we would input:

```
40 PRINT USING 50; "NAME", "SALES", "COMMISSION"
50 IMAGE 4X, 4A, 5X, 5A, 6X, 10A
```

(Again, note the use of the semi-colon and commas.)

The 4X, 5X, and 6X indicate the spacing that we are inserting. The 4A corresponds to the word NAME, the 5A to SALES, and the 10A to COMMISSION.

To print the values in N$, S, and C as we did before (in the loop), we would use:

```
60 PRINT USING 65; N$, S, C
65 IMAGE 10A, 3X, $DDDD.DD, 4X, $DDDD.DD
```

The D's indicate that numeric data will be printed in these locations, remember.

You may also write line 65 as:

```
65 IMAGE 10A,3X, $4D.2D, 4X, $4D.2D
```

If your computer allows it, the PRINT USING statement is a valuable tool to have at your disposal. It permits you to design and produce reports which are more readable and, hence, more acceptable to users.

18 Files

This chapter introduces one of the most powerful and important concepts in data processing - a means of storing large amounts of data separate from the program(s) that use them.

New Vocabulary CREATE command RECORD
FILE

18-1. What Are They?

We have studied three methods of entering data so far: the assignment statement (e.g., C = 0), the READ/DATA statement combination, and the INPUT statement. The first two methods include the data within the program itself, while the INPUT statement enables us to enter data through the keyboard as the program is running.

There is also a fourth, very powerful means of entering data into programs. This one involves the use of FILES.

A file is made up of a collection of RECORDS. A record can be thought of as a DATA statement in our earlier programs. In other words, a record contains the details about an employee, customer, item, etc.

Files are separate from your program rather than being a part of it. They are stored on tape or disk and attached to the computer that you are using. Your program is able to access them through a few special statements used in your program.

A file can easily contain many records. Our programs to date have contained fewer than ten records (DATA statements) but a file may contain hundreds or even thousands.

Your first encounter with a file will probably be with one that an instructor or someone else has set up for you.

Files offer a number of advantages over the use of DATA statements:

a. The data, although entered into the file only once, can be used by several programs.

b. Many different people can use the file if the creator of it gives them permission to do so.

c. The one-time recording of data means that errors are minimized. If a long file had to be retyped every time someone wanted to use it, errors would probably be introduced each time.

d. It saves considerable time for users of the information. As they only need to write the logic portion of their programs and none of the data, they save a great deal of time.

Files are what most businesses use of course. The larger customer, employee, and inventory files may include up to 50 000 records or more.

All companies use the technique of files to enter their data into programs; however, to learn the techniques of programming, it is simpler to use READ/DATA statements and INPUT statements as we have to date. It is also true that many programs simply do not require files.

A point to keep in mind, as you study this section, is that reading data from files does not change the overall structure of our programs. We are simply obtaining data from a location separate from the program; that is, from a file, rather than from assignment statements, DATA statements, or through the keyboard. We must, therefore, have some means of letting the computer know this fact. The special BASIC statements below will serve this pupose.

The following program is for reference as you read this portion of the text. It will not run on all computers as it appears here, but by adapting the special statements related to files to your particular computer, you would be able to perform the equivalent functions illustrated in the program.

The Example The program is one which employs the techniques under discussion. As mentioned, although your computer may not allow certain file statements to be used, it will probably permit an equivalent form.

You might like to refer to this sample program as we discuss files.

```
10 REM PROGRAM USING TWO FILES
12 REM EMPIN IS AN INPUT FILE - EMPOUT AN OUTPUT FILE
14 REM N IS EMPLOYEE # - N$ IS EMPLOYEE NAME
16 REM B IS YEAR OF BIRTH - H IS YEAR HIRED
18 REM W IS HOURLY WAGE RATE - S IS YEARS OF SERVICE
20 DIM N$(10)
30 FILES EMPIN, EMPOUT (FILES statement)
40 READ # 1; N, N$, B, H, W (Special READ statement)
(optional in some versions)
50    IF END #1 THEN 100 (Special end-of-data test)
60    S = 1999 - H
65    PRINT N; N$; B; H; W
70    IF S > 10 THEN 40
80    PRINT # 2; N, N$, B, H, W
(Special PRINT statement)
90 GOTO 40
100 END
```

We must know the layout of the file(s) used in the program before we are able to code it. These would be given to us - or we would design them ourselves.

What it Does This program reads data from the file EMPIN (employee number, name, etc.). It then calculates the number of years that the employee has worked for the company (assuming that the current year is 1999!).

Employee details for all employees are printed on the screen/paper. If the employee has worked for the company for 10 years or more, the employee details are also written into the file EMPOUT. The program then branches back to the READ statement to read the next employee from the file EMPIN.

To use files, there are a few special statements required in our program. These take the form of:

a. A FILES statement (or its equivalent).

b. A special form of the READ statement.

c. A special form of the IF statement to test for end-of-data.

d. A special PRINT statement, if we wish to print something into a file.

Unfortunately the method of using files varies considerably from computer to computer. The principles still hold though.

The situation is as follows :

If you have a microcomputer, the CRT and the computer's main memory may be within the same cabinet. If you are using a minicomputer or a mainframe computer, they will probably be separate as in Figure 18-1.

The tape/disk device shown in the figure may be built-in or separate, depending upon the size and make of computer that you are using. The device for storing files (or programs) could be a hard or floppy disk, or a cassette or regular tape.

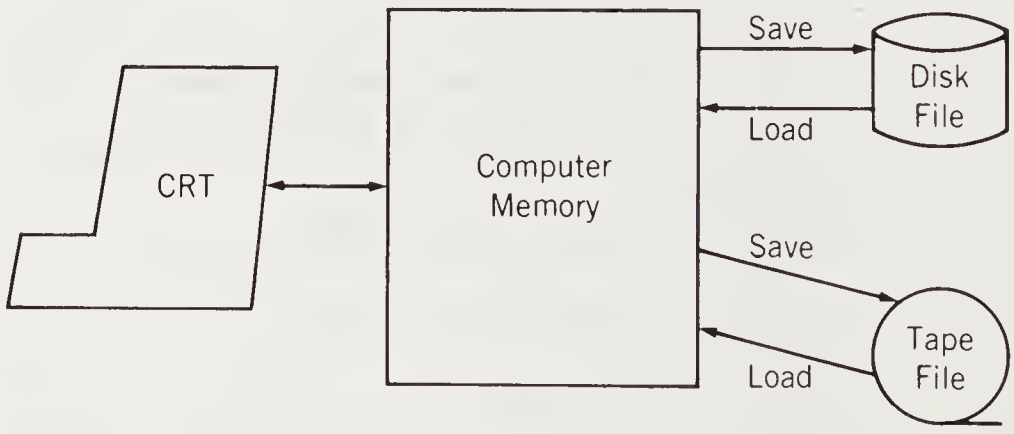


Fig. 18-1 File Storage

In a large installation there could be hundreds of files stored by different users. Even a microcomputer could have that many, but they usually do not. To keep them all sorted out, each file must have a name assigned to it.

This name is used in your program so that the computer knows which of its many files you wish to use.

Some computers use a FILES statement for this purpose:

```
22 FILES EMPIN
```

The word FILES must be plural even if you only use one file. If you were using more than one file, the FILES statement might look something like this:

```
30 FILES EMPIN, EMPOUT (see sample program above)
```

The computer looks at the names on the FILES statement and notes their sequence on it. From then on it thinks of the first file on the FILES statement as file # 1 (EMPIN in the above example) and the second file as file # 2 (file EMPOUT in our example). If there were more file names in the FILES statement, the computer would simply assign consecutive numbers to them in turn.

Because we decide the sequence that the file names appear on the FILES statement, it does not matter in what order we add them. The only requirement is that each of our working files must be listed on a FILES statement.

The FILES statement also tells the computer to set the file(s) up and get them ready to use. This process is referred to as OPENING the files. Some computers use a special statement called an OPEN statement for this purpose.

Two examples of OPEN statements are:

```
OPEN # 1: "EMPIN"
OPEN "EMPIN" FOR INPUT AS FILE #1, ACCESS READ
```

If your computer uses an OPEN statement, it does not require a FILES statement.

Your computer may use one of the above methods or a completely different one. There is a great deal of variance amongst computers when it comes to files and how they are implemented.

18-2. Special Requirements of Files

Special READ Statement In order to use the ordinary READ statement to read data from a file, a computer would have to be able to read our minds as well. (It has been told to go looking for DATA statements if it meets a READ statement, remember.)

Because of this, a special form of READ statement is used. One example might be:

```
40 READ # 1; N, N$, B, H, W (see sample program)
```

The #1 signifies the first file named in the FILES statement (or OPEN statement), that we looked at above. In this example we are assuming that the file contains records with five data items in them.

If there happened to be three files named on the FILES statement, and we wished to read the third one in the list, the READ statement might be:

```
40 READ #3; N$, E, R
```

Since we cannot "see" the information on a disk or tape, we must know beforehand what data would actually be on the file and in what sequence. Knowing these details, we are able to write a READ statement to access the data.

Special End-of-Data Test This is usually required when a file is being read. It might take the following form:

```
50 IF END #1 THEN 100
```

This means when the program reaches the end of file #1 (EMPIN in the above example – the first file named on the FILES statement), it will branch to line 100.

This is the equivalent of our end-of-data test which we used in programs throughout the text. Note that we need not be concerned with the actual "dummy data" as we did when we used DATA statements. The computer automatically takes care of that for us.

Many versions of BASIC combine the read *and* the test for end-of-data in a single statement, namely, the one just shown. If yours does this, you need only include this end-of-data test statement at the top of your loop (omit the READ statement), and it will both read a record and determine if it is the last one in the file.

Special Print Statement To print data into a file we use a PRINT statement such as the following:

```
90 PRINT #2; N, N$, B, H, W
```

Again the #2 signifies the second file named in the FILES statement (EMPOUT in the sample program).

Programs often read in data from one file and print results (input data and/or calculations) into a second one, as in the above example.

18-3. Creating a File

What if we wanted to set up a file to store some information? An OPEN statement will do this on many computers; others use a CREATE command. A CREATE command might look like this:

```
CREATE NEWFILE, 50
```

where CREATE is the keyword used for the command (as LIST, RUN, etc., are commands).

NEWFILE is the name that we have decided to use for our new file. The name is often limited to 8 characters or less in length and it must usually begin with a letter of the alphabet.

The 50 tells the computer to reserve storage space for up to 50 records in this file.

This CREATE command need only be typed once, before the program is run. From then on, the program can be run normally.

If your computer uses this command, at some point the two files in our sample program would have been created using it.

Rereading a File Within the Same Program Sometimes we wish to be able to start reading a file at the beginning again within the same program. This is possible through the use of the RESTORE statement, which has the effect of moving the data pointer back to the beginning of the file. It would look like this:

```
200 RESTORE #1
```

where the #1 is, as it always has been, the first file listed in the FILES statement or its equivalent.

The explanations of files in this chapter have been rather brief because of the wide variance in techniques cited above. It will be necessary for most of you to delve into your manuals to sort out the specific commands, statements, etc. required for your computer.

19 Conclusion

19-1. What We've Studied

We have covered quite a bit of ground in this text and we have discussed most of the capabilities of BASIC.

With the knowledge you have acquired to this point and with the practice you have had in applying it, you will find that you are able to write some useful and quite complex programs.

19-2. What We Haven't Studied

Some of the topics that we have not looked at include:

a. The setting up of USER-DEFINED functions. These are usually short mathematical operations that you need to use frequently in a program; hence, you define them using a special statement to be "called" when you need it.

b. Calling or invoking another program from within a program.

c. Output of results to a printer or other device attached to your computer.

d. Special statements available with some versions of BASIC which allow the use of sound (a range of electronic tones), graphics (drawing lines, diagrams, pictures), manipulation of character strings, manipulation of data in memory, etc.

e. Sorting a set of data into a specific sequence.

f. Certain built-in functions for mathematical operations, etc., (see Appendix B).

g. Arrays – a very powerful method of dealing with a group of related numeric or alphanumeric data (see Appendix C).

19-3. Beyond the First Steps

Often in a new field, the first steps are the most difficult. They require a mastering of basic concepts which are usually unlike anything we have encountered before. The result is that the learning process may take considerably longer than we expect it to.

If you feel, however, that you have a firm grasp of the concepts we have discussed, then you are ready to go beyond the first steps in BASIC.

Those of you who have a strong mathematical background, or who perhaps have experience in a specific field, will be able to extend yourselves far beyond this text.

Those of you with less mathematics in your background, or less of an inclination towards math, will find that you too can progress much further into the realm of programming.

As pointed out earlier, don't become too discouraged or impatient with yourself as you study this field. If you made it this far, I'm sure it wasn't without moments of frustration and discouragement. There may have even been times when you were sure that BASIC was going to beat you before you beat it!

But you made it! Keep that in mind on the road ahead.

Good luck to each of you in your chosen pursuits.

GLOSSARY and APPENDICES

GLOSSARY

ALGORITHM A series of steps used to solve a problem.

ALPHANUMERIC A data value containing letters, symbols and/or numerics, etc. It cannot be used in calculations.

ARRAY A group of related numbers. It could contain alphanumeric data as well though in some instances.

BASIC Beginner's All-purpose Symbolic Instruction Code.

BUILT-IN FUNCTION A feature which allows a programmer to call an operation up at any time; e.g., SQR for calculation of a square root.

CHARACTER A single letter, digit, symbol, etc.

COMMAND A keyword used to tell the computer what you want it to do; e.g., RUN, LIST, SAVE.

CONSTANT A value that does not change; e.g., 15.

CONVERSATIONAL A dialogue set up between the user and the computer.

CPU Central Processing Unit. The main component of a computer system. It includes memory and performs calculations, data moves, etc.

CRT Cathode Ray Tube. The "TV screen" terminal.

CURSOR The dash or block of light on a CRT screen that tells you where the next character will print.

DISK A device used to store programs or data on for later use. They may be "hard" or "floppy".

DOCUMENTATION The supporting paperwork for a program. It includes a copy of the program, flowchart, sample output, etc.

EXPONENTIATION Raising a value to a power.

FIELD A single data value; e.g., name, price, etc.

FILE A group of related records.

FLOWCHART A diagram of the logic of a program.

HARD COPY Output on paper.

HARDWARE The physical equipment that comprises the computer system; i.e., the CRT, CPU, disk devices, etc.

INCREMENT A "step value" which is added repeatedly. For example, the increment for line numbers in a program is most often 10.

INDEX The "counter" in a FOR-NEXT loop.

INITIALIZATION The starting of totals and counts at zero.

INPUT Data supplied to a program.

INSTRUCTION A line of a program.

INTERACTIVE see CONVERSATIONAL.

JUSTIFICATION Lining up of columns of printed output on the right-hand or left-hand side of each data value.

LIBRARY FUNCTION see BUILT-IN function.

LOOP A portion of a program which is repeated.

MATRIX see ARRAY.

NUMERIC A data value containing only digits, and possibly a sign (+ or -) and/or a decimal point.

ORDER of OPERATIONS The sequence that mathematical operations are performed in if there is any ambiguity.

OUTPUT The results from a program.

PARAMETER A value within a statement which may or may not change; e.g., the starting value of a FOR-NEXT loop.

PRINT ZONE An area on the screen/paper where output is printed.

PRINTER SPACING CHART A printed form to help us plan our output.

PROGRAM Instructions we use to tell the computer what we want it to do.

PROGRAMMING CYCLE A procedure to follow when writing programs.

RECORD The data values associated with an individual person, part, etc. It is equivalent to a DATA statement for most of our programs.

SIGNING-OFF The procedure to be followed when we are finished using the computer.

SIGNING-ON The procedure to be followed when you wish to begin to use the computer.

SOFT COPY Output printed on a CRT screen.

SOFTWARE The programs associated with a computer system.

STATEMENT A line of a program.

STRING see ALPHANUMERIC.

STRUCTURED PROGRAMMING A well-disciplined, orderly method of writing programs so that they are easy to read and understand.

SYNTAX The "grammatical" rules associated with BASIC statements.

TAPE A medium used to store programs and/or data for future use.

TERMINAL A device used to communicate with the computer; e.g., a CRT or teletype printer.

VARIABLE A value in a program that changes in value as different sets of data are read; e.g., N$, K, P6.

WORKSPACE The area in computer memory that you are working in. It is assigned to you automatically.

ZONE see PRINT ZONE.

APPENDIX A
Debugging a Program

If you are having a particularly difficult time debugging a program, try these techniques:

1. Check for the most common errors.

- **a.** Did you type the letter O instead of a zero, or vice-versa?
- **b.** Have you used correct variable names for the data being read in or input? That is, names such as N, Y, S for numeric data and N$, X$, S$ for alphanumeric data.
- **c.** Did you accidentally change variable names in "mid-stream"? Did you read in N, for example, and try to print out M?
- **d.** Do you have more or fewer variable names in the READ or INPUT statement than you have data values in a set of data?
- **e.** Does the IF statement, testing for end-of-data, branch to just below the loop as it should, or does it jump directly to the END statement? If it does, no totals will print.
- **f.** Did you include a DIM statement for your alphanumeric variables? If so, did you dimension the proper variable name?
- **g.** Did you initialize all totals and/or counts?
- **h.** Did you leave out a comma (or add one that shouldn't be there) in a DATA statement? Did you accidentally omit a data value in one of the sets of data?
- **i.** Are you using the correct format for the statement concerned? Check the format and rules for the particular statement in Appendix E or in the main body of the text.

2. "Play computer" with one or two sets of data as we did in exercises in the book; that is, step through the program one line at a time with sample data values to see if it's doing what you want it to be doing.

3. Add temporary PRINT statements in the program at appropriate locations to print out one or more values. This will tell you, at any point in the program, the value of a specific variable, and will often reveal an error which would otherwise be difficult to detect. It helps you to isolate an error to a line or two of a program, hence speeding up the debugging process.

This technique will also tell you whether or not a certain section of the program is being performed. If the indicated value is not printed, you know that the computer is not reaching this section of your program. You must then retrace your logic, "playing computer".

4. If you want to test a program several times and don't want the preamble printing each time (such as title, headings, instructions, etc.), add a temporary GOTO statement at the start of the program:

```
1 GOTO 110
```

Be careful not to jump over any critical statements such as those initializing totals, DIM statements, etc.

5. To test only one or two sets of data instead of all those in the program, temporarily move the dummy set of data further "up" the data list.

6. Some computers have a TRACE statement or a TRACE command which allows you to trace a given variable as the program runs. For example, every time the program encounters the variable N, it will indicate this fact, and print out the value contained in it at that instant.

APPENDIX B
Built-in Functions
(Library Functions)

These are timesaving and/or informative functions which are available automatically with the computer. Many of them are of a mathematical nature. There are usually more available with larger computers than with the smaller ones, but even the smallest computers offer a fair number of them. Consult the manual provided with your system to determine exactly which ones your computer has.

Built-in functions can be called forth at any time in a program. We have already used one; namely, the TAB function.

Others include the following:

a. ABS (Absolute Value): Determines the magnitude (size, without regard to sign) of a specified variable; i.e., it drops any sign the number might have. For example,

```
N = ABS(W)

If W is 32.49, N will be 32.49
If W is -16.5, N will be 16.5
```

b. INT (Integer): This function truncates ("chops off") any decimal positions that a number may have, leaving only the integer value. Hence if the variable T had the value 33.899, the effect of

```
30 S = INT(T)
```

would be to put the value 33 into the variable S. It does not round the result. This function is often used to round arithmetic values, however. This is done as shown below. Consider the three simplest cases:

i To round to no decimals (i.e. to a whole number), use

```
X = INT (V + 0.5)
```

If the value of V were 45.6789, the result of the above expression would be to place the value 46 into X. This is because the computer first adds 0.5 to 45.6789, obtaining 46.1789. It then drops the decimal portion of this value, leaving 46.

ii To round to one decimal position (e.g., 33.6, 21.2, 8.0, etc.), use

```
X = INT (10 * V + 0.5)/10
```

If the value of V were 45.6789, this expression results in 45.6789 being multiplied by 10 first (456.789). 0.5 is then added giving 457.289, the integer value of this is found (457), and this result is divided by 10 to give a final answer of 45.7. This is the original value, rounded to one decimal position.

iii To round to two decimal postions (e.g., 34.25, 9.50, etc.), use

```
X = INT (100 * V + 0.5)/100
```

If the value of V were 45.6789, this expression results in 45.6789 being multiplied by 100 first (4567.89). 0.5 is then added giving 4568.39, the integer value of this is found (4568), and this result is divided by 100 to give a final answer of 45.68. This is the original value rounded to two decimal positions.

It isn't always necessary to round our results, but it is frequently required. This method may seem a little involved for what we accomplish, but it is the simplest way to do it.

c. LOG (Logarithm): Determines the natural logarithm of a number (base e); e.g.,

```
K = LOG(D)
```

d. SGN (Sign of a value): Determines if a value is positive, negative, or zero. This function returns a value of -1 if the value is negative, +1 if the value is positive, and 0 if the value is zero; e.g.,

```
Q = SGN(X)
```

e. SQR (Square Root): Calculates the square root of a value; e.g.,

```
P = SQR(V)
```

will calculate the square root of the value in V and place the answer in P.

f. RND (Random Number Generator): The random number generator is used for many games and/or simulations. It generates a random number between 0 and 1 in most computers; i.e., a decimal value such as 0.234567 or 0.968723, using a form such as

```
C = RND(0)
```

The "argument" (the number in parentheses) must always be the same. It may be -1, 0 or 1 depending on the computer that you are using and/or the numbers you wish to generate.

Some computers allow the form:

```
C = RND(300)
```

which would generate a random number (an integer) between 1 and 300 in this case.

If you use this function in a program, observe the results carefully. It may produce the same series of random numbers every time the program is run.

Most often, we want to generate whole numbers. The following form will work on most computers.

```
N = INT (RND(0) * R + L)
```

The R represents the range of numbers you want (i.e., how many) and the L represents the lowest one.

Thus, if we use N = INT (RND(0) * 22 + 6), the computer will generate random numbers from 6 to 27 inclusive. That is, it will generate integers, in a range of 22 numbers, the lowest of which is 6.

g. Trigonometric functions: SIN (sine of an angle); COS (cosine of an angle); TAN (tangent of an angle); ATN (arctangent of an angle), etc.

The angle must be expressed in radians for most versions of BASIC; e.g.,

```
B = SIN(P)
F = COS(K)
etc.
```

h. Miscellaneous informative functions: Larger computers typically offer a number of additional useful functions, including

i Date

ii Time of day

iii Connect time — how long you've been signed-on

iv Length of a program in "computer words" or bytes

v Length of a string value in memory.

Check the manual supplied with your computer to see which ones it offers.

APPENDIX C
Arrays

Although arrays in their complex form are beyond the scope of this book, simple arrays offer such an advantage in many instances that some knowledge of them is invaluable.

An array (or MATRIX) is simply a group of related numbers. If we have a single row of them they make up a ONE-DIMENSIONAL ARRAY. For example, the numbers 1500 600 1200 500 4400, which could represent January sales for individual branches of a company, form a one-dimensional array.

Values arranged in a table format create a TWO-DIMENSIONAL ARRAY. Officially, this is the only array that should be called a matrix. For example,

1500	600	1200	500	4400
400	700	500	1400	900
800	2400	4500	800	6500

Here, the numbers could represent the individual branch sales of a company for the months of January, February, and March.

For programming purposes, an array is a section of storage which contains several "boxes" stored under a *single variable name*. The separate "boxes", called ELEMENTS, are referred to using a SUBSCRIPT in brackets after the array name. Consider a one-dimensional array of six positions called A. The first position of A would be referred to as A(1), the second position as A(2), etc. Variables may be used for subscripts too.

The power of the array can best be illustrated by way of example. Consider the problem we looked at in Section 12-1, which was to total the sales for a company by branch. The program is repeated here for convenience.

```
5 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
6 DATA 3, 4000, 2, 600, 2, 7000
7 DATA 1, 6000, 4, 900, 3, 450
8 DATA 2, 1400, 0, 0
15 T1 = T2 = T3 = T4 = 0 (you may need separate lines)
17 REM T1 IS TOTAL FOR BRANCH 1, T2 FOR BRANCH 2, ETC.
20 READ B, S
30 IF B = 0 THEN 110
```

```
40 IF B = 1 THEN 80
50 IF B = 2 THEN 90
60 IF B = 3 THEN 100
70 T4 = T4 + S
75 GOTO 20
80 T1 = T1 + S
85 GOTO 20
90 T2 = T2 + S
95 GOTO 20
100 T3 = T3 + S
105 GOTO 20
110 PRINT
120 PRINT "TOTALS ARE :"
125 PRINT " BRANCH 1 : " ; T1
130 PRINT " BRANCH 2 : " ; T2
135 PRINT " BRANCH 3 : " ; T3
140 PRINT " BRANCH 4 : " ; T4
150 END
```

To produce the same results with an array, we could use the following:

```
5 REM . . . PROGRAM TO CALCULATE BRANCH TOTALS
6 DATA 3, 4000, 2, 600, 2, 7000
7 DATA 1, 6000, 4, 900, 3, 450
8 DATA 2, 1400, 0, 0
12 DIM T(4)
13 FOR N = 1 TO 4
14    T(N) = 0
15 NEXT N
17 REM . . . THE ARRAY T WILL HOLD BRANCH TOTALS
20 READ B, S
30 IF B = 0 THEN 110
40 T(B) = T(B) + S
50 GOTO 20
110 PRINT
120 PRINT "TOTALS ARE :"
125 PRINT " BRANCH 1 : " ; T(1)
130 PRINT " BRANCH 2 : " ; T(2)
135 PRINT " BRANCH 3 : " ; T(3)
140 PRINT " BRANCH 4 : " ; T(4)
150 END
```

Line 12 is a DIMENSION statement that reserves space in the computer memory for our array. Some computers automatically reserve at least 10 spaces for an array which would make this statement unnecessary. It is a good idea to get into the habit of including it in any case. In our case we are keeping track of four totals, so we will need four spaces reserved for the array. This is accomplished by the 4 in brackets after the array name in the DIM statement.

Lines 13, 14, and 15 zero each position of the array, one at a time. We will be keeping totals in our array (total sales); and the rule that totals must be initialized (zeroed) still holds true. (Some computers do not require this initialization.)

Line 40 of the program adds the sales (S) to the correct branch total.

```
40 T(B) = T(B) + S
```

Note that line 40 conforms to the format that is required for a total; i.e.,

T = T + A

Hence, we can conclude that T(B) must be "where" the total is being kept, and the S (sales) must be "what" is being totalled.

T(B) will represent the B position of the array T. No matter which branch is involved, the sales will be added to the appropriate total. (B is the branch number read in.) We've thus eliminated several lines from our previous program through the use of an array.

An alternate method of printing the values in the array is with a FOR-NEXT loop:

```
130  FOR N = 1 TO 4
135  PRINT "BRANCH # ";N ;T(N)
140  NEXT N
```

The true power of the array is revealed, though, when there are a large number of variables required.

Consider, for example, the same company with 125 branches instead of just four.

Without using an array, we would need 125 IF statements to check for branch numbers and 125 variable names to keep the individual branch totals in (T1, T2, etc. etc.).

With an array, however, it couldn't be easier - we simply change the DIMENSION statement, the FOR statement that zeroes the array and the FOR statement of the loop that prints out the values in the array:

```
12 DIM T(125)
13 FOR N = 1 TO 125
130 FOR N = 1 TO 125
```

And that is all!

The statement in the loop that adds the sales to the branch totals:

```
40 T(B) = T(B) + S
```

still does exactly what it is supposed to - but it now adds the total sales for 125 branches instead of only four, as it did before. Run the program with higher branch numbers in the data.

Two-Dimensional Arrays Suppose our fictitious company has three divisions with four branches in each one.

Sales data for our company would now take the form:

Division	Branch	Sales($)
3	2	600
1	4	700
2	3	1200
3	1	1000
1	4	800

Assuming that we still wish to keep track of our branch totals, we will now need 12 separate totals. The easiest way to do this is again through the use of an array.

To set up our two-dimensional array in the computer we should visualize it as follows:

	Column 1	Column 2	Column 3	Column 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

This array consists of 3 rows of 4 columns, but we could have used 4 rows of 3 columns had we wished. The way I've set it up, the rows correspond to divisions and the columns to branches.

In order to dimension the above array in a program, use the form

```
12 DIM T(3, 4)
```

The first number within the brackets must be the number of ROWS (horizontal) in the array, the second one the number of COLUMNS (vertical).

The READ statement might be

```
30 READ D, B, S
```

where D is division number, B is branch number, and S is sales in dollars.

The statement within the loop to add the sales to the proper branch total would now be

```
40 T (D, B) = T (D, B) + S
```

To examine this, let's use a set of sample data. Suppose the program has read in the following DATA statement.

```
200 DATA 3, 2, 600
```

When line 40 is reached, the computer locates the array T, and sees that the sales S are to be added to position T (D, B) of the array. It is important to appreciate that the computer does not think of D and B as being division and

branch. Rather, they are interpreted by the computer as being the row number and the column number concerned.

Since it is always (row, column) within the brackets, the computer will look for the entry at the intersection of row 3 and column 2 in our example. This is because D is 3 and B is 2 for our sample data and the computer is using position T(D, B) of the array.

Using this approach, the sales will always be added to the correct total.

The PRINT statements would also have to be changed.

MAT Statements There are a number of special BASIC statements associated with arrays called MATRIX statements. These offer an efficient means of reading data into an array, printing out the contents of an array, or performing calculations using the values in an array.

Many computers do not allow their use but those that do will probably permit the following:

a. MATREAD

This statement will read enough data values to fill the array indicated; e.g.,

```
30 MATREAD A
```

will determine the size of the array A (by examining the DIMENSION statement) and will read the necessary number of data values from DATA statements. It will then fill the array on a row by row basis.

b. MATPRINT

This statement will print the contents of array A. It will print all values contained in the array, one row at a time.

c. Arithmetic Operations

```
125 MAT R = A + L
```

will add the corresponding elements of arrays A and L together and place the results in array R.

All three arrays must be identical in form, having the same number of rows, and columns.

The other arithmetic operations are also permitted.

```
120 MAT A = (C) * K
```

will multiply each element in the array K by a constant C.

This is useful if you wish to show the effect of a certain percentage growth rate for all entries in an array.

There are other MAT statements as well including ones which allow inverting of a matrix, forming an identity matrix, etc. I would direct those of you interested to your computer manual.

Notes on Arrays Individual values in an array may be "read into", printed, or used in calculations as any ordinary variable may be; e.g.,

```
40 READ A(3, 6)
160 PRINT B(5), C(3, 6)
200 X = H(2, 3) * A(R, 1) + 10/C
165 P = A(I, J) * N(D)
```

Arrays are very powerful but they can also become very complicated and confusing, particularly for the novice. In their more complex form, they are capable of much more than I might have implied here.

Nonetheless, even if you only master them at the level we have discussed I think you will find them very helpful.

APPENDIX D Comparison of BASIC Dialects

There are many similarities and some differences amongst the BASIC dialects offered in the marketplace.

Rather than be specific, I have indicated here where the similarities and differences are most likely to be. You must consult the manual associated with your computer for specific rules, formats, etc.

A. BASIC Statements

Group 1 The following statements/functions are identical in form and usage on almost all computers.

Assignment statements (with or without the word LET)
END
FOR/NEXT
GOTO
IF (basic form; e.g. 110 IF K = 0 THEN 200)
PRINT (basic forms; e.g., 80 PRINT D$, H)
READ/DATA
REMark statement

Group 2 These statements differ from one computer to the next.

a. INPUT statement — minor differences; e.g.,

INPUT A; J vs INPUT A, J
(semi-colon vs comma)

Some computers allow the form:

INPUT "ENTER AGE AND DATE ", A, D

Others require INPUT "ENTER AGE AND DATE "; A, D

b. DIMENSION statement

Some computers require this statement for all alphanumeric data that will be longer than one position. Others only require it for those longer than 10 positions or some such value.

c. ON-GOTO statements
Some computers use the "GOTO-N-OF" form of this statement.

Group 3 Many computers do not allow any form of the following statements:

a. PRINT USING/IMAGE

b. IF-THEN-ELSE (see Chapter 12)

c. IF-THEN-ELSE-DO-DOEND (see Chapter 12)

Group 4 Most versions of BASIC do not permit the use of these statements. They are used in structured programming. In the future, more and more versions of BASIC will allow their use.

They are, in fact, sets of statements, called STRUCTURES or CONSTRUCTS. They might include the following:

a.
```
WHILE A < 12                DO
  PRINT A                     PRINT A
  A = A + 1          or       A = A + 1
END WHILE                   WHILE A < 12
```

which will perform the indented lines until A is equal to 12.

b.
```
REPEAT
  PRINT A
  A = A + 1
UNTIL A > 12
```

which will repeat the indented lines until A is greater than 12.

B. Other Differences

Multiple Statements Some computers allow several statements to be combined on one line. The most common method for this is to separate individual statements with colons; e.g.,

```
10 PRINT : B = Q * W : PRINT Q, W, B
```

is equivalent to

```
10 PRINT
20 B = Q * W
30 PRINT Q, W, B
```

This is very helpful for inserting lines in an existing program. Keep in mind, however, that a program using multiple statements tends to be confusing to read at times. Also, you can only branch to the first statement on the line, since the other(s) do not have statement numbers.

Variable Names Some computers allow much longer names than we have been using in this text. For example, your computer may allow the following:

```
10 READ NUMBER, DEPT, WAGE, HOURS
20 PAY = HOURS * WAGE
30 PRINT NUMBER, DEPT, PAY
40 DATA 3333, 4, 6, 40
50 END
```

Line Renumbering Some computers allow you to renumber the lines of your program beginning at some specified line number, and increasing by a specified increment.

This is very helpful when you have used all the available lines between certain line numbers and you want to insert even more. The usual form is a command, such as RENUM.

When you use this command it will automatically adjust any IF statements or GOTO statements to reflect the new numbering. Some computers use the command RESEQUENCE to accomplish the same task.

Random Number Generator Some computers use RND(0), while others use RND(1) to generate a number between 0 and 1.

Some allow the form RND(N) where N is any number. The computer will then generate a random number between 1 and N.

Usually, the random number is placed in a variable; e.g.,

```
60 X = RND(0)
```

(See Appendix B for details.)

Miscellaneous Functions Some computers will print the date, time of day, connect time, computer time used, etc. upon request.

Miscellaneous Features There are a number of features which are available on most of the newer computer systems today. These might include the following:

a. Graphics – There are a number of computers that have powerful graphics capabilities. These allow the user to draw graphs, pie charts, circles, or elaborate diagrams. Some allow color as well. Special statements are included with these versions of BASIC to permit this.

b. Sound – Many computers today allow the use of sound in programs. By issuing the proper instructions these sounds (electronic tones) can be produced. An example of their use in business would be in an environment where users are busy. A program could call certain output to their attention by sounding a tone(s).

c. Manipulation of Data in Memory – Special statements such as PEEK and POKE allow this on some computers. These let you insert a specific value into

a specific location in the computer memory (POKE) or to examine one that is there (PEEK).

d. Manipulation of Data Strings – Certain statements allow us to extract certain portions of a string (alphanumeric data) such as the right-hand side, left-hand side, or some central portion of it.

Functions such as MID$, LEFT$, RIGHT$, etc., permit this.

e. Abbreviations are permitted by some computers for certain keywords in statements. For example, in some versions of BASIC, a question mark can be used instead of typing the word PRINT.

f. Some computers have special keys for certain commonly used commands; e.g., a RUN key, a LIST key, etc.

APPENDIX E
Text Summary

BASIC Commands

1. RUN to run a program and produce the output.
2. LIST to see what the program looks like.
   a. LIST 80 will list line 80 only.
   b. LIST 20 – 85 will list lines 20 through 85.
3. Retrieving a program from disk or tape. Depending on the computer, one of the following will probably be used:
   a. GET name
   b. LOAD name
   c. CLOAD "name"
   d. DLOAD "name"
   e. OLD
   etc.

   The name of the program you wish to retrieve must be substituted for the word "name" above.
4. Saving a program for later use on tape or disk. Depending on the computer, one of the following may be probably used:
   a. SAVE name
   b. CSAVE "name"
5. NEW (or CLEAR or SCRATCH)
   To clear your workspace before you begin to type a program.

BASIC Statements

1. DATA
Purpose: To hold data values until they are read by a READ statement; e.g.,

```
200 DATA 83, 100, "TELEVISION", 0.12
```

Common errors: i Including a comma in a data value;
i.e., should be 55000 and not 55,000
ii Forgetting a comma between values.
iii Adding a comma at end of the line.

2. END

Purpose: To signify the end of a program; e.g.,

```
250 END
```

3. FOR-NEXT

Purpose: To control how many times a statement or group of statements is executed; e.g.,

```
60 FOR B = 1 TO Q
70 PRINT B
80 NEXT B
```

4. GOSUB

Purpose: To branch to a subroutine (a statement or group of statements) elsewhere in the program. After the subroutine has been executed, control will return to the line below the one containing the GOSUB statement.

5. GOTO

Purpose: To branch to another line of a program; e.g.,

```
100 GOTO 75
```

6. IF

Purpose: **a.** To branch to another line of a program if a specific condition is true; e.g.,

```
110 IF C = 0 THEN 260
```

b. To determine whether or not a calculation is to be performed (not available on all computers).

```
125 IF Q < 10 THEN K = F * D/3
```

Common error: Branching to the line immediately below the IF statement; e.g.

```
50 IF A = 2 THEN 60
60 . . .
```

The program will examine line 50. It will then go to line 60 no matter what. If A = 2, it will go to line 60 as it is directed to, but if A is not equal to 2, it will simply "fall down" to the next line (line 60). Thus, the IF statement has served no purpose whatever. In other words, the program would run the same with or without line 50 in it.

7. INPUT

Purpose: To accept input data via the keyboard (a ? is printed out by the computer); e.g.,

```
80 INPUT N$, B, J or 80 INPUT N$;B;J
```

8. LET

Purpose: **a.** To assign a value; e.g.,

```
45 LET R = 0
```

b. To perform calculations; e.g.,

```
100 LET W = (C * 50)/V - (R + L/6)
```

Most computers allow you to omit the word LET. Some produce incorrect results if you use it!

Symbols for calculations : + for addition
- for subtraction
/ for division
* for multiplication
^ or ↑ or ** for exponentiation

Common errors: i Mismatching parentheses - there must be a right-hand parenthesis) for every left-hand one (.
ii Forgetting a multiplication symbol (*) between values being multiplied.

9. ON-GOTO

Purpose: To control which of several statements the program is to branch to. May sometimes be used in place of IF statements; e.g.,

```
80 ON T GOTO 110, 140, 200, 65
```

10. a. PRINT

Purpose: To print headings, results, etc.

```
110 PRINT X, B$, K
150 PRINT "SALES REPORT"
200 PRINT TAB(3); X; TAB(12); N$; TAB(20); P4
70 PRINT
80 PRINT C; J$; Z; D
```

Common errors: i Adding a comma at the end of a line.
ii Using commas and/or semi-colons incorrectly.
iii Forgetting quotation marks for headings.

b. PRINT USING

Purpose: To format printed results. (Not available on all computers.) (See Chapter 17.)

11. READ

Purpose: To read data from a DATA statement; e.g.,

```
40 READ N$, P, M7
```

Common errors: i Adding a comma to the end of the line.
ii Including quotation marks in the READ statement.

12. REMARK

Purpose: To add comments to a program; e.g.,

```
10 REM . . . D.C. RENNIE . . . PROBLEM 4-5
50 REM I IS ITEM NO. / P IS PRICE
```

13. RETURN

Purpose: This statement signifies the end of a subroutine. It returns program control to the line below the one containing the GOSUB; i.e., the one that sent control to the subroutine initially.

14. STOP

Purpose: To end a program before the END statement. It is often useful when using subroutines or in programs where the program can end in more than one way.

Miscellaneous

a. Data Types
- i Numeric – consist of digits, and possibly a sign (+ or -) and/or a decimal point.
- ii Alphanumeric (string) – includes letters, symbols, numbers, etc. It cannot be used in calculations.

b. Variable names
- i For numeric data – a single letter; e.g., T, K
  – a single letter plus a digit; e.g., R5, K8
- ii For alphanumeric data – a single letter followed by a $; e.g. R$, H$

c. Keeping a Columnar Total
- i Initialize the total in the opening section of the program; T = 0
- ii Add to it in the loop; T = T + A
- iii Print it out in the closing section

```
PRINT "TOTAL IS "; T
```

d. Keeping a Count
i Initialize the count in the opening section of the program; C = 0
ii Add to it in the loop; C = C + 1
iii Print it out in the closing section

```
PRINT "TOTAL COUNT IS "; C
```

e. Calculating an Average
i Keep a total of the desired value. (As in **c.**)
ii Keep a count of how many of the values in i are read. (As in **d.**)
iii Divide i by ii for the average. The total and count must both be performed as indicated above. Step iii should be performed in the closing section of the program (even though you will still obtain the correct results with it in the loop).

f. End-of Data Test
i Add a dummy set of data e.g. 245 DATA 0, "X", 0, 0
ii Test for the dummy set of data with an IF statement; e.g.,

```
70 IF K = 0 THEN 200
```

g. Altering a Program
i Adding a line – Choose a number between the two line numbers where you wish to insert a line.
ii Deleting a line – Type the line number and press RETURN (ENTER).
iii Changing/replacing a line – Type the new version with the same line number as the old.

h. Renumbering the Lines of a Program
(not available on all computers)
Type the command RENUM and press RETURN (ENTER). The command is RESEQUENCE with some computers.

i. Saving a Program
Type the command for your computer, and the program name (PROB1 in the examples below). The command usually uses the keyword SAVE, and the program name that you wish to save the program under must usually be within quotation marks; e.g.,

```
SAVE "PROB1"
or CSAVE "PROB1"
or DSAVE "PROB1"
```

j. Retrieving a Program
Typical commands used to retrieve a program are:

```
GET PROB1
or LOAD PROB1
or CLOAD "PROB1"
```

(assuming a program name of PROB1)

APPENDIX F
Reference Sheets

Note the method that your computer uses to do the following tasks (as you learn them).

a. SIGN-ON procedure:

SIGN-OFF procedure:

b. CLEAR your workspace:

c. LIST a program i All of it: _____

ii Part of it: _____

_____

d. RUN a program: _____

e. STOP a program as it is running: _____

RESTART a program that you stopped: _____

f. ADD a line to a program: _____

DELETE a line from a program: _____

CHANGE a line of a program: _____

g. NAME a program: _____

h. SAVE a program for later use:_____

_____

i. RETRIEVE a saved program:_____

j. i Number of PRINT ZONES:_____
Print positions per zone_____

ii VARIABLE NAMES: (numeric)_____
(alphanumeric)_____

iii PRINT POSITIONS per line (CRT)_____
(Printer)_____

iv SIGNIFICANT DIGITS (normal)_____
(extended)_____

Index

ISBN 0-13-729830-7